

Viewpoint

Teach Foundational Language Principles

Industry is ready and waiting for more graduates educated in the principles of programming languages.

THE NEED FOR more people to learn to program has received widespread attention recently (see, for example, www.code.org and its recent “Hour of Code” held during CS Education week in December of 2013 and 2014). While the ability to program has tremendous potential to support and channel the creative power of people, we should remember that programming languages continuously arise as the need to solve new problems emerges and that it is *language principles* that are lasting. As we discuss in this Viewpoint, language foundations serve an increasingly important and necessary role in the design and implementation of complex software systems in use by industry. Industry needs more people educated in language principles to help it deliver reliable and efficient software solutions to its customers.

Historically, many important principles of languages have arisen in response to the difficulties of designing and implementing complex systems. Garbage collection, introduced by John McCarthy around 1959 for the Lisp language, is now commonplace in modern programming languages such as Java and C#, as well as popular scripting languages such as Python and JavaScript.⁸ Dijkstra’s “Go To Statement Considered Harmful” *Communications* Letter to the Editor advocated the use of structured programming, which is enshrined in all modern programming languages.⁵ Type systems classify program expressions by



the kind of values they compute,¹² enabling compilers to prove the absence of certain kinds of errors and optimize code more effectively. Hoare’s assertion-al method provides a framework for establishing the correctness of programs.⁶

As the complexity of the systems we desire to build increases, new mechanisms to express programmer intent at a higher level are required in order to deliver reliable systems in a predictable manner. The design of new programming languages is driven by new classes of systems and the desire to make programming such systems within the reach of more people. New methods for expressing programmer intent take many forms including features of general-purpose languages, domain-specific languages, and formal specification languages used to verify properties of high-level designs.

Experiences with bugs like the recent TLS heartbeat buffer read overrun in OpenSSL (Heartbleed)¹⁰ show the cost

to companies and society of building fundamental infrastructure in dated programming languages with weak type systems (the C language in this case) that do not protect their abstractions. Several companies have developed new safe systems programming languages to address the challenge of programming scalable and reliable systems. Examples include the Go language from Google (<http://golang.org/>), the Rust language from Mozilla (<http://www.rust-lang.org/>), and the Sing# language from Microsoft (<http://singularity.codeplex.com/>). Such languages raise the level of programming via new type systems that provide more guarantees about the safety of program execution.

Domain-specific languages (DSLs) go further by restricting expressive power to achieve higher-level guarantees about behavior than would be possible with general-purpose languages. The SQL database query language is a classic example, based on the relational algebra,² which enables sophisticated query optimization.

DSLs continue to find application in industry. Google’s Map/Reduce data parallel execution model³ gave rise to a number of SQL-inspired DSLs, including Pig (<http://pig.apache.org/>) from Yahoo. The Spiral system from ETH Zurich (<http://www.spiral.net/>) generates very efficient platform-specific code for digital signal processing from declarative specifications of mathematical functions and optimization rules. Intel makes Spiral-generated code available

as part of its Integrated Performance Primitives library. Colleagues at Microsoft recently developed a DSL called P for programming asynchronous event-driven systems⁴ that allows a design to be checked for responsiveness—the ability to handle every event in a timely manner—using model checking.¹ Core components of the Windows 8 USB 3.0 device driver stack were implemented and verified using P.

Another important class of languages are specification languages, which allow the designers of systems and algorithms to gain more confidence in their design before encoding them in programs where it is more difficult to find and fix design mistakes. Recently, Pamela Zave of AT&T Labs showed the protocol underlying the Chord distributed hash table is flawed¹⁴; she modeled the protocol in the Alloy language⁷ and used the Alloy Analyzer tool to show that “under the same assumptions about failure behavior as made in the Chord papers, no published version of Chord is correct.” Emina Torlak and colleagues used a similar modeling approach to analyze various specifications of the Java Memory Model (JMM) against their published test cases,¹³ revealing numerous inconsistencies among the specifications and the results of the test cases.

Our recommendations are three-fold, visiting the three topics discussed in this Viewpoint in reverse order (formal design languages, domain-specific languages, and new general-purpose programming languages). First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, its application in design formalisms, and experience the creation and debugging of formal specifications with automated tools such as Alloy or TLA+. As Leslie Lamport says, “To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper.”⁹ The methods, tools, and materials for educating students about “formal specs” are ready for prime time. Mechanisms such as “design by contract,” now available in mainstream programming languages, should be taught as part of introductory programming, as is done in the introductory programming language


Many important principles of languages have arisen in response to the difficulties of designing and implementing complex systems.

sequence at Carnegie Mellon University.¹¹ Students who learn the benefits of principled thinking and see the value of the related tools will retain these lessons throughout their careers. We are failing our computer science majors if we do not teach them about the value of formal specifications.

Second, would-be programmers (CS majors or non-majors) should be exposed as early as possible to functional programming languages to gain experience in the declarative programming paradigm. The value of functional/declarative language abstractions is clear: they allow programmers to do more with less and enable compilation to more efficient code across a wide range of runtime targets. We have seen such abstractions gain prominence in DSLs, as well as in imperative languages such as C#, Java, and Scala, not to mention modern functional languages such as F# and Haskell.

Third, anyone who has a desire to design a new programming language should study type systems in detail; B.C. Pierce’s *Types and Programming Languages*¹² is a very good starting point. The fact that companies such as Microsoft, Google, and Mozilla are investing heavily in systems programming languages with stronger type systems is not accidental—it is the result of decades of experience building and deploying complex systems written in languages with weak type systems. To move our industry forward, we very much need language designers who come to the table educated in the formal foundations of safe programming languages—type systems.

Conclusion

Future applications and systems will increasingly rely on principled and formal language-based approaches to software development to increase programmers’ productivity as well as the performance and reliability of the systems themselves. Software developers will need a solid understanding of language principles to be effective in this new world and increased emphasis on these principles in computer science education is required. For readers interested in the topic of programming languages in education, we strongly urge consulting the work of the SIGPLAN Education Board (<http://wp.acm.org/sigplaneducationboard/>), which rewrote from scratch the “Knowledge Area” of “Programming Languages,” contained in the first public draft of the 2013 Computer Science Curricula Report (<http://cs2013.org>). 

References

1. Clarke, E.M., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 2001, I–XIV, 1–314.
2. Codd, E.F. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (June 1970), 377–387.
3. Dean, J. and Ghemawat, S. MapReduce: A flexible data processing tool. *Commun. ACM* 53, 1 (Jan. 2010), 72–77.
4. Desai, A. et al. P: Safe asynchronous event-driven programming. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.
5. Dijkstra, E.W. Letters to the editor: Go To statement considered harmful. *Commun. ACM* 11, 13 (Mar. 1968), 147–148.
6. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM* 12, vol. 10 (Oct. 1969), 576–580.
7. Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, 2012.
8. Jones, R. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall, 2012.
9. Lamport, L. Why we should build software like we build houses. *Wired* 25 (Jan. 2013).
10. OpenSSL Project. OpenSSL Security Advisory [07 Apr 2014]. (Apr. 7, 2014); http://www.openssl.org/news/secadv_20140407.txt.
11. Pfenning, F. Specification and verification in introductory computer science. Carnegie Mellon University; <http://c0.typesafety.net/>.
12. Pierce, B.C. *Types and Programming Languages*. MIT Press, 2002, I–XXI, 1–623.
13. Torlak, E., Vaziri, M. and Dolby, J. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.
14. Zave, P. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.* 42 (Mar. 2012), 49–57.

Thomas Ball (tball@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RISE) group at Microsoft Research, Redmond, WA.

Benjamin Zorn (zorn@microsoft.com) is a principal researcher and co-manager of the Research in Software Engineering (RISE) group at Microsoft Research, Redmond, WA.

Copyright held by authors.