A Case for Combining Industrial Pragmatics with Formal Methods

Eric L. McCorkle

I. Background

A. Dependent Types

Dependently-typed languages [9] possess a type system with enough expressive power to describe formal specifications and prove that implementations behave according to them. Dependent type systems are necessarily undecidable as they perform formal reasoning about the behavior of an implementation; however, with this comes the power of being able to guarantee very strong safety properties of programs. Dependent types have seen use in a compiler [8], low-level programming [3], OS kernel components [1], web programming [4], and others. Looking further, the NSF DeepSpec [5] initiative actively seeks to develop methods for applying dependent types and verification to real-world applications. Several languages have arisen from current research efforts on dependent types, including Coq, Agda, Idris, and more recently, Lean. There are also a number of verification tools that have arisen from automated reasoning research, including ACL2, Dafny, and others. While it is possible to build software using these tools, they tend to be ill-suited for large-scale industrial software development.

B. Pragmatics of Industrial Programming Languages

There is a substantial body of knowledge and practice that deals with the factors that make a programming language usable in an industrial setting. Industrial-scale software development necessitates solving problems such as organization of large bodies of code, backward/forward compatibility, managing evolution over time, promoting good development practices, maintainability, and other such issues. Knowledge on this subject arguably exists in its purest form as institutional knowledge in organizations that maintain widely-used languages, such as OpenJDK (the organization responsible for the Java language), and tends to be published in industry conferences, keynotes and other talks or design documents for major languages rather than in traditional academic sources. Despite its existence primarily outside of academic sources, we believe this body of knowledge is valuable and beneficial. It encompasses both the demands that emerge at very large adoption and development scales as well as techniques proven effective at resolving such demands [7], deals with presenting complex ideas in a manner that is accessible to everyday programmers [6], and addresses the nontrivial task of implementing advanced ideas in the context of a working programming language.

II. A Vision of Pragmatic Dependently-Typed Languages

Our goal is to present the case that industrial pragmatics and formal methods can be integrated, that doing so would be benefical particularly from a security perspective, and to present a vision of how this might be accomplished.

A. Formal Methods in Iterative and Agile Settings

Agile, iterative development methodologies are a reality of industrial software development. Moreover, even in the most traditional, regimented processes, the pattern of software development employed by *individuals* tends to follow an iterative, exploratory path. These realities are at odds with the design of current dependently-typed languages and many formal verification tools, which tend to expect a program to be proven correct before it can be executed.

A key component of our vision is the notion that formal methods are not *fundamentally* at odds with agility and iterative development. To realize this vision, it is necessary for dependently-typed languages to adopt a "pay-as-you-go" model, presenting users the *option* of formally verifying functional components rather than the *requirement* of doing so. We advocate for the adoption of a gradual proof-checking design for dependently typed languages. In this design, proof-checking is separated from traditional type-checking, which generates proof obligations that can be proven to guarantee safety. This can be combined with gradual typing to provide a smooth transition from prototype to verified code. In this paradigm, dependent type systems amount to a reasoning system built into the language.

B. Managing Large Bodies of Verified Software

The challenge of how to manage large bodies of verified software- including the proofs -must be addressed if dependent types are to be usable in industrial settings. Proof automation is a major component of this vision. It has already been used to achieve an impressive level of automation and has been used to verify implementations of complex systems with almost no user input [2], [3]. Modern dependent-typed languages have some manner of proof-automation environment; we advocate a design where proofs can be written in the language itself and run in a managed environment. We also advocate the integration of object-oriented classes and typeclasses with dependent types and the design of new language features that allow the abstraction of proofs and theories. Finally, we advocate the application of API design to formal theories to create useful abstractions for verifying programs.

REFERENCES

[1] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pages 431–447, New York, NY, USA, 2016. ACM.

[2] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. *Journal of Functional Programming*, 18(5/6):599–647, 2008.

[3] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Notices*, 46(6):234–245, June 2011.

[4] Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. *SIGPLAN Notices*, 50(1):609–622, January 2015.

[5] National Science Foundation. Deepspec: The science of deep specification. http://www.deepspec.org/.

[6] Brian Goetz. State of the lambda: Libraries edition. Technical report, OpenJDK Project, September 2013. http://cr.openjdk.java.net/ briangoetz/lambda/lambda-libraries-final.html.

[7] Brian Goetz. Move deliberately and don't break anything. Keynote Address, 2015. http://gotocon.com/cph-2015/presentation/Keynote:

[8] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[9] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.