

Types for Systems Design: A Position Paper

Robert Harper
Carnegie Mellon University
<http://www.cs.cmu.edu/~rwh>

October 1, 2003

There is a enormous gulf between the design principles governing a software system and its implementation. Designs are expressed in many forms, from blackboard drawings, informal analogies, box-and-pointer diagrams, state transition systems, and, perhaps most importantly, an unstated understanding among a team of developers. The role of the design description is normative: it determines not only what *should* be done, but also what *should not* be done.

Even if an initial implementation coheres well with its design, over time it inevitably diverges from it. The code becomes its own design specification, and the design loses its normative role and becomes a mere description of the code. The result is a morass for both the developer and the end user. For the developers it becomes increasingly difficult to understand what is going on with the program, and increasingly risky to modify and extend it. For the users the program becomes increasingly idiosyncratic, because there is no clear model of how it is “supposed” to behave.

To avoid this it is necessary that *the design live with the code* in a form that *must* evolve with the code and which permits *enforcing* compliance of the code base with the design. In principle a design description should include everything from the end-user conceptual model to the specification of the behavior of individual program components and their interaction with one another. But this is not just a matter of documentation strings! The design description *must have computational content* in the sense that it determines verifiable properties of the execution behavior of the executable code. Most importantly, the verification *must be performed whenever the code is modified*. Only then can we be certain that designs make computational sense and that computations conform to a sensible design.

Research on *type systems* addresses the problem of how to integrate system descriptions with system implementations in such a way that coher-

ence can be assured. The simplest type systems — such as Java’s — provide minimal descriptive power in exchange for concise specification and simple compliance checking. Richer type systems — such as ML’s or Haskell’s — go further by exploiting parametricity to permit enforcement of abstraction properties and enforcing modular structure. Type systems are the most successful, and widely used, “formal method” yet devised.

These successes instill optimism that much more can be done. Extending the expressive power of type systems, while retaining their close correspondence with running code, is an active area of current research. Here is a small sample:

1. Enforcement of large-scale architectural constraints on systems [1].
2. Ensuring compliance with locking disciplines, or, more generally, interaction protocols for API’s [5, 7].
3. Ensuring secure information flow in a program [8].
4. Managing “ownership” of data structures among software components [3].
5. Bounding resource usage by a program [9].
6. Expressing and enforcing data structure invariants [6, 10].
7. Generating certified object with checkable safety and security properties [4].

The trend is clear. As the reliability and usability of software systems gain importance, so also do methods for ensuring that these conditions are met and maintained throughout the life cycle of a system.

Robert Harper is a Professor of Computer Science at Carnegie Mellon University. He is best known for his contributions to the design and implementation of Standard ML, the design of the LF Logical Framework, and the development of type-theoretic methods for specifying and implementing programming languages.

References

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th international conference on Software engineering*, pages 187–197. ACM Press, 2002.

- [2] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002.
- [3] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. *ACM SIGPLAN Notices*, 38(1):213–223, 2003.
- [4] Karl Crary, Robert Harper, Peter Lee, and Frank Pfenning. Automated techniques for provably safe mobile code. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX '00)*, pages 406–419, Hilton Head Island, SC, January 2000. IEEE Computer Society Press.
- [5] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, May 2001.
- [6] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277. ACM Press, 1991.
- [7] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proc. International Conference on Functional Programming*, Uppsala, Sweden, September 2003.
- [8] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.
- [9] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer, 1998.
- [10] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of the 26th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 214–227, January 1999.