Adoption-Oriented Language Design

LEO A. MEYEROVICH ◆ LMEYEROV@EECS.BERKELEY.EDU ◆ DARPA ISAT ON ADOPTION, 2013

Language designers optimize features for adoption, yet research provides little guidance. Worse, optimizing for adoption leads to foregoing the suggestions that research does provide. For example, advanced type systems are breaking ground in guaranteeing correctness, yet after 30 years, we rarely see even ML-style type inference. As Erik Meijer relates, if the mountain will not come to Mohammed, then Mohammed must go to the mountain [1]. To do so, we ask: how can researchers explicitly optimize for adoptiblity to improve the value proposition of our work?

We propose three constructive steps towards *adoption-oriented language design*. Designing for adoption takes many forms because adoption itself is multifaceted: it is a desirable result, an exploitable process, and applies to both languages and features. Driven by our survey of social science literature on adoption [2] and our analysis of adoption factors for over 200,000 projects and 15,000 developers [3,4,5], we identify three constructive areas of adoption-oriented design:

I. **Feature Streamlining:** Improving adoptability of a feature
II. **Language Targeting:** Improving the adoptability of a language
III. **Network Effect Constructs:** Designing features that improve with adoption

I. Feature Streamlining: Improving Adoptability of a Feature

We need structured approaches to improving feature adoptability. As one idea, we look to the *diffusion of innovation* (DoI) model of adoption that Rogers [6] synthesized from thousands of case studies. Notably, he found common catalysts and obstacles for adoption. We repurpose these factors as an adoptability rubric and, by applying it, motivate targeted improvements to features.

Rich types, such as dependent types and type qualifiers, provide a playground for applying our DoI-inspired rubric. Researchers already recognize an adoption problem for types and proposed gradual typing and blame as solutions. Unfortunately, these barely address DoI factors:

✓ **Compatibility***:* Gradual typing streamlines adding new code with strong static guarantees to a legacy codebase with weaker guarantees, and blame improves error reporting for such integrations. Both improve the *compatibility* of richly typed code with weaker legacy code.

Code fragments are still entirely richly typed or not with these solutions, and so programmers must still make hard shifts. A la carte levels of type checking would improve compatibility. (Full streamlining has costs, however, such as type-directed programming.)

X **Simplicity***:* Blame guarantees that strongly typed code is not the source of errors when combined with more weakly typed code. However, it says little about error handling. For understanding of a feature (*simplicity)*, programmers may desire predictability. Imagine using a runtime monitor for checking gradual types (i.e., via a higher-order contract): it should only throw errors at a few expected locations and times, not at random instructions.

X **Relative Advantage and Observability**: The benefit of an innovation should be made apparent to programmers. One benefit of rich types is in detecting bugs, so a compiler can show a benefit of types by keeping a bug log: “These 507 errors would not have been caught by a weaker type system”, “These 203 fixes occurred when type annotations were added”, etc.

The DoI factors of *compatibility, simplicity,* and *observability* of *relative advantage* motivate improvements to type systems. We might similarly optimize other features and even languages!

II. Language Targeting: Improving Adoptability of a Language

Languages are adopted through various social processes. We propose tools for targeting them:

- **Targeting Niches with Analytics***:* Proebsting [7] posited that adopted languages fill niches. Popular instances are PHP for the web, Max/MSP for audio, and Matlab for statistics.

Consequently, designers spend time isolating and specializing for a niche. For example, a group designing a DSL for hardware construction adapted our adoption surveys to understand their users, and we are tracking compiler use for our own DSL. Analytics tools such as these surveys and usage trackers should be better understood and supported.

- **Reinvention through Deployment:** Adoption can drive improvement, as is seen for the reinvention of laws [8] as they enter new domains. A reinvention may be an improvement because it fixes past mistakes (*social learning*) or generalizes for new scenarios (*adaptation*).

  Tool support may help accelerate the reinvention process. For example, the language-as-a-library notion enables rapid implementation of features, so a big remaining bottleneck for achieving rapid design is in how to introduce domain users into the design loop. User testing consumes significant time resources, so community management tools might lower the cost of expert trials with a demographically targeted equivalent of Mechanical Turk.

We should further explore such tools and processes for targeting niches and refining features.

III. Network Effect Constructs: Improving with Adoption

Our most radical call for adoption-oriented language design is for features that strengthen with adoption. This is starting already: developers have asked 4.3 million questions on Stackoverflow (January, 2013), given 8.2 million answers, and all with an overall 80% answer rate. The more a language is adopted, the more Stackoverflow activity there is on it [9], and the more likely a question can be answered by searching over past responses. The *network effect* refers to Stackoverflow's value being dependent upon its adoption. Given the societal scale of programming and program use, the network effect provides an enormous resource for languages!

Most aspects of languages should exploit the network effect. We already see cases beyond debugging, such as in open source libraries, corpus-based code completion, and cooperative bug isolation. Network effects might also address big language problems: cooperative verification, cooperatively secure runtimes, and cooperative profile-driven optimization. What can't they help?

Conclusion

Adoption is a top concern for language designers, so the research community should better engage with it. We showed several constructive ways to do so, such as redesigning individual features like advanced type systems to be more adoptable, and designing features and processes for improving overall language adoption such as through analytics platforms. Most radical of all, we recognize that one of the best resources for a programmer is the network effect, so we should more aggressively focus on features that strengthen with adoption. There is much to do!

References

[1] E. Meijer. 2007. Confessions of a Used Programming Language Salesman. In *OOPSLA '07*.
[2] L. A. Meyerovich and A. S. Rabkin. 2012. Socio-PLT: Principles for Programming Language Adoption. *Onward! '12*.
[3] L. A. Meyerovich and A. S. Rabkin. 2012. Social Influences on Language Adoption. (Google Tech Talk video: http://www.youtube.com/watch?v=v2ITaI4y7_0).
[4] L. A. Meyerovich and A. S. Rabkin. 2012. How Not to Survey Developers and Repositories: Experiences Analyzing Language Adoption. In *PLATEAU '12*.
[5] L. A. Meyerovich and A. S. Rabkin. 2012. Interactive Visualizations of Language Adoption. http://www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/viz/index.html
[6] Rogers, E. M. *Diffusion of innovations*. Simon and Schuster, 1995.
[7] T. A. Proebsting. Disruptive programming language technologies. Talk at MIT. (Slides available: http://ll2.ai.mit.edu/talks/proebsting.ppt), November 2002.
[8] H. R. Glick and S. P. Hays. Innovation and Reinvention in State Policymaking: Theory and the Evolution of Living Will Laws. The Journal of Politics, 1991.
[9] S. O'Grady. 2012. The RedMonk Programming Language Rankings. http://redmonk.com/sogrady/2012/09/12/language-rankings-9-12/