Language-Based Verification Will Change The World

Tim Sheard
Computer Science
Portland State University
Portland, OR, USA
sheard@cs.pdx.edu

Aaron Stump
Computer Science
The University of Iowa
Iowa City, IA, USA
astump@acm.org

Stephanie Weirich
Computer and Information Science
The University of Pennsylvania
Philadelphia, PA, USA
sweirich@cis.upenn.edu

ABSTRACT

We argue that lightweight, language-based verification is poised to enter mainstream industrial use, where it will have a major impact on software quality and reliability. We explain how language-based approaches based on so-called dependent types are already being adopted in functional programming languages, and why such methods will be successful for mainstream use, where traditional formal methods have failed.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Applicative (functional) languages; F.3.1 [**Logics and Meanings of Programs**]: Mechanical verification; F.4.1 [**Mathematical Logic**]: Mechanical theorem proving

General Terms

Languages, Verification

Keywords

Language-Based Verification, Dependently Typed Programming

1. A PARADIGM SHIFT IN THE MAKING

Generations have passed since the early days of verification and formal methods (e.g., [9]), and one could be excused for thinking that the at-times acrimonious debate is over: formal methods can provide strong guarantees about the functional correctness of software, some might concede, but they are too costly for mainstream use. If used at all, it will only be for the most safety-critical (and small) components of systems. Thus, they will have no real impact on the never-ending crisis of software, which researchers have grown so accustomed to that entire fields are premised on its insolvability [1].

There are undeniable aspects of truth to this view. The idea of even specifying programs like the Firefox web browser or Microsoft Office is daunting in the extreme. And full functional verification of even quite modest versions of realistic programs is still considered an almost incredibly difficult *tour de force*, worthy of publication in the most selective venues (e.g., [17, 13, 16]). Any company whose business depended on carrying out full functional verification of large pieces of industrial software would have a Hobbesian existence: not just short, but also nasty and brutish.

So why do we believe not only that there is hope for mainstream industrial application of formal methods, but that verification is poised to change the world? Because the prototypical *tour-de-force* (TDF) examples of verification in the literature are not representative of the much broader class of verification activities, which are suitable for mainstream adoption. We will explore this point by considering an example of such a TDF verification effort, namely the recent work on seL4, a fully verified operating-system kernel [13]. This work received an SOSP best-paper award, and is widely regarded as one of the most impressive formal verifications ever reported in the literature. We will explain why thinking of a TDF verification like that of seL4 as representative involves two serious misunderstandings: what we call the whole-system and the full-correctness fallacies. We will then argue that solutions being developed in the programming languages and computation logic communities – in particular, language-based verification methods using *dependent types* – are enabling a new approach to verification, which is free from these problems. We conclude by considering the research challenges in software engineering and programming languages that must be addressed in order to realize the vision of mainstream verified software.

2. TOUR DE FORCE VERIFICATION

In [13], Klein et al. describe the full functional verification of the seL4 OS kernel, written in 8700 lines of C and 600 of assembly. They prove, in a staggering 200,000 lines of formal proof in the Isabelle theorem prover, that the kernel refines an abstract, underspecified model of the kernel.[1] A critical component of the verification is an intermediate model of the seL4 operating system, written in a simple fragment of the functional programming language Haskell. This model, which is automatically translated to a model in the Isabelle theorem prover, helps the authors

[1]We conjecture that the ratio of effort required per line of proof versus line of mainstream imperative code is something like 10 to 1, making this the equivalent of 2 million lines of C code (of a kind which requires a very strong or highly trained engineer to write).

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

to bridge the gap between ISABELLE's logical language, and the low-level C programming language.

This very impressive work would seem to confirm the verification skeptic's position: verification, however valuable, is just too costly to use for mainstream programming. But let us consider this work more carefully. As is typical for a TDF verification, in the seL4 verification, the authors prove a very strong specification, in this case, the strongest currently imaginable: the kernel refines a high-level abstract specification of what an OS kernel should do. So their work shows that seL4 is as fully correct as we know how to specify an OS kernel to be. And proving full correctness ultimately requires reasoning about the entire 8700-line C program as a whole. Certainly much of such a proof is concerned with proving simpler properties of smaller units of the whole system. But the statement of the final theorem refers to the entire C-code kernel, and hence its proof must reason, in however elegantly modularized a manner, about the system as a whole.

3. LANGUAGE-BASED VERIFICATION

Our belief in the imminent industrial adoption of formal methods is based on rejecting the idea that verification requires whole-system reasoning and full correctness. Instead, like many authors in the programming-languages literature, we maintain that verification of specifications which fall far short of full correctness, carried out without whole-system reasoning, can provide tremendous benefits for software quality and reliability. As we will explain, these lightweight verification methods are conceptually much closer to existing programming methodology than other formal methods, thus lowering the barrier to adoption in practice.

The methods we have in mind are **language-based**: they avoid whole-system reasoning by extending the programming language with annotations for specifications and proofs, which are written as integral parts of the program text. No theorem need ever be written about the system as a whole; such global properties may be entailed by the weaker local invariants that are machine-certified, but that entailment itself is not machine-certified. Language-based methods have been developed in both the object-oriented and functional programming communities. For examples of the former, see the work on algorithmic verification of object-oriented programs (e.g., [4, 18]); and also approaches such as design by contract, which emphasize the benefits of formal specification (and focus less on formal verification of such specifications) [6, 21]. Here we will consider functional programming with dependent types, the examplar of this approach to lightweight verification which we know best. Recent examples of dependently typed functional languages (besides several of our own) include Agda, Epigram, Ynot, and several others [22, 23, 19]. HASKELL also supports a limited form of dependent types via so-called Generalized Algebraic Datatypes (GADTs) [24].

4. TYPE CHECKING AS VERIFICATION

Formal verification is used every day by industrial programmers in the form of type checking. Strong static type systems in industrial languages like JAVA and C#, and in languages like HASKELL and OCAML used mostly (but not exclusively) in academia, are designed to rule out simple kinds of program errors at compile-time, such as calling a floating point operation on non-numeric arguments, or using an integer as a function pointer. In combination with certain runtime checks (e.g., for in-bounds array accesses), these type systems guarantee that at runtime, type-correct programs will respect the abstractions expressed by the language's type system. Thus, programs that have passed the type checker have been verified, albeit with respect to relatively weak specifications (i.e., types). The basic idea of the approach we advocate is to extend the language's type system, so that semantically richer abstractions can be described and enforced at compile-time.

Let us consider a simple functional-programming example. In languages like HASKELL or OCAML, we may write a function to append two lists, with the following type:

```
append : list 'a -> list 'a -> list 'a
```

This type says that the `append` function takes in two lists storing elements of any type `'a`, and returns a list of elements of type `'a`. Static type checking guarantees that this function, if called with lists whose elements are all of some common type, will return such a list, if the function indeed terminates normally (i.e., in finite time and without raising exceptions).[2]

Dependently typed programming languages allow us to describe richer abstractions using types. This is done with type expressions that themselves contain data expressions. A commonly used example is the type `list 'a 'n` of lists with length. Here, the additional argument `'n` to the `list` type-constructor is a natural number specifying the length of the list. So we have the following typing (assuming suitable notation for string and list literals) for this example list:

```
[ "Santa" , "Fe" , "NM" ] : list string 3
```

The constructors for lists with length are:

```
nil : list 'a 0
cons : 'a -> list 'a 'n -> list 'a ('n+1)
```

The empty list (`nil`) naturally has length 0, and a list built with `cons` will have length `'n` + 1, if the sublist it is given has length `'n`. Finally, and more interestingly, it is possible to write an append function with the following type:

```
append : list 'a 'n -> list 'a 'm -> list 'a ('n+'m)
```

This type expresses a non-trivial semantic property of `append`: given a list of length `'n` and a list of length `'m`, the list returned by `append` (assuming `append` terminates normally) will have length `'n` + `'m`. This property is not one that can be expressed or checked in traditional type systems for functional programming languages, or in industrial languages.

There is, of course, a lot more to writing verified code than just specifying a property like this one between the lengths of the input and output lists. We will explain more about the costs and challenges of dependently typed programming, after we highlight its advantages over other verification methods.

[2]Static type checking does not enforce normal termination in languages like HASKELL and OCAML: programs can still run forever or raise an exception. We note in passing that (compile-time) verification of termination turns out to play an important role in dependently typed languages (see, e.g., [26]).

4.1 Benefits of dependent types

The chief benefit of dependent types for verification is the very small conceptual and engineering gap between (functional) programming as it is usually done, and dependently typed programming. For a programmer without special training in logic or theorem proving, the intuitive meaning of the type `list 'a 'n` should be, we believe, easy to grasp, based on the meaning of `list 'a`. To understand and program with lists with length, there is no need to learn a separate specification language, special modeling techniques, or a separate verification toolset, as there is with all other formal methods we are aware of. The specification language is just the type system of the programming language, and the program verifier is just the type checker. The method applies directly to source code, not a separate modeling language. The power of language-based verification comes from the tight integration between programming, specifying, and verifying. Indeed, for dependently typed functional programming, those three activities are supported through a single language (i.e., the dependently typed programming language) with a single tool (the compiler).

This small step from programming-as-usual to verified programming is the chief reason we believe dependently typed programming will be widely adopted in the near future, certainly within academia, but also trickling down to industry (much as features like closures – a central idea from functional programming – are being developed, as of this writing, for version 1.7 of the Java language). Indeed, a form of dependently typed programming is already available, thanks to GADTs in recent versions of HASKELL. Other benefits of verified programming with dependent types, which we cannot explore here for space reasons, include the ability to type check highly generic programs (e.g., a single function `map` that can map a given n-argument function over tuples of corresponding elements from n given lists, collecting the list of outputs); and also a more expressive specification language than traditional logics, thanks to richer typing features (e.g., type-level recursion).

4.2 Challenges

There is, of course, no free solution to the problem of verified programming. With dependent types, the cost manifests itself when the type checker is unable to determine automatically that two type expressions are equivalent. For example, suppose we have obtained a list `L` of type `list 'a ('n+'m)` by calling our `append` function. We can easily find ourselves in a situation where we need the type checker to confirm that `L` can also be viewed as having type `list 'a ('m+'n)`. While knowledge of properties like commutativity of addition can be built into the compiler as a special case (as done in Dependent ML [28]), we cannot do this for all functions programmers might wish to include in type expressions. Several solutions to this problem have been proposed in the literature. With *hybrid type-checking*, equivalences which cannot be checked statically are turned into runtime checks by the compiler [10]. Another approach requires the programmer himself to include proofs in his code, where necessary to convince the type checker that types are equivalent. For our example, this proof would be essentially a proof of commutativity of addition.

So it might seem we have come full circle, back to the onerous writing of proofs. Two things save our proposed approaches, however, from becoming just another form of TDF verification. First, dependent types provide better supoprt for incremental, pay-as-you-go verification than traditional methods. Adding modest specificational information like the length of lists will not usually impose a large proof burden. This is because the only proofs required are ones about the specificational information appearing in types (like the proof that `'n + 'm = 'm + 'n`). This does not require proving anything about the surrounding program which is operating on lists with length. So the whole-system problem is avoided: we reason not about our code, but about the specificational functions and data appearing inside type expressions for that code. Certainly, if one wishes to prove something about a complex specificational function, or about some function from one's implementation, then that is also possible in dependently typed languages. But one must then be prepared for the burden of proof, which can be very high. By rejecting the idea that the only verification worth the name is full-correctness verification, dependently typed programming opens up an exciting continuum, where richer specifications lead one in a gradual way towards more complex forms of reasoning. It is up the programmer to decide how far to go. It may be that one can delineate certain patterns of specification which ensure that all necessary proofs can indeed be found fully automatically. If one goes beyond these patterns, then proofs may be required. But that is not a limitation: proofs are generally required at a certain point for deep reasoning about correctness, under any approach. Dependently typed languages provide a principled way to integrate such proofs directly into programs, when needed.

5. PL AND SE

Programming Languages (PL) and Software Engineering (SE) are two fields that, at least conceptually, need each other. Yet their interactions are relatively few. We believe that verified programming with dependent types opens up a new avenue for possible interdisciplinary research between the fields. This is largely due to the fact that with the introduction of dependent types, traditional software engineering processes will become even more critical for effective programming, and even more technically challenging. For example, consider restructuring and refactoring [20]. Type-preserving restructuring will be, we conjecture, significantly more involved for languages with dependent types, since the types impose strong constraints on the space of legal programs into which a starting program might be restructured. The very (presupposed) difficulty of this problem means that it will be even more error-prone and difficult to carry out by hand than for languages with traditional type systems, thus increasing demand for automated restructuring tools.

Verified programming with dependent types also adds another hard-to-predict variable to the problem of allocating resources for large software projects. How should a project decide how much of its budget to spend on lightweight verification, and when should and how should it spend it? Is it better to start with dependent types expressing certain carefully chosen invariants, or is that the verificational equivalent of optimizing without profiling (i.e., should we wait to add this kind of rich type information until we have some evidence that a certain subsystem of the code is difficult to get right)? Questions like these may cover well-trodden ground, but the incremental, pay-as-you-go nature of verification with dependent types should, we conjecture, enlarge the range of possible answers. How to balance the increased

complexity of verified programming with the increased rewards of correctness appears to us, as outsiders to the field, to be a challenging SE problem.

6. OTHER FORMAL METHODS

TDF verification techniques like *theorem proving* are able to achieve full correctness of software, hardware, and other systems, but at high cost in labor and training (see, e.g., [3, 14] for other examples). Weaker deductive techniques like *model checking* are automatic, but generally work only for finite state systems, and can be computationally expensive [7]. Powerful automated theorem-provers like SAT/SMT solvers are increasingly used as backends for model checking and other automatic verification methods (see competition reports like [5, 15] for results and further references). Indeed, advanced static analysis techniques for (concurrent and sequential) programming languages and compilers continue to rely on automated solving for different logics [27, 11]. Impressive progress has been made applying automatic-verification techniques for static bug finding [29]. Abstract interpretation and standard type checking provide relatively efficient approaches to code validation, though for less general problems [8, 25]. Testing and run-time monitoring remain mainstays of correct program development in practice [12, 2].

In contrast to these methods, language-based verification seeks to provide incremental support for a full continuum of correctness, from weak properties that are easily verified automatically, possibly using solvers like SAT or SMT solvers; all the way to full correctness proofs requiring manual theorem proving. Dependently typed programming makes this power available to the programmer by a natural extension of the one language in which the programmer is guaranteed to be fluent: the programming language itself. This opens up the possibility that strong guarantees about software can truly be obtained in an incremental manner, without the steep learning curve and additional tool sets required by other methods.

7. CONCLUSION

We have argued that language-based, lightweight verification – embodied, in the functional programming paradigm, by dependent types – will change the world. This is a prediction, not a prescription. The problems of correct software are so pressing, and the dependence of society on software is so great, that given the right technical means, verified programming will become a widespread reality in mainstream industrial programming. The fallacies of whole-system verification and full functional correctness, which have restricted verification largely to *tour-de-force* examples, need hold us back no longer. Language-based verification, particularly dependently typed programming, can enable the gradual adoption of formal methods, based on increasingly semantically rich types. The present ubiquity of type systems will inexorably give rise to the future ubiquity of verified programming. How we use the resulting expressive power is likely to require a joint effort of both the Programming Languages and Software Engineering fields.

Acknowledgments. This work is part of the TRELLYS project (NSF award 0910510), which is seeking to design and implement a next-generation dependently typed functional programming language, using a community-based design process. We thank other members of the TRELLYS team for many helpful conversations about dependent types and verified programming.

8. REFERENCES

[1] *The 7th IEEE/ACM International Conference on Autonomic Computing and Communications*, Washington, D.C., 2010.

[2] J. Andrews. General Test Result Checking with Log File Analysis. *IEEE Transactions on Software Engineering*, 29(7), July 2003.

[3] D. Aspinall and J. Sevcík. Formalising Java's Data Race Free Guarantee. In K. Schneider and J. Brandt, editors, *20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 22–37, 2007.

[4] M. Barnett, B-Y. Chang, R. DeLine, B. Jacobs, and R. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. de Boer, M. Bonsangue, S. Graf, and W. de Roever, editors, *Proceedings of the 21st International Symposium on Formal Methods for Components and Objects*, number 4111 in Lecture Notes in Computer Science, pages 364–387. Springer-Verlag, 2008.

[5] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and Results of the 3rd Annual Satisfiability Modulo Theories competition (SMT-COMP 2007). *International Journal of Artificial Intelligence Tools*, 17(4):569–606, 2008.

[6] L. Burdy, Y. Cheon, D.R. Cok, M.D. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[8] P. Cousot. The Verification Grand Challenge and Abstract Interpretation. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science, pages 227–240. Springer, 2007.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] C. Flanagan. Hybrid Type Checking. In G. Morrisett and S. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[11] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–292. ACM, 2008.

[12] M. Jorde, S. Elbaum, and M. Dwyer. Increasing Test Granularity by Aggregating Unit Tests. *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008*, pages 9–18, 2008.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[14] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[15] D. Le Berre and L Simon, editors. *Special Issue on the SAT 2005 Competitions and Evaluations*, volume 2, 2006.

[16] X. Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In S. Peyton Jones, editor, *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[17] J. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a Verified Relational Database Management System. In J. Palsberg, editor, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[18] 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.

[19] C. McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[20] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[21] B. Meyer. Eiffel*: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[22] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 229–240. ACM, 2008.

[23] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[24] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. Reppy and J. Lawall, editors, *International Conference on Functional Programming (ICFP)*, pages 50–61, 2006.

[25] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[26] A. Stump, Vilhelm Sjöberg, and S. Weirich. Termination Casts: a Flexible Approach to Termination with General Recursion. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Workshop on Partiality and Recursion in Interactive Theorem Provers*, 2010.

[27] T. Terauchi. Checking race freedom via linear programming. *SIGPLAN Not.*, 43(6):1–10, 2008.

[28] H. Xi. Dependent ML An approach to Practical Programming with Dependent Types. *J. Funct. Program.*, 17(2):215–286, 2007.

[29] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.