Socio-PLT: Principles for Programming Language Adoption

Leo A. Meyerovich
UC Berkeley
lmeyerov@eecs.berkeley.edu

Ariel Rabkin
UC Berkeley
asrabkin@eecs.berkeley.edu

Abstract

Why do some programming languages fail and others succeed? What does the answer tell us about programming language design, implementation, and principles? To help answer these and other questions, we argue for examining the sociological groundings of programming language theory: *socio-PLT*.

Researchers in the social sciences have studied adoption in many contexts. We show how their findings are applicable to programming language design. For example, many programming language features provide benefits that programmers cannot directly or immediately observe and therefore may not find compelling. From clean water to safe sex, the health community has repeatedly identified and surmounted similar observability barriers. We use such results from outside of programming language theory to frame a research agenda that should help us understand the social foundations of languages. Finally, we examine implications of our approach, such as for the design space of language features and the assessment of scientific research into programming languages.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: general

General Terms Languages, Human Factors

Keywords history, sociology, programming language adoption

1. Introduction

One goal of programming language research is to improve software. Achieving this usually means having research ideas, in some form, eventually adopted by programmers. Adoption, however, is a serious challenge. This paper examines how we can leverage scientific insight and methodologies from studies of group behavior (particularly in sociology) to address a fundamental gap in programming language research and practice.

In this paper, we use "adoption" to refer to two related processes. Feature adoption is the process of language designers trying, using, modifying, and spreading a programming language feature. Language adoption is the parallel process by which programmers take up a programming language. Both forms of adoption are possible avenues for improvement.

Today, conducting research that is successfully adopted is a matter of art, not of reliable procedure — and perhaps more happenstance than art. Language designers sometimes share their thoughts about the design process they went through and about the process by which their languages have become adopted as relevant contributions [34, 36, 53]. However, scientific principles are lacking. Adoption should not be analyzed as just an eventual marketing barrier but also, for example, the process of exploiting social learning to improve an innovation (Section 3.4). This paper outlines an agenda for putting more science into the design process.

Our work focuses on the sociological foundations of language and feature adoption. In contrast, we do not examine intrinsic properties such as the precise mathematical benefits of a new lambda calculus. We are more concerned in whether outside interests led to the examination of the lambda calculus derivative and how that derivative might then evolve to be relevant outside of the academic community [53]. Likewise, we do not examine whether *individuals* find the syntax of one language more cognitively or psychologically consonant than another [62] but ask if the spread of a semantic feature is being impeded by an orthogonal one such as syntax [34]. Rather than understanding the psychological processes of individual programmers, we want to understand the social aspects of programming languages.

This paper makes three main contributions.

1. **We explain why understanding and exploiting adoption ought to be more of a scientific priority for the programming language community.** As part of this, we survey hard-won lessons, unsubstantiated beliefs, and recurring questions in the language community (Section 2).

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.

2. **We show that scientifically grounded social theories are available for understanding and inducing adoption** (Section 3) yet are not known to the programming language community (Section 2). To do so, we draw upon a wide body of social sciences and, to help future investigation, identify them. For example, safe sex advocacy research might teach the Haskell community how to better persuade programmers to use an advanced type system.
3. **We present a research agenda for investigating and exploiting language adoption** (Section 4). The agenda includes a specific set of questions and hypotheses (Sections 2 – 4) and new language features whose value directly derives from adoption (Section 4.3).

Our paper is loosely structured around the three points above, with a section centered on each one.

Beyond the present work, we are performing qualitative and quantitative analysis of programming language adoption as well as using these notions in the design of our own systems – we are practicing what we are preaching. The focus here, however, is to present the underlying social theories and subsequent style of analytic reasoning that led to our research program.

Focusing on social adoption issues is timely for several reasons. First, the increased prominence of domain-specific languages and software crises in security, parallelism, and mobile have reinvigorated language demands throughout computing. Furthermore, sociology and allied social sciences have undergone a quantitative revolution that has yielded transferable principles and methodologies. Finally, languages and programs are increasingly developed by open community-based processes, so the activities of designers and programmers are increasingly accessible to researchers. The need is old, but we now have the tools and data to address it.

The purpose of this paper is to bring together research not previously known to the programming language community and to analyze its implications. We raise more questions than we answer. These questions are marked out in numbered boxes, divided into open-ended questions and falsifiable hypotheses. Many of them can be examined with basic empirical techniques that are typical of the social sciences. Answering other questions may require new research techniques.

2. Hard-won experiences

This section describes some of the past successes and failures of the programming language research community in getting its techniques adopted. Along the way, we will point out some of the hard-won insights of a half-century of experience. These insights, however, are only part of the story. A major lesson we derive from the review is that the community does not have a complete nor adequate understanding of the adoption process.

2.1 Grand Ambitions to Incremental Advances

We begin at the high level, discussing the role of researchers in language design. Sometimes, researchers have committed themselves to ambitious standardization efforts and industrial collaborations. At other times, researchers have offered radical new language designs. The underlying philosophy in both cases is that research can reshape practice. We see that it occasionally does – but only rarely by building languages for direct adoption.

Design by committee. Committees of experts (including language researchers) have repeatedly tried to design general-purpose programming languages. These efforts have routinely resulted in high-profile and costly failures. PL/1 and Algol-68 are roughly contemporaneous examples of this tendency. Ada is a more recent one. In each case, the result was a complex design that was hard to implement and widely disliked [34].

The language designers made several methodological missteps. Perhaps the most serious was to attempt to placate all users, based on an incomplete understanding of their needs (Section 3.4), rather than having a substantive dialogue with users to understand their priorities. Another was to not have precise end goals (Section 3.3).

Committees sometimes produce good designs – indeed, the designs of Algol-60 and Haskell are routinely lauded. The correct lesson to draw is that design committees have risks associated with them. These risks are widely known in other fields and programming language designers rediscover them at their own cost.

Whole-cloth radical design. Language researchers often produce radically new designs for general-purpose languages. For instance, in 1978, John Backus called for a point-free functional style [7]. Point-free programming, today, is still not well known outside of the academic community. Likewise, the Prolog vision of declarative programming has still not become a dominant paradigm. Even when a radical design is adopted, such as dataflow languages for composing electronic music [51], it may be significantly evolved and specialized. Whole-cloth designs are opportunties for substantial improvements, but the convoluted paths of ultimately successful approaches suggest that extrinsic factors are influential.

Engineering language designs. Others have offered a contrasting vision of the language design process. In 1973, C. A. R. Hoare advised that a language designer should "have excellent judgment in choosing the best" features, but should not "include untried ideas of his own. His task is consolidation, not innovation [35]." The role of designing practical languages and of experimenting with language features ought to be separate, presumably with the programming language research community focusing on the latter.

Hoare's advice confronts researchers with a dilemma. How are we to conduct a representative trial of a new language feature without incorporating it into a reasonably us-

able tool and having normal users attempt to use it? As argued by Markstrum [50], having the inventor of a feature try it out and then argue persuasively for it is a poor substitute for empirical evidence. In 1973, conducting a qualitative and quantitative analysis of how a language feature is used would have been very challenging. Today, however, we can access large repositories of source code and easily communicate with their developers. This is just one way to substitute systematic study — scientific research — for "excellent judgment."

In practice, the original designers of today's popular languages are typically not experienced in programming language design. Rather, Proebsting [65] noted that a common pattern is for programmers with expertise in other domains to create a language when they perceive an unmet need. Dennis Ritchie (C), Guido van Rossum (Python), Larry Wall (Perl), James Gosling (Java), and Rasmus Lerdorf (PHP) are all examples of this pattern. If the pattern continues to hold, the programming language community should go beyond directly justifying the importance of particular features to programmers [37]. We should better consult with the wider software development community to see what is relevant [53, 82] and much more effectively transfer our results to new language designers, who will usually come from outside of our community (Section 3.3).

2.2 The Road to Functional Programming

The evolution of functional languages provides a case study into socio-technical aspects of language adoption. We find that leaders of the functional programming movement are and have been actively enmeshed in adoption issues. They have consistently made adoption an explicit goal and guided their efforts by their (occasionally flawed) understanding of it.

2.2.1 Continuations

Continuations illustrate a 50 year long feedback loop between theory and practice. Algol 60's construct for jumping out of blocks begins Reynolds's [67] history of continuations. Implementing `goto` was simple, akin to implementing continuations, but formally reasoning about it challenged academia. Early on, van Wijngaarden [79] showed how to reduce the construct away by what is now known as the continuation-passing style (CPS) transformation. At least one direct practical result is famous: Dijkstra's warning "Go to Statement Considered Harmful." [19, 67] Contemporaneously, Landin [43] extended his SECD machine to support Algol by adding the general but non-functional J operator.

Subsequent generations of researchers and practitioners grew the J operator into a variety of continuation operators, complete with functional reasoning [22, 81] and optimized implementations. The feedback loop continues today. Notably, the orchestration and performance needs of server-based web applications are posing research challenges for design and implementation [42, 66, 80].

We must wonder, however:

Question 1. *Why are industrial language designers adopting generators and coroutines instead of continuations?*

Generators and coroutines have weaker expressive power: they are equivalent to one-shot continuations [39, 57]. Nor are these weaker forms new: it has been many decades since they were introduced [16, 46, 83]. This timeline is troubling and evokes questions about the essential design and adoption trajectory of control constructs.

2.2.2 Higher-order functions

Higher-order functions also prompt alarming questions about feature adoption. Consider Backus's advocacy of point-free functions. He argued that most languages were too complicated for anything beyond incremental evolution but using point-free functions would be simple enough to enable extension [7]. However, point-free functions are not prevalent today, language implementations are complicated, and, as measured by Chen et al. [13], simplicity is a poor indicator for adoption.

Question 2. *Was Backus right in that language complexity is a sufficiently persuasive reason to adopt functional programming? More strongly, is worse better [23] in that implementation simplicity trumps language generality?*

Hughes and Wadler disagreed. Ten years after Backus, Hughes instead argued that non-functional programmers must be shown the benefits of functional programming if they are to use it [37]. In particular, he believed developers prize modularity and that they could be swayed by demonstrating how higher-order functions achieve it for examples like numerical differentiation and integration. A decade later, Philip Wadler suggested that designers were targeting the wrong properties: interoperability and footprint are more important than is widely realized and performance is less so [82]. Departing from Hughes's simple examples, Wadler advocated developing sizable "killer applications" that are facilitated by functional programming and proactively collaborating with corporations.

Question 3. *What actually convinces programmers to adopt a language?*

Today, functional programming may be a victim of the persuasiveness of higher-order functions. Instead of adopting higher-order functions explicitly, many popular languages use objects: closures and objects are similar [17, 78] in many of the ways that motivated Hughes [37]. Higher-order functions are a successful *reinvented* adoption (Section 3.4).

The success of objects is troubling for functional programming research. New functional languages often feature structural typing, type classes, and verified implementations, while object-oriented languages more typically innovate in nominal typing, inheritance, and meta-object protocols [10].

Technical differences impede going from one to the other. The challenge in applying research ideas across these different domains is exacerbated by the social gap between the communities: tacit knowledge is hard to transfer [60] and what is important for one community need not be for another (Section 3.3).

Hypothesis 1. *Working on the closure side of the duality increases influence on programming language researchers, but decreases influence on both practitioners and software engineering researchers.*

2.2.3 Lazy evaluation, ML, and sexy types

Lazy evaluation, ML, and sexy types are more recent and controversial. Lazy languages abstract away from the von Neumann computer model of a program as a sequence of instructions, which is a deep change for programmers. Introduced contemporaneously [55, 77] with laziness, Damas-Hindley-Milner "ML-like" type systems assist programming over static semantics. Haskell was created to collectively explore such ideas [64] – the Great Experiment of functional languages. Researchers today are continuing the experiment with "sexy" [64] types (phantom, existential, etc.) that enrich Damas-Hindley-Milner.

Hughes argued that, next to higher-order functions, laziness is the other prominent feature of functional programming that compels adoption [37]. Despite such early enthusiasm, Peyton Jones's Haskell retrospective [64] fifteen years later concluded that laziness caused "prolonged embarrassment" and dropped it from the list of what really matters. Instead of laziness, Peyton Jones valued the newer ideas of monads, type classes, and sexy types.

Question 4. *How can we prevent sexy types from being another source of "prolonged embarrassment" to the research community?*

In this section, we saw that the limited adoption of functional programming techniques was not due to lack of interest from researchers. Rather, understanding adoption is so challenging that it justifies its own field of scientific study.

2.3 Calls for Social Awareness in Language Design

Our paper builds upon the growing awareness and investigation of human factors for programming languages and software engineering. Here, we survey representative work in the field.

Language retrospectives. SIGPLAN already sponsors a series of conferences on the history of programming languages (HOPL). The bulk of the papers are retrospectives by language designers and they provide valuable source material for our inquiries.

Unfortunately, these retrospectives are clouded by a focus on intrinsic feature design. It would be valuable to collect information about decision making: how did user populations change and react over time and how did that relate to the language design? Today's language designers are not necessarily trained nor ideally placed to evaluate how different factors influence adoption. Gathering data about language trends [13] is an important first step.

Question 5. *What data should language designers track?*

User-centric language design. Language designers are increasingly examining extrinsic factors in a rigorous way.

These efforts often focus on individual users. For example, the Psychology of Programming Interest Group (PPIG) community established a precedent for examining individual psychological [61] and cognitive [62] factors. In addition to such efforts, we argue for socio-technical research of languages that studies group phenomena.

Scientific norms for rigorous socio-technical research are in their infancy (Section 4.4). Consider Wadler's call for application-driven research in the functional programming community. Research that supports concrete applications that the community is struggling with is aligned for impact (Section 3.3) and can exploit valuable perspectives and knowledge that are otherwise unavailable (Section 3.4). However, empirical attempts are easily marred by inappropriate methodology [30, 50]. For example, it is difficult to draw conclusions from studies on students [69] unless, of course, novices are the topic of interest [61].

Question 6. *How do we perform, evaluate, and benefit from research into developing applications in a language?*

Adoption-centric language design. Language and feature designers sometimes explicitly focus on adoption.

Erik Meijer's "Confessions of a Used Programming Language Salesman" [53] describes his programming language research in terms of adoption. For example, believing interoperability with pointer-based languages to be essential to the adoption of Haskell, he co-invented phantom types, a solution that was contentious at the time. Meijer argues that interesting and important cases for language features appear when we stress them. For the same reason, he casts the development of limited calculi as an inherently incomplete approach to the theoretical design of a feature (Section 3.4).

A second takeaway from Meijer is that the benefits of a feature – such as higher-order functions improving modularity – is only a piece of the adoption process. *Switching costs* will dissuade adoption, such as learning a new technology or interfacing with legacy code, and so they should also be considered. He attributes this insight to the notional *change function* [15]:

$$\text{Change Function} = F(\frac{\text{Perceived Crisis}}{\text{Perceived Pain of Adoption}})$$

The change function models the adoption probability of an innovation as the perceived benefit tempered by the perceived cost. To Meijer, language researchers are failing to

gain traction because their solutions are too painful: language features do not optimize the denominator.

Even such a simple model is subtle because the costs and benefits are subjective. Under it, a valid alternative hypothesis for adoption failure is that the costs and benefits of language research are not correctly perceived. For example, programmers may perceive the benefits of sexy types to be much lower than language designers do. Likewise, the perceived cost may be higher than the actual objective cost. Consider unit testing enthusiasts who see little value in additionally following a static typing discipline, and the type theorists who disagree:

Question 7. *What are the perceptions about testing and typing that drive the differing beliefs, how do they relate to practice, and what opportunities and course corrections does this pose?*

Hypothesis 2. *Both programming language designers and programmers incorrectly perceive the performance of languages and features in practice.*

The change function is insightful but simplistic; it does little to help understand many basic phenomena about competing technologies. Sociologists have therefore empirically developed more nuanced models such as Rogers' diffusion of innovation and Mark's ecological theory (Section 3.2).

Question 8. *What are appropriate models for adoption?*

Empirical software engineering. Empirical software engineering, typically performed by mining software repositories and developer communications, is an emerging area of research. One particularly relevant result is that modularity bugs are often better understood in terms of human communication boundaries rather than strict code boundaries [9]. We hope to reproduce such successes of the empirical software engineering community, and with more emphasis on both programming languages and social principles.

A recent promising course of empirical analysis has been for addressing the burgeoning problems of concurrency and parallelism. Lu et al. performed an exemplary study of concurrency bugs in practice [47] that shows language and analysis research ignores wide-spread technical problems. Constructively, they also show that the statistical properties of actual bugs help motivate and guide context-bounded solutions [58]. Such analyses [6, 27, 32] are quite rare compared to the large volume of research devoted to concurrency and parallelism. Even in these studies, few socio-technical details are examined, such as the relevance of education or team structure.

We stress the importance of demographics. For example, when studying generics in large Java software, Parnin et al. [63] found that only one or two individuals in each project are responsible for most of the uses. Who are these individuals? Does their education and work history matter? Or perhaps is their seniority in the project or experience with the code base more significant? Similar questions are pertinent for investigating the additional finding that most developers use generics in a narrow way or not at all.

Hypothesis 3. *Developer demographics influence technical analysis.*

3. Mobilizing Transdisciplinary Knowledge

While the programming language research community is not well-versed in adoption phenoma, historical linguistics, public health, economics, and other fields have long studied it. Social scientists have developed both predictive models, providing insight into how adoption works, and practical methodologies prescribing how to intervene. For medicine, the research saves lives.

According to Google Scholar, over 40,000 papers directly cite Rogers' work on the diffusion of innovation, and Rogers estimates that at least 4,000 of the papers empirically validate his model [68]. His is just one model of adoption. Our presentation cannot be all-encompassing. Instead, we point out the implications for language research in four fields of study: longevity, diffusion of innovation, network models, and social constructionism.

3.1 Long-living Pariahs

How do we prevent a language from dying? Peyton Jones's motto for Haskell is to "avoid success at all costs [64]." He warns that lowering adoption barriers will hurt the ability to modify Haskell, so embracing adoption is an evolutionary dead end. In contrast Wadler and Meijer [53, 82] explicitly advocate lowering Haskell's adoption barriers. Temporarily ignoring the question of evolution, we examine longevity. First, users will abandon any system that provides insufficient value: do not ignore adoption. Less intuitively, we found examples in other domains where high adoption barriers *improve* longevity.

Maintenance. Completely ignoring adoption hurts the longevity of an innovation. Consider a simple model where languages provide value to users but every technology change has a cost (the *switching cost*). Balcer and Lippman [8] show that "there is a threshold such that the firm will immediately adopt the current best practice if its technological lag exceeds this threshold; otherwise, it (temporarily) avoids the switching costs and postpones." For an incumbent language to survive, it cannot fall too far behind its competitors in providing relevant utility to its users. Similar simple analytic models have been used to explore a variety of related phenomena.

Hypothesis 4. *Programmers will abandon a language if it is not updated to address use cases that are facilitated by its competitors.*

Strictness. One surprise is that seemingly harmful adoption barriers can strengthen a community. Quantitatively analyzing religions, Iannaccone [38] found that strict ones –

those that "destroy valuable resources or limit non-group activities" – can increase the commitment, participation, and ultimately, longevity of the group. For example, the Mormon church prohibits alcohol and caffeine consumption. Likewise, the Amish severely restrict the use of modern technology. These strict policies burden the constituents, yet Iannaccone found such barriers significantly strengthen the community.

The insight is that "strictness reduces free riding... potential members are forced to choose whether to participate fully or not at all. The seductive middle ground is eliminated, and, paradoxically, those who remain find that their welfare has been increased." The efficacy of polarization is surprising: "perfectly rational people can be drawn to decidedly unconventional groups. This conclusion sharply contrasts with the view, popular among psychiatrists, clinical psychologists, and the media, that conversion to deviant religious sects and cults is inherently pathological." Historical linguists, independently of Iannaccone, also found that social isolation increases the longevity of natural language, which they consider to be just another social construct.

Functional programming imposes a barrier to entry for programmers who originally learned procedural languages. It is possible that the social benefits of this barrier are partly responsible for the benefits usually attributed to technical benefits of functional programming.

Hypothesis 5. *The productivity and correctness benefits of functional programming are better correlated with the current community of developers than with the languages themselves.*

Burnham et al. [11] analyze additional ways barriers can be exploited to improve stability. Consider a marriage under stress: social and bureaucratic barriers lengthen the divorce process and therefore provide a window of time in which a bickering couple might resolve the issue. Likewise, Burnham et al. found that the retention of e-commerce users may be increased by introducing otherwise unnecessary barriers. If an upset customer cannot easily transfer his data to a competitor, he may change his mind or the system developer may have time to address the grievance.

Overall, we find that stability and longevity are well-studied. Unsurprisingly, adoption should not be ignored because a language must provide sufficient value to its users to survive. Less obviously, increasing adoption barriers can also benefit language longevity under particular circumstances.

3.2 Diffusion of Innovation

The process an individual goes through to adopt an innovation motivates the most extensively studied model of adoption: Roger's *diffusion of innovation*. In this model, diffusion is the process in which an innovation is communicated through different channels over time along a network.

The diffusion of *innovation* is distinct from the diffusion of *information* [29]. There may be a long gap between becoming aware of a technique or language and beginning to use it in practice. Sometimes, adoption never happens, even if the potential adopter is convinced of the possible benefits. Adoption is a non-trivial process. Similar reasoning shows that the change function model rediscovered by Meijer (Section 2.3) ignores important phenomena.

Hypothesis 6. *The diffusion of innovation model better predicts language and feature adoption than both the diffusion of information model and the change function.*

Rogers identifies five steps in the adoption process:

1. Knowledge: an individual is made aware of an innovation but has not yet investigated it.
2. Persuasion: an individual is interested in and seeking information about an innovation.
3. Decision: an individual evaluates pros and cons of an innovation and makes a decision to adopt.
4. Implementation: an individual employs an innovation and analyzes its use.
5. Confirmation: an individual finalizes the adoption decision, such as by fully deploying it and publicizing it.

Crucially, *adoption may fail or stall at any point of the process.* For a programmer to adopt a language, they must hear about it, understand if it is relevant, decide to act upon it, try it out, and then decide to continue. Importantly, the strategies that are most effective in one stage may be ineffective or even detrimental for later ones. For example, impersonal mass communication is good for providing initial knowledge, but peer communication is more effective at the persuasion stage.

Rogers found several factors that influence adoption:

- Relative advantage: the improvement over a previous innovation.
- Compatibility: how well an innovation integrates into an individual's needs and beliefs.
- Simplicity: how easy the idea is to use and understand.
- Trialability: how easy it is to experiment with.
- Observability: the ability to see results.

In surveying the literature, we found that programming language researchers concentrate heavily on the first of these, touting the relative advantage of their innovation. This focus was also noted by Meijer [53] in explaining his atypical efforts to address the second factor of compatibility (perceived costs). Research on domain-specific languages has illuminated some forms of compatibility, but as evidenced by Meijer's experience, there is much more to understand and do. Likewise, while designers often discuss simplicity,

mathematical or syntactic reductionism should not be conflated with the meaning here.

We should also consider the other three of the five important factors for adoption. Suppose Hughes succeeds in persuading programmers to try a functional language. Will they be able to apply it to their domain and observe compelling modularity benefits? Likewise, are sexy types sufficiently easy to use and understand, or is more work needed?

Hypothesis 7. *Many languages and features have poor simplicity, trialability, and observability. These weaknesses are more likely in innovations with low adoption.*

Observability is a particularly challenging adoption factor. For example, consider statically-typed languages. The tradeoff between static typing and comprehensive unit testing may change based on the size of the project. Users will do their initial exploration in the context of small programs, making it hard for them to perceive the benefits they would reap in a large software system. We suspect that limited observability of this sort is a barrier to the adoption of many programming language techniques.

Researchers outside of our field rely heavily on the diffusion of innovation model. For most adoption concerns, we therefore suggest starting with an investigation of the enormous body of empirical and analytic research under the model. Both its breadth and depth have been successfully demonstrated many times over.

3.3 Network Models of Diffusion

The diffusion of innovation model considers networks of adopters. Above, we focused on the decision-making of individuals. Here, we talk about the influence of network structure, for which Geroski [25] showed there are a variety of useful models. We focus on two examples: distinguishing different kinds of relationships between people and the competition between ideas.

Diffusion of information. Often, knowledge is not the missing ingredient. For example, a group of researchers (Kelly et al.) intervened in 1992 on the epidemic HIV outbreak in several American cities [41]. First, in a 3-night study of bars in the cities, they measured factual knowledge about HIV prevention and found that was already widespread. Despite acknowledging the fatal nature of the disease and the existence of effective protection measures, 31-49% of the surveyed at-risk residents reported engaging in extremely dangerous practices within the previous two months.

Hypothesis 8. *Most professional programmers are aware of functional and parallel programming languages but do not use them. Knowledge is not the adoption barrier.*

Homophily. The HIV researchers decided to use a social effect – persuasion by ones' peers – to spur adoption of protective measures. Instead of targeting individuals at random, the researchers exploited social network properties. They asked bartenders to identify popular customers over a period of one week. The identified *opinion leaders* were taught over four 90-minute sessions how to endorse risk-reducing behavior to their peers. In time-staggered interventions across the cities, the researchers found average dangerous behavior dropped by 15%-24%. This is a large impact and took relatively little time and effort.

Kelly et al. exploited several network properties when selecting change agents. For example, consider the model of strong and weak ties [28]. For any three individuals A,B, and C with two strong relationships (A,B) and (A,C), the third (B,C) is at least a weak one. Two powerful properties arise: people will form cliques based on strong ties, and these clusters will be connected using weak ties. First, that meant Kelly et al. could rely upon weak connections to rapidly spread basic knowledge. Second, Kelly et al. exploited the fact that people with strong relationships are similar (*homophily* [52]). The change agents selected by Kelly et al. understood issues specific to the community and were in an overall position of influence.

Like measures to reduce HIV risk the benefits of many programming language features are not directly observable. Likewise, there is a gap between what people know and what they do. Social factors, such as education and corporate policy, may widen or narrow this gap.

Question 9. *How can we exploit social networks to persuade language implementers and programmers to adopt best practices?*

Organizational innovation. Programming languages are often used in a commercial setting, where it has been useful to consider individuals of a corporation to form a structured social network. Sociologists and economists have long studied communication structure and innovation in organizations. For example, Minstrom [56] linked the "presence and actions of policy entrepreneurs" to legislative-level school choice reform.

As Wadler [82] suggests, researchers should work with enterprises. Well-studied considerations such as the size of the organization, its age and structure, and who is involved in the collaboration influences how a firm innovates. The choice of what firm to collaborate with and how might further benefit from understanding organizational learning [71] and tacit knowledge [60]. These concepts are useful even if a language designer does not personally interact with a firm.

Niches, ecological models, and DSLs. A powerful network model is Mark's [49] ecological theory. Studying music, he showed a genre can be modeled as competing for the time, energy, and preference of its listeners. The insight is that listeners enjoy and prioritize music that they can discuss with their friends, so music spreads along social lines. This creates a connection between an innovation and its demographic. Just as Jazz music is part of a broader Jazz culture, a domain-specific language should be considered in conjunction with the users of the domain. For example, scientific

users of Python are a sufficiently cohesive group that they created the SciPy series of conferences.

Ecological theories suggest that a DSL is more likely to be adopted if it solves a problem for a community rather than an equally sized group of socially disconnected individuals. For example, while SAT and SMT solving are general techniques, they are primarily used in niches such as verification that correspond to existing professional social groups.

Question 10. *How can we find and optimize for domain boundaries?*

The ecological model is also useful for considering the way that learning one language influences the decision to learn another. As evidenced by the popularity of web programming, modern programming is multilingual. If an individual is a member of overlapping communities, they may be familiar with many languages. However, due to the competition for time, a user can only strongly like – and develop deep expertise – in a small number of languages.

Question 11. *How many languages do programmers strongly and weakly know? Is there a notion of linguistic saturation that limits reasonable expectations of programmers?*

3.4 Social Construction of Designs

Adoption can improve innovations in ways that are hard to achieve in the laboratory. Researchers such as Backus [7] design languages according to *hard technological determinism* [74], which views individual technologies as inevitable and therefore can be designed without considering the social context. Sociologists, however, often examine an opposing model: society determines the design and acceptance of technology. Programming languages are *socially constructed*. For example, quantitative researchers showed that innovations evolve in a predictable way as they spread across different communities. Language designers that ignore social context are therefore missing opportunities to exploit the natural evolution of technology and do not even consider that designs may be systemically flawed by opposing the natural evolution.

Glick and Hays found social learning [26, 33] and adaptation [33, 68] guide the evolution (*reinvention*) of an innovation. Social learning is a powerful resource. For example, researchers have long examined its use in the social construction of legislation and other policies. Laws generally spread from state to state in the US, and Glick and Hays showed that, by observing early adopters, later adopters of a piece of legislation are often *more* innovative in their legislation because they build upon the experiences of earlier forms.

Norvig famously wrote that "design patterns avoid limitations of implementation language" [59], implicitly suggesting that language designers should heed developer patterns as constructive criticism. The research community has a good history of capitalizing on large-scale social learning, exemplified by recent research into support for mocking and dependency injection as a response to unit testing patterns. Social learning is a powerful resource – there are many more developers than researchers and they spend time on different things – but identifying and exploiting it is difficult.

Question 12. *How can feature designers more directly observe and exploit social learning?*

Adaptation is another aspect of social construction that is inherently difficult for the research community to address without participating in the overall community. Often, a group must modify an innovation in order to use it. For example, parallelism researchers have long studied how to schedule tasks across clusters. For web computations, long-tail activity (particularly sudden load spikes, such as the "Slashdot" effect caused by social media mentions) emerged as requiring innovation in elastic configuration. Such needs vary by domain; a researcher needs domain-specific knowledge. As an example, consider vectorization. In a recent study [48], Maleki et al. show that different compilers (ICC, GCC, XLC) greatly vary in which loops they can vectorize in practice. The authors conclude that "there is no universal criteria to determine what are the important patterns."

Question 13. *How can researchers, language implementors, and programmers cooperate to expedite adaptation?*

Ignoring the social context of adoption may lead to failure even when the innovation is needed by the target community. Generally, more comprehensive laws follow new, less developed laws, as predicted by social learning and adaptation. Counter-intuitively, however, Hays [33] also found that "states with greater societal problems respond with weaker laws." The societal context of a problem, such as whether it is controversial, impacts the design space of relevant solutions. Such understanding may help understand the relatively low traction for language-level solutions to security and parallelism.

Question 14. *Has controversy restricted the design space for socially-important programming language techniques?*

3.5 Summary of Pertinent Social Sciences

In our survey above for social science research relevant to programming languages, our challenge was not in finding any at all but sifting through an abundance. The examples in this section were primarily drawn from economics and, under the umbrella of diffusion of innovation research, studies from public health to religion to public policy.

Less discussed here, we found historical linguistics and computer supported cooperative work (CSCW) to also be closely related. Historical linguistics focuses on topics such as language spread and evolution, multilingualism, and textual corpora that spans millenia. CSCW studies the role of computational intermediaries and the human processes that

it augments. The security community, for example, has begun to view many of its core problems in terms of CSCW [5].

All of these social sciences are relevant, well-studied, and provide methodologies and insights missing from programming language research.

4. Socially-optimized language design

This section combines what the programming language community knows (Section 2) with what sociology has demonstrated in the last 50 years (Section 3). We outline a research agenda that can help us design better and more adoptable programming languages. Our presentation is divided into four parts: language adoption factors, feature adoption factors, new types of features powered by adoption, and programming language research norms.

4.1 Improving Language Adoption

As we discussed in Section 3.2, adoption by an individual is a multi-step process. Decisions on the part of the language designer can help or hinder adoption at each step. In this section, we focus on the last two steps, implementation and confirmation. In the implementation step, a potential user tries out an idea, and during confirmation, the user chooses whether and how to go forward with adoption.

Trial costs. The first costs a user will experience in implementing a linguistic innovation are trial costs. These appear in several ways. There is the cost of installing or using an implementation. There is also a learning curve, the time cost of familiarizing oneself with the language. This cost depends on the prior knowledge of the potential adopter as well as the language design. Minimizing trial costs means, for example, making a language similar to one that the programmer already knows and uses. An explicit monadic interface to standard libraries would be an adoption barrier for Java programmers but a familiar convention for Haskell ones.

As a result, language designers aiming for ease of use would benefit from more information about the past experiences and background of their target audience. This target audience might be quite different from the average programmer. Not all developers are ever inclined to try out a new language, and the ones who are might have a very different background from either language researchers, the average industrial programmer or enthusiastic mailing list participants. Crucially, as early adopters may have different backgrounds and tendencies from other demographics, guiding a language design solely by their experiences risks overlooking the needs of the larger, more conservative demographic. Characterizing and distinguishing early and late language adopters would be useful.

Question 15. *What sorts of programmers are early adopters of new languages and tools? What features and languages are they familiar with?*

Analysis and Confirmation. Once developers have committed to a trial of a new language, they will then examine it in context. Language designers often have strong beliefs about the desiderata of a programming language. Sometimes, they disagree. Convincing programmers to adopt a language means satisfying *programmers'* values, and so it is natural to find out what potential user populations value in a programming language.

To give a concrete example, some language designers believe that compile times are a major annoyance to programmers, and therefore design their languages for rapid compilation. For example, the Go developers made compilation speed a priority [4]. Other designers assume that lengthy compile times are acceptable. (For example, the C++ template mechanism allows arbitrary computation to be pushed into the compiler; language designer Bjarne Stroustrup views this as a useful capability for the language to have [75].) Where is the validation for either of these assumptions about user preference? For example, it is known that users of web search engines respond negatively to even a few hundred milliseconds' additional delay. Further, users respond negatively even when they do not consciously notice the delay [70]. Do programmers react similarly?

Question 16. *How averse are programmers to longer compilation or interpreter startup times? How willing are they to trade time for improved error checking?*

The answers here would be valuable since it would give designers more guidance in tuning their implementation. If users prefer immediate *concrete* feedback (from unit tests), that would suggest that designers ought to move some static checks off the critical compilation path. It might also turn out that developers care about how quickly their language environment flags mistakes. This would imply that code generation, rather than static checks, should be moved off the critical path.

Even the idea of the edit/compile cycle may need to be deprecated in exchange for live [51] and direct manipulation [31, 73]. User preference may vary from group to group — and here again, being a potential early adopter may correlate with particular preferences.

Question 17. *How does latency sensitivity vary across user populations?*

Hypothesis 9. *Developers primarily care about how quickly they get feedback about mistakes, not about how long before they have an executable binary.*

Latency is one place where empirical evidence would help improve the adoptability of language designs. Domain-specific-languages offer another. DSLs have become a prominent research topic. Many researchers focus on the reduced cost to implement a language. We suspect the reason users care about them is their low adoption cost. In particular, *embedded* DSLs will often leverage syntax, libraries, *and semantics* of the host language. If DSLs are to be the solution to parallelism [12], cloud computing [84], and other

software crises, we should move embedded DSL research beyond the ease of construction.

Hypothesis 10. *Users are more likely to adopt an embedded DSL than a non-embedded DSL and the harmony of the DSL with the embedding environment further increases the likelihood of adoption.*

4.2 Improving Feature Adoption

We now turn from improving language adoption to feature adoption across languages. Like language adoption, the feature adoption process can be analyzed through the lens of Rogers' diffusion of innovation model. However, while language adoption is carried out continuously by millions of programmers, feature adoption is conducted by a smaller number of language design practitioners. This population is likely to have its own culture(s), and the culture will in turn include a value system that is different from both language researchers and average programmers.

Trial costs. As before, we begin by discussing the trial phase of adoption: the adopter has been successfully persuaded to try out a feature. This can be a more complex process than trying out a complete language, since the feature needs to be integrated into an implementation of a language. Approaches such as languages-as-libraries [54, 76] may, therefore, have more long-term value in simplifying the essential presentation of ideas, rather than directly benefiting individual programmers that consume these libraries.

Hypothesis 11. *Particular presentations of features such as a language-as-a-library improve the likelihood of short-term and long-term feature adoption.*

Implementation and analysis. The relationships between different language features, and between features and implementation, are not always straightforward. They can be particularly obscure for designers who do not have extensive programming language experience.

Consider the BitC language. BitC is designed to be a type-safe language for operating systems development and backwards-compatibility with C. The designers chose to use type-classes for modularity. They found problems, however: "Research languages can adopt simplifications on primitive types (notably integers) that systems languages cannot." Trying to use a complex type system for primitives, in turn, put unacceptable pressure on operator resolution for primitive types. The language designers concluded that "type classes just don't seem to work out very well as a mechanism for overload resolution without some other form of support" [40]. In particular, building the input/output library proved far harder than they had initially expected.

Their conclusion took years of prototyping and experimentation. The trial cost for a theoretically-established feature – type classes – was high. Evaluating it by integration into a full-fledged language and using the new language and implementation to write a large body of code is hard.

BitC's scale of implementation is beyond the capacity of most research projects. Expediting evaluation is an important research challenge. It may be, for instance, that a handful of crucial use cases (such as I/O) account for the likely sources of trouble. If so, this is a piece of wisdom that ought to be captured for designers.

Question 18. *Can we ease evaluation of proposed language features? Can we catalog and predict the likely sources of trouble?*

Hypothesis 12. *Implementing an input-output library is a good way to test the expressive power and functionality of a language.*

Much functional programming research, according to Hughes, is about modularity. Modularity cannot be evaluated properly in small-scale experiments. It is intrinsically a phenomenon that matters most with large code-bases, long development periods, and multiple developers. Code reuse is always a human process, and typically a social process. Hence, understanding the social aspects is a fundamental part of research into reuse mechanisms.

Question 19. *Which other programming language features also require socio-technical analysis?*

Confirmation. Once a feature has been found feasible and valuable, it may still not be adopted. Many languages, including Scala, Python, and Java, have open processes in which proposed alterations to the language are discussed publicly [1–3]. All these community processes have a similar structure: a motivated individual or group writes a document describing a change to the language. The document is circulated and discussed. Eventually, some decision is made. (Different languages are controlled differently. For Python, ultimate authority belongs to Guido van Rossum. In other languages, such as Java, authority belongs to a committee.) For our purposes, the key point is the discussion and vetting of proposals is largely done in public, meaning that we can study the factors that encourage or discourage adoption of features in the language. This process would allow us to test several hypotheses.

Question 20. *To what extent do distinct language communities have distinct values? Are there values that are important in one community and completely irrelevant to another?*

Question 21. *How do the values of a language community change over time? For instance, do designers become more or less performance-focused as languages become popular? More or less focused on ease of implementation?*

The answers to these questions should matter to researchers who want their ideas adopted. Knowing what language communities value and how they assess proposals will enable researchers to better adapt their work to the perceived needs of these communities.

Even after a feature has been incorporated into a language, it may still be rejected at the implementation or confirmation stages of adoption. Most users may find the feature unacceptably difficult to use. For example, C++ supports arbitrary multiple inheritance, but coding standards will prohibit developers from ever using this feature [14, 18]. Eich's Law shows that partial feature adoption is expensive for a platform. "If you are liberal in what you accept, others will utterly fail to be conservative in what they send," and subsequent language design and implementation updates must be backwards compatible with the little-used feature.

One way to evaluate implementation of a linguistic feature is to see how often, and in what contexts, it is used [44]. Systematic study of open-source code repositories is one strategy for this. Another possible approach is to instrument the language implementation with data gathering. As yet, it is not clear whether the latter strategy has benefits to offset its privacy cost.

Hypothesis 13. *Open-source code bases are often representative of the complete universe of users.*

Hypothesis 14. *Most users are tolerant of compilers and interpreters that report back anonymized statistics about program attributes.*

4.3 Collaborative Features Powered by Adoption

Instead of just focusing on designing languages and features in ways that *bolster adoption*, we should also create designs that *improve with adoption*. According to Metcalfe's law, the value of a network goes up as the square of the number of users [72]. There are millions of programmers and billions of users; researchers should consider them to be an exploitable resource. We draw inspiration from the open source community, where modern languages come with third-party libraries, third-party tutorials, third-party forums, third-party consultants, and third-party frameworks and extensions.

In a simple thought experiment, we rapidly envisioned features that address many basic issues in programming languages by exploiting adoption. These included optimization, safety and correctness, configuration, robustness, abstraction level, and metaprogramming. As a few examples:

1. **Optimization**: An individual program is generally used by many users. Can programs speed up as the number of users increase, perhaps by augmenting tracing [24] with collaborative profiles?
2. **Safety**: Cooperative bug isolation [45] collects profiling data from many executions and analyzes the results statistically. This is just a starting point. For example, a key challenge facing symbolic execution researchers is finding an input that takes the program down a particular execution path. These tools are best at local exploration near an already known path, so collecting execution traces from many users would aid non-local exploration.
3. **Configuration**: Software settings such as for security and system defaults are inherently tied to social norms and practice [5]. Can languages streamline the design of open and configurable systems? Every use of a system is an additional example of typical use [21].

We have used these ideas in our own research – enabling search and verification of JavaScript applications through user traces – and are increasingly seeing it in work by others.

Collaborative features face technical challenges. For example, mass gathering of user data requires performance and privacy research, and exploiting the data introduces a source of complexity and distrust. Likewise, the design of such features are non-obvious. For example, if we use collaborative traces to verify software as part of its standard execution, we may need to reconsider what we mean by basic concepts such as sandboxing, which is no longer an all-or-nothing activity. We see that there are big opportunities for practical impact and basic research.

4.4 External Aspects: Improving the Research Process

In this section, we shift focus from language and feature artifacts to, instead, research methods and norms. As our community continues to resolve purely technical questions and outpace practice, examining social factors will become increasingly important for basic advances. We cannot rely upon outsiders to perform this research for us: sociologists and economists have their own work to do. Part of performing this research will be establishing scientific norms.

Sociotechnical analysis. The technical analysis of features should more routinely consider social context. Both Meijer and Jonathan Shapiro repeatedly found technical barriers to reusing standard functional programming concepts in richer settings [40, 53]. Simplified models such as extended lambda calculi inherently hide issues and therefore provide little assurance about technical relevance. It may be tempting and even useful to cast the challenge mathematically as feature composition, where we simply examine larger and larger calculi. This simplification is wrong. For example, language research into modularity must, at some point, connect to the phenomena surrounding how different people collaborate in putting together code.

Hypothesis 15. *Many programming language features (such as modularity mechanisms) are tied to their social use.*

Critical peer review of social factors. Markstrum [50] found that programming language research – including otherwise high-quality technical research – is rife with general claims backed by purely anecdotal evidence. Tolerating such claims propagates potentially misleading data and worse, may wrongly signal that the topics do not merit further consideration. Sociology is a science; we suggest treating claims about social factors in programming languages as such.

Supportive peer review of social methods. Wadler presented application-driven research as so under-served by the conference process [82] that it needed the protection of its own conference series. Drawing upon such momentum, we should also protect core sociological methods such as quantitative analysis, qualitative studies, and ethnographies.

Incorporating social methods into the research pipeline is a jarring change. In comparison, the computer human interaction community already considers social research important enough to solicit and accept for their main conference series [20]. As the research is not immediately applied and therefore objectionable to many, Dourish presented why the computer-human interaction community is learning to not short-sightedly judge research with inappropriate expectations on "implications for design" conclusion sections [20].

While the programming language community has and continues to develop powerful mathematical tools and understanding of languages, we cannot say the same for fundamental social aspects of languages. There is a blind spot in research methodology and values that has allowed a poor understanding of an important and basic foundation of programming languages to remain unaddressed for decades.

5. Conclusion

The programming language community has made enormous progress in exploring technical issues in language design and implementation. It has even made steady strides in understanding human factors for individual users. However, language designers are frustrated as to how to approach the systematic design of languages that users will adopt.

We showed social factors are a key overlooked foundation and that they can be scientifically approached. Social science research has much to teach us about addressing these recurring language design concerns. These lessons are not passive; we analyzed implications for core concerns and practices. Hoare's 1973 "Hints on Programming Language Design" [35] is not merely a classic: it has no identifiable successor. It is now 40 years later. We have a wealth of additional experience and data to draw upon; it is time to write an updated version. Synthesizing and validating the body of available information is necessarily socio-technical research.

By focusing on sociological perspectives of adoption, we have raised 21 open-ended questions and 15 more specific hypotheses about programming languages. These vary from directly actionable issues such as what data we should track or how to exploit collaboration in the design of basic language features, to longer-term research concerns such as our expectations for language use by programmers and even how to approach language design in a scientific way. The genesis of these questions demonstrates that focusing on sociological principles provides a way to advance the understanding and design of programming languages.

While this paper primarily focused on adoption, there are other aspects of programming language research that benefit from sociological insight. Programming languages facilitate communication and collaboration between language designers, programmers, and users. How does this happen and how can we improve it? The time is ripe for the principled examination of the sociology of programming languages.

6. Acknowledgements

This paper has grown over side-discussions with many language and system designers over the years, including George Necula, Armando Fox, David Patterson, Mark Wegman, Dan Grossman, and others. Richard Rhodes and Johanna Nichols helped point us to results in historical and sociolinguistics, and Coye Cheshire and Ashwin Matthew on technology adoption and evolution. Philip Guo, Philip Reames, and Kurtis Heimerl provided valuable feedback on various revisions.

References

[1] Index of Python Enhancement Proposals. `http://www.python.org/dev/peps/`.

[2] Scala improvement process. `http://docs.scala-lang.org/sips/`.

[3] The Java Community Process. `http://jcp.org/`.

[4] FAQ - The Go Programming Language. Online: `http://golang.org/doc/go_faq.html`, Accessed April 2012.

[5] M. S. Ackerman. The intellectual challenge of CSCW: The gap between social requirements and technical feasibility. *Human-Computer Interaction*, 15:179–203, 2000.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[8] Y. Balcer and S. A. Lippman. Technological expectations and adoption of improved technology. *Journal of Economic Theory*, 34(2):292–318, December 1984.

[9] C. Bird, B. Murphy, N. Nagappan, and T. Zimmermann. Empirical software engineering at Microsoft Research. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 143–150, 2011.

[10] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 331–344, 2004.

[11] T. A. Burnham, J. K. Frels, and V. Mahajan. Consumer switching costs: A typology, antecedents, and consequences. *Journal of the Academy of Marketing Science*, 31(2):109–126, 2003.

[12] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting

productivity and performance with selective embedded JIT specialization. In *First Workshop on Programmable Models for Emerging Architecture*, 2009.

[13] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *IEEE Software*, 22: 72–78, May 2005.

[14] M. Cline. Inheritance — multiple and virtual inheritance. Online: `http://www.parashift.com/c++-faq-lite/multiple-inheritance.html`, 2011.

[15] P. Coburn. *The Change Function: Why Some Technologies Take Off and Others Crash and Burn*. Portfolio Hardcover, 2006.

[16] M. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[17] W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 557–572, 2009.

[18] L. M. Corporation. Joint Strike Fighter C++ Coding Standards. Online: `http://www.scribd.com/doc/3969122/Joint-Strike-Fighter-C-Coding-Standards`, December 2005.

[19] E. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[20] P. Dourish. Implications for design. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 541–550, 2006.

[21] L. Fang and K. LeFevre. Privacy wizards for social networking sites. In *Proceedings of the 19th international conference on World wide web*, pages 351–360, 2010.

[22] M. Felleisen and D. Friedman. *Control Operators, the SECD-machine, and the [1]-calculus*. Indiana University, Computer Science Department, 1986.

[23] R. Gabriel. The rise of "worse is better". *Lisp: Good News, Bad News, How to Win Big*, 1991.

[24] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, 2009.

[25] P. Geroski. Models of technology diffusion. *Research policy*, 29(4-5):603–625, 2000.

[26] 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*, 53(03):835–850, 1991.

[27] P. Godefroid and N. Nagappan. Concurrency at Microsoft–an exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[28] M. Granovetter. The strength of weak ties. *American journal of sociology*, pages 1360–1380, 1973.

[29] D. Gruhl, R. Guha, D. Liben-Nowell, and A. Tomkins. Information diffusion through blogspace. In *Proceedings of the 13th international conference on World Wide Web*, pages 491–501, 2004.

[30] S. Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 933–946, 2010.

[31] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 91–100, 2008.

[32] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 94–105, 1994.

[33] S. P. Hays. Influences on reinvention during the diffusion of innovations. *Political Research Quarterly*, 49(3):pp. 631–650, 1996.

[34] C. Hoare. The 1980 ACM Turing Award lecture. *Communications of the ACM*, 24(2), 1981.

[35] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford, CA, USA, 1973.

[36] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III)*, pages 1–55, 2007.

[37] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[38] L. R. Iannaccone. Strictness and strength revisited: Reply to Marwell. *The American Journal of Sociology*, 101(4):pp. 1103–1108, 1996.

[39] R. James and A. Sabry. Yield: Mainstream delimited continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*, page 20, 2011.

[40] Jonathan Shapiro. Retrospective thoughts on BitC. Mailing list message. Available online: `http://www.coyotos.org/pipermail/bitc-dev/2012-March/003300.html`, March 2012.

[41] J. Kelly, J. St Lawrence, L. Stevenson, A. Hauth, S. Kalichman, Y. Diaz, T. Brasfield, J. Koob, and M. Morgan. Community AIDS/HIV risk reduction: the effects of endorsements by popular people in three cities. *American Journal of Public Health*, 82(11):1483–1489, 1992.

[42] S. Krishnamurthi, P. Hopkins, J. McCarthy, P. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[43] P. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2): 89–101, 1965.

[44] B. Lerner, D. Grossman, and C. Chambers. SEMINAL: searching for ML type-error messages. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 63–73, 2006.

[45] B. Liblit. *Cooperative Bug Isolation*. PhD thesis, UC Berkeley, 2004.

[46] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20 (8):564–576, 1977.

[47] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339, 2008.

[48] S. Maleki, Y. Gao, T. Wong, D. Padua, et al. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, 2011.

[49] N. Mark. Birds of a feather sing together. *Social Forces*, 77 (2):pp. 453–485, 1998.

[50] S. Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7, 2010.

[51] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[52] M. McPherson, L. Smith-Lovin, and J. Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, pages 415–444, 2001.

[53] E. Meijer. Confessions of a used programming language salesman. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 677–694, 2007.

[54] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for AJAX Applications. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–20, 2009.

[55] R. Milner. Logic for computable functions: description of a machine implementation. 1972.

[56] M. Mintrom. Policy Entrepreneurs and the Diffusion of Innovation. *American Journal of Political Science*, 41(3):738–770, 1997.

[57] A. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):6, 2009.

[58] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, 2007.

[59] P. Norvig. Design patterns in dynamic programming. *Object World*, 96(5), 1996.

[60] M. Osterloh and B. Frey. Motivation, knowledge transfer, and organizational forms. *Organization science*, pages 538–550, 2000.

[61] J. Pane, B. Myers, and L. Miller. Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 198–206, 2002.

[62] C. Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*, 2010.

[63] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceeding of the 8th working conference on Mining software repositories*, pages 3–12, 2011.

[64] S. Peyton Jones. Wearing the hair shirt: a retrospective on Haskell. *Invited talk at POPL*, 206, 2003.

[65] T. A. Proebsting. Disruptive programming language technologies. Talk at MIT. (Slides available: `http://ll2.ai.mit.edu/talks/proebsting.ppt`), November 2002.

[66] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 23–33, 2000.

[67] J. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3):233–247, 1993.

[68] E. Rogers. *Diffusion of innovations*. Free Press., 1995.

[69] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, 2010.

[70] E. Schurman and J. Brutlag. Performance related changes and their user impact. In *Velocity Web Performance and Operations Conference*, 2009.

[71] P. Senge. The fifth discipline: The art and practice of the learning organization: Book review. 1993.

[72] C. Shapiro and H. R. Varian. *Information rules: A strategic guide to the network economy*. Harvard Business School Press, Boston, 1999.

[73] B. Shneiderman. 1.1 direct manipulation: a step beyond programming languages. *Sparks of Innovation in Human-Computer Interaction*, 1993.

[74] M. Smith and L. Marx. *Does technology drive history?: The dilemma of technological determinism*. The MIT Press, 1994.

[75] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 4–1–4–59, 2007.

[76] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *PLDI*, 2011.

[77] D. A. Turner. SASL language manual. Technical Report CS/75/1, Department of Computational Science, University of St. Andrews, 1975.

[78] A. van Straaten. RE: What's so cool about Scheme? `http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html`, June 2003.

[79] A. van Wijngaarden. *Recursive definition of syntax and semantics*. North Holland Publishing Company, 1966.

[80] R. Von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: scalable threads for internet services. *ACM SIGOPS Operating Systems Review*, 37(5):268–281, 2003.

[81] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, 1992.

[82] P. Wadler. How enterprises use functional languages, and why they don't. *The Logic Programming Paradigm: A*, pages 209–227, 1998.

[83] W. Wulf, R. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *Software Engineering, IEEE Transactions on*, (4):253–265, 1976.

[84] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.