Language Design Meets Verifying Compilers (Keynote)

David J. Pearce
david.pearce@consensys.net
ConsenSys
Brooklyn, New York, USA

Abstract

The dream of developing compilers that automatically verify whether or not programs meet their specifications remains an ongoing challenge. Such "verifying compilers" are (finally) on the verge of entering mainstream software development. This is partly due to advancements made over the last few decades, but also to the increasingly significant and complex role software plays in the modern world. As computer scientists, we should encourage this transition and help relegate many forms of software error to the history books. One way of increasing adoption is to design languages around these tools which look, on the surface, like regular programming languages. That is, to seamlessly integrate specification and verification and offer something that, for the everyday programmer, appears as nothing more than glorified type checking. This requires, amongst other things, careful consideration as to which language features mesh well with verification, and which do not. The design space here is interesting and subtle, but has been largely overlooked. In this talk, I will attempt to shed light on this murky area by contrasting the choices made in two existing languages: Dafny and Whiley.

CCS Concepts: • **Theory of computation → Program reasoning**; • **Software and its engineering → Compilers**.

Keywords: Verifying Compilers, Formal Methods

ACM Reference Format:
David J. Pearce. 2022. Language Design Meets Verifying Compilers (Keynote). In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3564719.3570917

1 Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been well studied. Hoare's Verifying Compiler Grand Challenge was an attempt to spur new developments in this area [37]. According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [37]. Hoare's intention was that verifying compilers should fit into the existing development tool chain, "*to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components*".

Early systems that could be considered as verifying compilers include that of King [44], Deutsch [25], the Gypsy Verification Environment [32] and the Stanford Pascal Verifier [57]. The Extended Static Checker for Modula-3 [24] followed. Later, this morphed into the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work [31]. Numerous others have blossomed in this space, including Spec# [4, 5, 28, 56], Dafny [49, 51–54], Why3 [8, 29, 30], JML [11, 16–18, 48, 73], VeriFast [41, 42], SPARK/Ada [59], AutoProof for Eiffel [77], Frama-C [21, 33, 45, 78], KeY [1], Viper [35, 61, 76], SPARK/Ada [3, 13], and Whiley [14, 65, 67, 68]. Such tools build upon Hoare logic [36] and discharge proof obligations using automated theorem provers such as Z3 [22], CVC4 [6, 7], Yices2 [26], Alt-Ergo [19], Vampire [38, 46] or Simplify [23].

Work in verifying compilers often targets established languages (e. g., Java, C, C#, etc). And yet, such languages were not designed for verification and contain problematic features, including: *fixed-width number representations* [50, 60], *unrestricted aliasing* [47], *side-effects* [55], *closures* [20] and *flexible threading models* [15, 74]. This leads to the obvious idea that designing a language from scratch might be beneficial. *But what does the design space look like?* In this talk, I will attempt to shed light on this question.

2 Language Design

Certain language features have far reaching consequences. For example, the choice to employ static type checking reverberates throughout a language's design. There are many (percieved) benefits from static type checking, but there are also costs. Typically these costs are paid for through language restrictions (e.g. subtyping) and/or "escape hatches" (e.g. downcasts). Rust epitomizes this tradeoff where the pursuit of a single design goal (i.e. safe memory management without garbarge collection) leads to a language which has both: numerous strong restrictions (e.g. around uniqueness

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12.
https://doi.org/10.1145/3564719.3570917

and borrowing); and, sizeable escape hatches which compromise soundness (i.e. `unsafe` code). Similarly, traditional non-null checkers require unsafe casts to ensure common code patterns can be checked [2, 10, 27, 28, 39, 40, 58, 75].

The advent of verifying compilers presents both opportunities and challenges. On the one hand, the dichotomy between restrictions and escape hatches appears a relic of the past. With a verifying compiler, can we not have safe memory management without garbage collection **and** avoid debilitating language restrictions? Likewise, can we not scatter partially initialised (i.e. nullable) variables throughout our program **and** retain soundness? In short, can we not have our cake and eat it? The challenge is that the underlying technology, whilst a significant step forward, remains imperfect. Therefore, we must design languages to maximise this new technology whilst carefully navigating its limitations.

2.1 Assumptions

An important question is what can be assumed about this new technology. This boils down to what the underlying verification technology can reliably deliver. Can we assume it will quickly answer any (logical) question we ask of them? The answer, clearly, is no. This is an important difference from traditional compilers where, for example, type checkers are assumed to work reliably and efficiently. Instead, we must design our languages around the notion that these tools may or may not be able to answer the question. This means we need routes to success even when the theorem prover cannot immediately answer the question. For example, by introducing syntax for lemmas to manage complex proofs. Likewise, allowing assertions to help guide the verifier and/or assumptions to override the verifier when absolutely necessary.

Another aspect of this issue is how deeply we embed verification in the compilation pipeline. For example, suppose *definite assignment* is a required part of our compilation pipeline. Instead of employing a standard (conservative) dataflow analysis for this, we could employ the verifier here. Doing this might allow more complex code to pass definite assignment. And yet, at the same time, it would mean compilation itself could not succeed *unless* verification succeeded. In short, this may not (yet) be a sensible trade off. Instead, it might be better to stick with a traditional dataflow analysis, but allow for communication between the two. In Whiley, for example, this is achieved using a `fail` statement which is treated differently depending on context: for the traditional dataflow analysis, it is simply an *assumption* that code cannot reach beyond this point; for the verifier, this is an *assertion* which must be checked to ensure unreachability. This separation ensures traditional compilation can always succeed, and is never blocked by the verifier. Of course, to get the strong guarantees provided by verification this must (eventually) be applied. But, in building up to this, it is often useful to apply other lightweight approaches first, such as specification-based testing, bounded model checking, etc [9, 12, 34, 69, 70].

2.2 Design Choices

The design space for languages which support verification is surprisingly complex. Here we illustrate just a few examples to illustrate the range of different concerns.

Specification vs Implementation. Languages supporting verification often emphasise (with good reason) the distinction between what is *specification* and what is *implementation*. For example, Dafny has different array types for specification (e.g. `seq<T>`) versus implementation (e.g. `array<T>`). Likewise, syntax in *functions* differs from *methods* (e.g. `if x<0 then 0 else 1` vs `if(x<0) {return 0;}`). In contrast, Whiley has a single array type (e.g. `T[]`) which employs *mutable value semantics* [71, 72] — meaning it can be used interchangeably in both functions and methods [66].

Flow Typing. When specifying functions one must often distinguish cases in the postcondition, and flow typing proves useful here [62, 63]. Consider this example in Whiley:

```
function index(int[] a, int x)->(int|null r)
ensures (r is int) ==> (a[r] == x)
```

The key here is that, to type check the expression `a[r]`, the type checker must *retype* `r` afters a result of the type test `r is int` — exactly what flow typing was designed for! Furthermore, we observe that Dafny provides similar functionality over algebraic datatypes.

Memory Management. Finally, we illustrate a hypothetical feature made possible with verification. Safe memory management without garbage collection is, of course, a desiriable property for certain (e.g. system) languages. Rust adopts a single-minded approach based on borrowing and reference lifetimes. Rust's goal is to enforce key properties about aliasing, and employs a flow-sensitive type checking algorithm for this [43, 64]. Such an algorithm is more powerful than the traditional flow-insensitive type checker used in most existing languages. Nevertheless, it remains insufficient and, as a result, Rust places heavy restrictions on programmers (and often forces them into an `unsafe` place).

In a language supporting verification we can solve this problem more simply. Consider an *alias width* operator `#p` which returns the number of aliases for the object reachable by `p`. This can be used to reason about aliasing in specifications or for expressing key invariants (e.g. uniqueness). The statement `delete p` has precondition `#p == 1`; and, similarly, the assignment `p = new x` has postcondition `#p == 1`. With some care (esp. around when a variable is *moved* versus *copied*), we have a system for safe memory management without garbage collection. This offers significantly greater expressivity (e.g. than Rust), but makes the

verifier solely responsible for memory safety (i.e. rather than mixing borrow checking and `unsafe` blocks as in Rust).

3 Conclusion

Designing programming languages around verifying compilers is an interesting, but often overlooked, challenge. Nevertheless, the advent of mainstream verifying compilers represents a new era in programming languages, and a great opportunity exists to rethink their design for the better.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification — The KeY Book — From Theory to Practice*. LNCS, Vol. 10001. Springer.

[2] Nathaniel Ayewah and William Pugh. 2010. Null dereference analysis in practice. In *Proc. PASTE*. ACM Press, 65–72.

[3] J. Barnes. 1997. *High Integrity Ada: The SPARK Approach*. Addison Wesley Longman, Inc., Reading.

[4] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *JOT* 3, 6 (2004), 27–56.

[5] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. 2011. Specification and verification: the Spec# experience. *CACM* 54, 6 (2011), 81–91.

[6] C. Barrett and C. Tinelli. 2007. CVC3. In *Proc. CAV*. 298–302.

[7] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proc. CAV (LNCS, Vol. 6806)*. Springer-Verlag, 171–177.

[8] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Proc. BOOGIE*.

[9] J. Bowen and M. Hinchey. 2006. Ten Commandments of Formal Methods … Ten Years Later. *IEEE Computer* 39, 1 (2006).

[10] Patrice Chalin and Perry R. James. 2007. Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In *Proc. ECOOP*. 227–247.

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proc. FMCO*. 342–363.

[12] H. Chamarthi, P. Dillinger, M. Kaufmann, and P. Manolios. 2011. Integrating Testing and Interactive Theorem Proving. In *Proc. ACL2*. 4–19.

[13] Roderick Chapman and Florian Schanda. 2014. Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK. In *Proc. ITP*. 17–26.

[14] J. Chin and D. J. Pearce. 2021. Finding Bugs with Specification-Based Testing is Easy! *PROGRAMMING* 5 (2021), Article 13.

[15] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Proc. TPHOL*. 23–42.

[16] David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *Proc. NFM (LNCS, Vol. 6617)*. Springer-Verlag, 472–479.

[17] David R. Cok. 2014. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Proc. F-IDE*, Vol. 149. 79–92.

[18] D. R. Cok and J. Kiniry. 2005. ESC/Java2: Uniting ESC/Java and JML. In *Proc. CASSIS*. 108–128.

[19] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. 2018. Alt-Ergo 2.2. In *Workshop on Satisfiability Modulo Theories (SMT)*. HAL CCSD.

[20] Byron Cook, A. Podelski, and A. Rybalchenko. 2011. Proving Program Termination. *CACM* (2011), 88–98.

[21] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Proc. SEFM*. LNCS, Vol. 7504. Springer-Verlag, 233–247.

[22] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. TACAS*. 337–340.

[23] D. Detlefs, G. Nelson, and J. B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *JACM* 52, 3 (2005), 365–473.

[24] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. 1998. *Extended Static Checking*. SRC Research Report 159. Compaq Systems Research Center.

[25] L. Peter Deutsch. 1973. *An interactive program verifier*. Ph.D.

[26] Bruno Dutertre. 2014. Yices 2.2. In *Proc. CAV (LNCS, Vol. 8559)*. Springer-Verlag, 737–744.

[27] T. Ekman and G. Hedin. 2007. Pluggable Checking and Inferencing of Non-Null Types for Java. *JOT* 6, 9 (2007), 455–475.

[28] M. Fähndrich and K. R. M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*. ACM Press, 302–312.

[29] J. Filliâtre and A. Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proc. ESOP*. 125–128.

[30] Jean-Christophe Filliâtre. 2012. Verifying Two Lines of C with Why3: An Exercise in Program Verification. In *Proc. VSTTE (LNCS, Vol. 7152)*. Springer-Verlag, 83–97.

[31] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. 2002. Extended Static Checking for Java. In *Proc. PLDI*. 234–245.

[32] D. I. Good. 1985. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*. 55–75.

[33] Alwyn E. Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. 2013. Verification of Numerical Programs: From Real Numbers to Floating Point Numbers. In *Proc. NFM (LNCS, Vol. 7871)*. Springer-Verlag, 441–446.

[34] Alex Groce, Klaus Havelund, Gerard J. Holzmann, Rajeev Joshi, and Ru-Gang Xu. 2014. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *AMAI* 70, 4 (2014), 315–349.

[35] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 2013. Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions. In *Proc. ECOOP*. 451–476.

[36] C.A.R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12, 10 (1969), 576–583.

[37] C.A.R. Hoare. 2003. The verifying compiler: A grand challenge for computing research. *JACM* 50, 1 (2003), 63–69.

[38] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. 2011. Invariant Generation in Vampire. In *Proc. TACAS (LNCS, Vol. 6605)*. Springer-Verlag, 60–64.

[39] Laurent Hubert. 2008. A non-null annotation inferencer for Java bytecode. In *Proc. PASTE*. ACM Press, 36–42.

[40] Laurent Hubert, Thomas Jensen, and David Pichardie. 2008. Semantic Foundations and Inference of Non-null Annotations. In *Proceedings of the International conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Springer-Verlag, 132–149.

[41] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. NFM*. 41–55.

[42] B. Jacobs, J. Smans, and F. Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Proc. APLAS*. 304–311.

[43] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked Borrows: An Aliasing Model for Rust. In *Proc. POPL*. Article 41.

[44] S. King. 1969. *A Program Verifier*. Ph. D. Dissertation. Carnegie-Mellon University.

[45] Nikolai Kosmatov and Julien Signoles. 2013. A Lesson on Runtime Assertion Checking with Frama-C. In *Proc. RV (LNCS, Vol. 8174)*. Springer-Verlag, 386–399.
[46] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Proc. CAV (LNCS, Vol. 8044)*. Springer-Verlag, 1–35.
[47] Gregory Kulczycki, Heather Keown, Murali Sitaraman, and Bruce W. Weide. 2007. Abstracting Pointers for a Verifying Compiler. In *Proc. SEW*. IEEE, 204–213.
[48] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. 2005. How the Design of JML Accommodates Runtime Assertion Checking and Formal Verification. *SCP* 55, 1-3 (2005), 185–208.
[49] K.R.M. Leino and R Monahan. 2010. Dafny Meets The Verification Benchmarks Challenge. In *Proc. VSTTE*. 112–126.
[50] K. R. M. Leino. 2001. Extended Static Checking: A Ten-Year Perspective. In *Informatics — 10 Years Back, 10 Years Ahead (LNCS, Vol. 2000)*. 157–175.
[51] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. LPAR (LNCS, Vol. 6355)*. Springer-Verlag, 348–370.
[52] K. R. M. Leino. 2012. Developing Verified Programs with Dafny. In *Proc. VSTTE*. 82–82.
[53] K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97.
[54] K. Rustan M. Leino. 2018. Modeling Concurrency in Dafny. In *Proc. ETSS*. Springer-Verlag, 115–142.
[55] K. R. M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *Proc. ECOOP*. 491–516.
[56] K. Rustan M. Leino and Wolfram Schulte. 2007. A verifying compiler for a multi-threaded object-oriented language. In *Summer School Marktoberdorf 2006: Software System Reliability and Security*. IOS Press.
[57] D. Luckham, SM German, F. von Henke, R. Karp, P. Milne, D. Oppen, W. Polak, and W. Scherlis. 1979. *Stanford Pascal Verifier user manual*. Technical Report CS-TR-79-731. Stanford University, Department of Computer Science. 124 pages.
[58] C. Male, D.J. Pearce, A. Potanin, and C. Dymnikov. 2008. Java Bytecode Verification for @NonNull Types. In *Proc. CC*. 229–244.
[59] John W. McCormick and Peter C. Chapin. 2015. *Building High Integrity Applications with SPARK*. Cambridge University Press. https://doi.org/10.1017/CBO9781139629294
[60] David Monniaux. 2008. The pitfalls of verifying floating-point computations. *ACM TOPLAS* 30, 3 (2008).
[61] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proc. VMCAI*. 41–62.
[62] D. J. Pearce. 2013. A Calculus for Constraint-Based Flow Typing. In *Proc. FTfJP*. Article 7.
[63] D. J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Proc. VMCAI*. 335–354.
[64] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM TOPLAS* 43, 1 (2021), Article 3.
[65] D. J. Pearce and L. Groves. 2015. Designing a Verifying Compiler: Lessons Learned from Developing Whiley. *SCP* (2015), 191–220.
[66] D. J. Pearce and J. Noble. 2011. Implementing a Language with Flow-Sensitive and Structural Typing on the JVM. *ENTCS* 279, 1 (2011), 47–59.
[67] D. J. Pearce, M. Utting, and L. Groves. 2019. An Introduction to Software Verification with Whiley. In *Proc. ETSS*. Springer-Verlag, 1–37.
[68] D. J. Pearce, M. Utting, and L. Groves. 2022. Verifying Whiley Programs with Boogie. *Journal of Automated Reasoning* (2022).
[69] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. 2016. Your Proof Fails? Testing Helps to Find the Reason. In *Proc. TAP (LNCS, Vol. 9762)*. Springer-Verlag, 130–150.
[70] Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2014. How Test Generation Helps Software Specification and Deductive Verification in Frama-C. In *Proc. TAP*. 204–211.
[71] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2021. Native Implementation of Mutable Value Semantics. *CoRR* abs/2106.12678 (2021).
[72] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. 2022. Implementation Strategies for Mutable Value Semantics. *JOT* 21, 2 (2022).
[73] José Sánchez and Gary T. Leavens. 2014. Static verification of PtolemyRely programs using OpenJML. In *Proc. FOAL*. ACM Press, 13–18.
[74] Jan Smans, Bart Jacobs, and Frank Piessens. 2008. VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. 220–239.
[75] Fausto Spoto. 2008. Nullness Analysis in Boolean Form. In *Proc. SEFM*. IEEE, 21–30.
[76] Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019. Modular Verification of Heap Reachability Properties in Separation Logic. ACM Press, Article 121.
[77] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. 2015. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In *Proc. TACAS*. 566–580.
[78] Grigoriy Volkov, Mikhail U. Mandrykin, and Denis Efremov. 2018. Lemma Functions for Frama-C: C Programs as Proofs. *CoRR* abs/1811.05879 (2018).

Received 2022-08-12; accepted 2022-10-10