The Programming Language Wars

Questions and Responsibilities for the Programming Language Community

Andreas Stefik
University of Nevada, Las Vegas, U.S.A.
stefika@gmail.com

Stefan Hanenberg
University of Duisburg-Essen, Germany
stefan.hanenberg@icb.uni-due.de

Abstract

The discipline of computer science has a long and complicated history with computer programming languages. Historically, inventors have created language products for a wide variety of reasons, from attempts at making domain specific tasks easier or technical achievements, to economic, social, or political reasons. As a consequence, the modern programming language industry now has a large variety of incompatible programming languages, each of which with unique syntax, semantics, toolsets, and often their own standard libraries, lifetimes, and costs. In this paper, we suggest that the programming language wars, a term which describes the broad divergence and impact of language designs, including often pseudo-scientific claims made that they are good or bad, may be negatively impacting the world. This broad problem, which is almost completely ignored in computer science, needs to be acted upon by the community.

Categories and Subject Descriptors D.3 [*Programming Languages*]; H.1.2 [*Information Systems*]: User/Machine Systems — Software Psychology

General Terms Languages, Human Factors, Experimentation

Keywords The Programming Language Wars; Stability of the Academic Literature; Evidence Standards

1. Introduction

"It surrounds us and penetrates us; it binds the galaxy together."
- Obi-Wan Kenobi in Star Wars

Modern society has seen a significant transformation with respect to the discipline of scientific discovery. The advent of computers has led to thriving industries for creating new and innovative ideas. In Biology, chips that process datasets so large as to be unthinkable 20 years ago are now easily analyzed. Psychologists, medical researchers, physicists, and the entire software industry, regularly use computations as a core part of daily work. Without technologies that our community invents as the core foundation, innovations in some disciplines, like those the at the Large Hadron Collider, would be effectively impossible to evaluate. In a sense, programming languages bind the galaxy together, affording a crucial role that no other discipline can lay claim to.

As computer scientists, while we hold many roles in regard to these innovations, one is perhaps the most crucial: *we are the stewards of computation*. By steward, we mean that computer scientists, and especially programming language designers, have made products that are fundamental to advances in nearly all areas of science, massively influence academic computer science, and cause seismic shifts throughout the software industry. Languages that we create provide the foundation for how we explain to the computer what we would like it to do.

Unfortunately, as is becoming increasingly obvious to some, the computer science discipline has either been unable or unwilling to solve many of the basic problems in programming languages—perhaps notably, identifying the impact of specific language designs on people. As a result, we find our discipline in the situation where different developers and researchers spend time on a divergent set of language products. Due to a variety of complex factors we will discuss, we then find heated quarrels at different venues (e.g., academia, conferences, offices) where competing groups argue for or against particular languages or features—often with little evidence to support competing claims. Broadly, these and many other issues form the foundation of the programming language wars, causing a variety of social ills. This includes what in our view may be one of the most massive duplications of effort in human history.

Consider for a moment what actually occurs today in regard to programming language use in practice. In academia, students learn many language products, some of which are quickly abandoned, replaced, or changed. Educational products are built and pushed even if the inventors have never

Onward! 2014, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-3210-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661136.2661156

presented any evidence regarding the design's impact on programmers or learners. In professional communities, the problems hold similarities. For scientists in other fields, developers in the software industry, or students starting out, these individuals might assume that the designers have carefully vetted the design decisions with evidence, but an ongoing systematic analysis we have conducted on the literature has found that this is almost never actually true [43]. In effect, the simple fact that our major languages do not have evidence showing that their design makes sense is self-evident when we realize that, for novices, a randomly generated language is about as easy to use as Perl or Java [44].

We write this work because the future of science and the software industry depends, at its deepest level, on our community. Scientists and engineers in other fields we work with generally expect we are a scientific community, which uses evidence, creates theories, and uses the peer review process to promote science, not beliefs. Thus, while we think the language community's focus on the mathematics of programming languages is obviously appropriate for technical challenges, we argue here that this knowledge is necessary, but not sufficient, for solving the observed chaos in the software development world. Thus, we write here on what we call *The Programming Language Wars*, a concept we will describe throughout this essay.

What is this essay's contribution? This essay contributes in three ways. First, we describe the problems faced by the broad scientific community because of the programming language wars. Second, we define a set of questions to the programming language community that need to be answered if scholars are serious about solving, even partially, the major problems that exist in language design today. Finally, we discuss the responsibilities the broad computer science community has in regard to the language wars.

Structure of this essay. In the next section, we describe the language wars itself, framing the problem and discussing its many facets in regard to modern society, including its current foundation of evidence. We then define a set of responsibilities various groups or individuals have with respect to the language wars. Finally, the last two sections summarize and conclude this essay.

2. Framing the Language Wars

"Previous generations have been absolutely convinced that their scientific theories were well-nigh perfect, only for it to turn out that they had missed the point entirely."

- Pratchett, Stewart, and Cohen, The Science of Discworld

In this section, the primary purpose is to discuss, while recognizing that our explanation of such a complex social problem will not please all scholars, what the programming language wars are, including the roles people play in it. Using the language wars analogy, we then move to an analysis of the spectrum of views we have privately heard from scholars on it. Finally, we provide information on the current foundation of evidence in the language wars.

With this in mind, it behooves us to ask the obvious question—*what are the programming language wars*? While we decline to provide a formal definition, we think the language wars has at least three fundamental components: 1) language divergence, 2) language impact, and 3) language communities. While we describe these ideas briefly in the next few paragraphs, each is very complex and will be fleshed out as we continue this essay. First, we observe extreme divergence today in the field of programming languages. This **language divergence** occurs on a spectrum from the deceptively trivial (e.g., what should the syntax of a loop be?) to the debatably deeper (e.g., strong or weak typing? what kind of inheritance model?). Note that our claim is not that divergence is good or bad; only that it exists. Examples of this issue will be discussed at length throughout this essay.

Second, the language wars are related to what we term **language impact**. This potentially includes both positive (e.g., creative language solutions, features that positively impact human productivity) and negative impacts (e.g., duplication of effort, societal monetary cost). The concept of language impact is perhaps one of the most extraordinary mysteries in all of computer science and is poorly understood. This issue is at the heart of our section on the foundation of evidence in language design.

Finally, the language wars clearly involve **language communities**. By this we mean, they involve a variety of actors, with various roles. We think discussing these roles is important for understanding the rest of this work. Thus, we describe them next.

2.1 Roles in the Language Wars

"And we have just one world. But we live in different ones."

- Dire Straights, Brothers in Arms

We use the rather militarized term the "programming language wars" throughout this work to describe the situation in the software development world for two reasons. First, the term implies an adversarial role in the community, or at least one of competition. Second, while it is almost never discussed in the peer-reviewed scholarly literature, we feel that, despite its lack of formalization, many in the community have heard the term informally.

As part of the language wars, it makes sense to think about the roles the various actors have and how these roles impact the success or failure of language products. In essence, we acknowledge here that many individual participants of the programming language wars, which naturally encompasses a broad range of people (e.g., students, entry level professionals, CEOs at major corporations), harbor different motivations for their participation. While our list is hardly intended to be all encompassing, we elucidate it here

as an exemplar of the possibilities, beginning with what we term **owners** and **followers**.

First, the concept of an owner is literal: we classify this role as one that owns a programming language. Examples would be Microsoft, which owns C#, or Oracle, which owns Java, or the primary developers of a smaller language, like Ruby. Owners, by definition, have a vested interest in the success of a language product (e.g., fame, interest, belief, money). Contrasting owners, we have followers, by which we mean any individual involved in a language, regardless of success (e.g., promoting it, using it) or failure (e.g., denouncing it). While followers may not have the same motives as owners, they may also have vested interests. For example, those that learned programming using C, and that are familiar with its syntax, might be more willing to adopt a C-like language such as Java.

While owners and followers may have their own motivations, we define three additional roles that may be played by either group: 1) **believers**, 2) **dependents**, and 3) **volunteers**. In the first case, we imagine believers as those that find a particular language's design convincing. This might hypothetically be an individual that thinks that C# is better than Java (or vice versa), that lambda functions are important for a language to have (or not), or that a particular syntax is elegant (or not). Belief by such individuals may or may not be based on evidence.

A second role is that of dependents. In this case, we take these individuals as those that need a particular language to complete their work or that otherwise require the success of a language for an external reason. For example, if a software development house has a million lines of, for the sake of argument, correct PHP code written in a server back-end, then they are dependent on the success of PHP financially (Facebook might serve as an exemplar). This can be the case regardless of whether a group owns the language technology. We call those in this category *financial dependents*.

In this same respect, students are also dependents in the sense that they are typically required to use language products chosen for them by faculty. While this is normal, it is important to recognize that it sometimes takes students years to learn a language like C++ competently. As such, with that time invested, students are dependent upon the success of the language or its constructs. We call individuals in this situation *intellectual dependents*. Obviously, the two are not mutually exclusive.

Finally, the logical opposite of the dependents is the volunteers. In this group, programmers may experiment with, or otherwise use, a programming language for their own reasons. Such users may fall into categories like early adopters, trying Apple's Swift because they want to. Similarly, they may enjoy using a variety of language products for personal reasons. The point is, volunteers are those that have little practical vested interest, financially or intellectually, in the success of a language product. There may be a spectrum between dependent and volunteer.

While we imagine it is obvious to most readers, we want to point out that our description of roles barely scratches the surface. We mention them at the relative beginning of this paper because we think it is important to acknowledge that the programming language wars has participants. While our roles are not exact, they do give us a rough baseline by which to think about the various groups. With this in mind, we can now move to what we call the language wars spectrum: possible outcomes or results of the language wars.

2.2 Possible Outcomes of the Language Wars

We begin this discussion with two diverging points of view, each being an extreme position we have heard from scholars toward what would stop the programming language wars. The first view we call "One Language to Rule Them All," the idea where the only possible resolution to the language wars is for all possible developers under all possible conditions to use the same language. Next, we move to the opposite of this view, which we will call "Unique Snowflakes," where all developers, under all possible conditions, are so unique that they need their own programming language. We point out these extremes for two reasons: 1) we actually hear claims like this from language designers, scholars, and developers, [1] and 2) such views are difficult for us to believe without supporting evidence—by discussing them we become aware of the fact that more balanced points of view need to be considered. This leads us to our first research question:

RQ	In the future, what point on the spectrum between "One Language to Rule Them All" and "Unique Snowflakes" is desirable?

2.2.1 One Language to Rule Them All

"One Ring to rule them all, One Ring to find them, One Ring to bring them all and in the darkness bind them."

- The Lord of the Rings

One belief we have commonly heard amongst programming language designers, scholars, and users, is the mistaken view about the language wars that any solution must account for all possible tasks under all possible conditions. We reject this view, finding it little more than a characterization of possible outcomes of the language wars. In this section, we discuss this belief, which we call "One Language to Rule Them All."

Consider for a moment why the One Language to Rule Them All belief is invalid in its purest form. We see many possible reasons, but point out just a few we consider important, namely: 1) humans vary, 2) problem domains vary, and 3) even a perfect language, if it were created (somehow), may not be adopted. First, while some developers

[1] Out of professional courtesy, we ask readers not to ask us which individuals have made such claims; we decline to say.

will take the obvious fact that developers have natural variation in their needs and abilities as arguing for the Unique Snowflakes view, it is important to recognize that developers vary, while also recognizing that this variation is not as high as many believe. For example, in the authors' studies on language design, we typically post demographic information in our experiments. However, while we talk about it little in these works, it seems to us that variation amongst most samples are less important than the wide demographic variance.

For example, consider that children in school in many countries are now learning computer science at an increasingly younger age, in part because organizations like Code.org have helped bring awareness, but also because many educators realize that they live in the 21^{st} century—programming is now a fundamental and basic skill. Many young students likely want to solve problems their elders do not, like perhaps wanting to learn how to program animations or games, or any other variety of activities. On the other hand, professional programmers working at NASA obviously have different needs, which probably require different languages or libraries. Put more crudely, we harbor doubts that almost any serious application created by professionals is done in Alice or Scratch, let alone the next generation of space shuttles, but we also recognize that children might learn the foundations of programming with such tools. This brings us to a research question:

RQ	What language designs help the largest user base possible under the greatest number of circumstances?

Second, separate from needs of users, we need to consider that scientists and private industry uses programming languages for a wide-variety of goals. This includes applications as varying as bio-engineering, physics, to video games or movie editing, amongst many others too numerous to mention. While it is clear that many applications can be created with languages that exist today (e.g., many applications are written in Java or C++), some application domains may benefit from having a unique programming language. While many potential areas could be pointed out, we think one obvious one is in parsing, where tools like Antlr provide concise ways to represent parsing rules that can be genuinely complex when written by hand in a language like C or Java. We admit, however, that we know of no concrete evidence on this point, nor can we ascertain with any kind of mathematical certainty exactly which domains may require a new language and which do not. We suspect that, when history is written, the number of times authors have claimed a new domain specific language was needed will be considered a significant exaggeration.

From another perspective, with our third point, Meyerovich and Rabkin [32] point out that why a programming language is adopted is a complex question involving many factors, including some as seemingly mundane as what language one previously programmed in—evidence for what we previously called intellectual dependence. Even a somehow perfect language may not be adopted. The reasons for not adopting such a perfect language might, among many possible factors, depend on market observations or network effects. For example, if developers assume that a certain language will be maintained for decades or is in common use, working with it may be perceived as more reasonable, despite perceived or actual imperfections. This leads us to the following research question:

RQ	Which domain specific languages benefit humans?

Such a question implies a number of different perspectives. This includes at least estimates regarding the future of the language, availability of developers, learnability, usability, or other concepts. To give a specific example, we know of no evidence that the programming language R makes statistics work easier to accomplish than having a statistics API, with similar features, built into another language. Given this, whether R is needed requires more evidence than proofs that a particular statistical calculation, or language feature, works or is fast.

2.2.2 Unique Snowflakes

The Lord said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."

- Genesis 11:6–7, New International Version

If we take the logical opposite of One Language to Rule Them All we obtain what we call Unique Snowflakes—the idea that all developers need a unique programming language for all possible tasks. Before discussing the issue, we point out that this view is refuted by empirical evidence in the literature. We discuss two issues, namely that 1) it is not supported by existing evidence and that 2) erring too close to the Unique Snowflakes view in society may have negative consequences, especially for the scientific community.

First, and most crucially, we think there already exists evidence in the literature showing the Unique Snowflakes view is unlikely to be valid. Consider for a moment what the empirical data would have to look like if it were to be reasonable. Namely, developers need to vary so widely that no consistent measures can take place—this view is not what we observe from careful measurements of programmer behavior so far.

For example, consider the study by Rossbach et al. [39], which showed nearly an 8-fold difference in bug rates between a software transactional memory based solution when compared to course or fine grained locks. Or, consider that,

in nearly all replications of studies on static or dynamic typing (e.g., with or without a development environment, with or without documentation, the use of generics), that static typing appears to increase programmer productivity on average [12, 25, 31]. In studies of novices, differences in syntax have significant explanatory power, as judged by the eta values reported by the studies, in saying why we observe that some languages are easier to understand and use initially than others [44]. Put succinctly, every study, and every future study, that documents that programmers on average do better/worse with one language feature over another increasingly refutes the Unique Snowflakes view of the world.

Second, even if all the previous evidence is incorrect, biased, or flawed in important ways, this does not automatically imply that we should promote the Unique Snowflakes view. Modern society requires programmers work together and it seems all too obvious that if all programmers had their own language, collaboration would be more difficult, if not impossible. Put another way, even if people have measurably unique needs under all or many conditions, this does not imply that society obtains a tangible benefit from extreme language divergence.

We think one tangible area that may not benefit from divergence is syntax. Historically, while few question the creativity associated with creating new language products, it is important to recognize that papers often study one feature, while simultaneously combining interesting and new ideas with modifications to old ones. For example, many programming languages vary how to write a loop, without evidence that the approach is "better" nor often even with public reasons why the syntax is different from other alternatives. We think scholars and practitioners need to demand more evidence from scholars on such issues, especially when language designers make claims. This is especially true given documented evidence from several studies showing that syntax has a significant impact on novices [8, 9, 44] and that compiler error design impacts professionals [40].

2.3 What is the Societal Impact of the Language Wars?

Another issue of importance in regard to the broader programming language wars is its impact on the world at large. This effect is difficult to quantify, in part because computer scientists often seem to find talking about the language wars to be taboo in peer-reviewed research, and in part because little scientific research has been done on the language wars as a phenomenon itself. In this section, while first admitting that evidence on the societal impact of the language wars is difficult to gather, we take a first attempt at discussing what we see as the consequences of this problem. We focus our efforts on issues we think are important to academia and industry, including duplication of effort and cost.

In academia, we think few computer scientists have come to terms with just how fractured the programming language community is. Consider a short survey undertaken by the current authors. While we know of no systematic analysis of the types and kinds of programming languages used at colleges and universities across the United States, or the world, it may be enlightening to capture this information on a global scale. While an admittedly inadequate start, we began such an analysis by conducting a short survey of 39 colleges and universities in the mid-west, analyzing only what languages were used by these schools in the introductory course. Of these, approximately 9 programming languages were used, including C, C++, Alice, VB, Python, Java, C#, COBOL, and ADA. Evidence that each of these choices make sense, as a whole or in terms of individual language features, cannot be established from the literature.

Differences in instruction within academia aside, industry also is highly fractured in regard to its use of programming languages, both in regard to variations in adoption (as has been studied recently) and in regard to libraries and toolkits. For example, in NetBeans 7.4, we counted a total of 466 different toolkits as being available for use as Javascript libraries. Such toolkits, which are basically small additions to Javascript, provide developers capabilities out of the box for coding on the web. To most scholars, this should come as no surprise and the situation is similar for other languages. In practice, when developers adopt a language, they generally garner access to very large standard libraries and extensions, for good or ill. Some may consider this just an expression of our community's creativity and on the surface this view may seem appealing. We are concerned, however, that such large divergence is lowering productivity on a societal scale.

Second, given the number of languages, toolkits, and frameworks, it seems clear that there is duplication of effort in the programming language community. As a trivial example, while we think that academic circles themselves often have checks and balances in the peer-review process to ensure originality with scholarly papers, this seems to be virtually non-existent with the software our community creates. Consider that for new languages, constructs such as hash tables, lists, networking APIs, speech APIs or other available libraries are reimplemented, leading us to our next research question:

RQ	Of the time used in developing programming languages, what percentage was spent reinventing the same solution (e.g., a HashTable in Java vs. C#)?

While there is little debate that changes to languages can have benefits, sometimes even minor changes to new languages (e.g., differences in loop structures), or in some cases patent or copyright issues, impact whether parts of a standard library can be reused. Naturally, this leads to corporations like Oracle (or Sun) creating the entire JDK, with others creating all of .NET, an obvious duplication of effort. When we look broader, at the C++ standard libraries, libraries for Ruby, Perl, Python, or other technologies, and

additionally look at APIs that developers in smaller projects reimplement over and over again in different languages, we are suspicious the community as a whole would benefit if duplication of effort were reduced across technological sectors.

For software companies there is a different perspective, which depends on market constraints. If, for example, Microsoft decides to develop a language such as C# as a competitor of the programming language Java,[2] the duplication of effort must have initially been considered worth the cost. However, for single developers that actually use one language and need to write similar code in a different one, the corresponding duplication of code becomes problematic. It seems to be especially serious as soon as multiple clients are required to write the same or similar code snippets in different languages—in other words, while companies may debatably benefit from duplication, clients may not. This leads to another research question:

RQ	Of the whole time used in software development, what percentage was spent reimplementing the same solution in a different language?

With the idea in mind that many language designers and adopters of new languages have reinvented the wheel many times throughout history, we also want to point out that it is not entirely clear whether this duplication of effort was worth it. For example, we harbor doubts that yet one more implementation of a hash table, in yet one new language, should automatically be considered positive. Similarly, we harbor doubts that yet one more way to write a loop in core language syntax is worth investigating or funding, unless the new method also takes into account the language wars broadly—using evidence to make decisions or changes. Before we can begin to ask whether an investment is worth it, though, we need to begin measuring it, leading us to the next question:

RQ	What is the total cost throughout history of developing all computer programming languages, supporting libraries, and tools?

Given that many languages exist, one other concern we have about the programming language wars is that it seems reasonable to assert that groups that change programming languages, which is common because of the speed at which our discipline changes, must ultimately redo work and adapt their technologies (e.g., porting). In some cases, such porting procedures are straight-forward. For example, moving between new versions of Java or C++ may require effort, but would be arguably less effort than converting from Fortran to Perl. In academia, the situation is similar. When faculty members change language products, say from C++ to Java, as has been common in recent years, textbooks, tutorials for parts of the standard libraries, slides, assignments, and examples all must change to the new approach. This leads us to our next research question:

RQ	What is the total historical productivity lost by switching/updating programming languages in various settings (e.g., academic, industrial, military)?

Further, if we imagine that, over time, language designers and academics became increasingly reliant on valid and reproducible evidence in making decisions, it stands to reason that at least some aspects of language design would cease to be replicated in new languages. For example, while our tests only encompass a small set of possible solutions compared to the whole, the empirical data we have gathered leads to some conclusions. Consider the four alternative looping constructs listed in Figure 1, which shows outputting the word "Hello-" and a counter variable on the screen 10 times.

New language designers might think changing syntax for a loop is an expression of creativity, but we are suspicious it is more in line with ignorance of the available evidence [8, 9, 44]. While we mean no language in particular, a mathematically correct and reasonable language may very well be hard to understand. Given the wide proliferation of products we create for academia and the software industry, whether a construct is clear is a decision that should be based on hard evidence, not just opinion and argument. In this case, evidence in studies is clear—the word "for" makes little sense to novices, for reasons that should be obvious, compared to words like "repeat." More crucially, if the syntax of a language is different, the standard library must also be different, often leading to duplication of effort and reinventing the wheel. This line of reasoning leads us to the next research question:

RQ	If language designers created languages with an increasingly larger evidence base, what would the impact be over the next century?

Overall, our point in this section is that the language wars is having a significant impact on society as a whole. From our perspective, the impact is largely negative. While societal issues are never solvable by one research group, and the language wars is a deep problem with many factors, investigating the impact on society needs to be done if we want future generations to not look back on our work with surprise—wondering why it took us so long to even run a study on, for example, the syntax of a loop. This issue of the foundation of evidence is the next topic of discussion in this work.

[2] see for example corresponding articles such as `http://news.cnet.com/2100-1001-242268.html\&title=Microsoft-brewing-Java-like-language\&desc=--+CNet+article`

```
i = 0
repeat 10 times
  i = i + 1
  output "Hello-" + i
end
```

(a) Quorum

```
hello(X):-
  (X > 10, !);
  (write('Hello' - X), Next is X+1, hello
      (Next)).
:-hello(1).
```

(b) Prolog

```
1 to: 10 do: [:i |
  Transcript show:
    'Hello-', (i asInteger)].
```

(c) Smalltalk

```
for(int i = 1; i <= 10; i++) {
  System.out.println("Hello-" + i);
}
```

(d) Java

Figure 1. Loop styles in various programming languages that output a string and a counter variable.

2.4 Language Design Needs a Stronger Foundation of Evidence

"You know nothing, Jon Snow."

- Ygritte, Game of Thrones

In recent years, many authors have shown increasing concern about the stability of evidence in academic computer science, including programming language design. Before diving into language design, however, let us first take a slight detour to the field of software engineering, as authors from this community probably started the trend toward reanalyzing the history of our evidence practices, with a critical eye.

In the mid-1990s's, Tichy argued that the field of software engineering was in disarray, largely because authors were continuously publishing non-evidence based work. By non-evidence based, what was meant was that even prestigious journals were publishing what amounted to opinion pieces, not backed by data and observations [50]. As Tichy states:

> *There are plenty of computer science theories that haven't been tested. For instance, functional programming, object-oriented programming, and formal methods are all thought to improve programmer productivity, program quality, or both. It is surprising that none of these obviously important claims have ever been tested systematically, even though they are all 30 years old and a lot of effort has gone into developing programming languages and formal techniques.*

This claim by Tichy was not one of opinion, stemming from a careful reading, and cataloging, of the literature in comparison to other disciplines. There is no nice way to say it—Tichy showed evidence that software engineering as a discipline was exhibiting pseudo-scientific practices. Other disciplines were using data and observations, while software engineers were using anecdotes, with sometimes authors arguing against even the idea of experimentation. As Tichy notes, many at the time argued that experimentation was, "inappropriate, too difficult, useless, and even harmful"—arguments we feel match closely to those made by some language researchers today. Recent work exploring similar ideas found that the field of software engineering is still lagging considerably behind in its academic rigor [26].

While the evidence practices in software engineering have been investigated since the mid-90s, this has not yet been fully completed in the programming language design community. For example, while some are familiar with works summarizing the literature on novices [23] or end users [27], these works talk about what is in the literature, but do not investigate whether what is in the literature is correct or exhibits stable knowledge that can be relied on. In the literature, however, several scholars have claimed that our community has similar problems. This includes at least Tichy, Markstrum [30], and Hanenberg [16].

To our knowledge, however, the first work to attempt to formally document the evidence practices of the programming language community was conducted only recently by the current authors and our research collaborators [43]. This work is preliminary, only evaluating scholarly workshops on language design, but the results showed clear evidence that the community was not using high quality evidence gathering techniques. Work on more prestigious venues, like OOPSLA, ICFP, or others, are in progress, but our suspicion from looking at the approximately 1700 other papers we have cataloged so far, in addition to those discussed in the cited work, is that the evidence practices may in fact be worse in these venues. In short, with notable exceptions, the programming language community has used little empirical data regarding the impact of products used by millions of people in nearly its entire history. This unfortunate observation forces us to seriously question the scientific validity of any claim made by a language designer that does not have actual data in hand that we can double check and replicate. This leads us to a research question in regard to the evidence practices of the programming language community as a whole:

RQ	How can we increase the programming language design community's evidence standards?

We readily admit that we do not know the answer to the previous question, but do want to offer some limited speculation that others can freely explore, hopefully to refute or improve our guesses. One possibility as to why the language community has not been gathering rigorous evidence, beyond technical considerations, is that in academia, there is little incentive to do so. Papers can be published perhaps even more easily with small incremental changes to existing work. Further, given that academic institutions often rely considerably on paper counts, literally the number of papers an academic publishes as a metric, academics are directly discouraged from working to alleviate hard and long-standing problems that might take years of hard study if it leads to only one, or a few, publications. In other words, while we admittedly have no direct observations, we wonder whether modern methods for evaluating tenure and promotion are having a negative impact on the quality of work in the academic literature.

Evidence issues aside, while the literature is currently sparse, given that the current authors have had considerable contact with the language community, we want to be clear about one fact: from our perspective, while there may not be much evidence in the literature in regard to issues like language impact, we do think language designers *care deeply about such issues*. We think one way in the future that it might be possible to thus make progress is in the creation of a formal catalog of language features, their impact on people, and standard reproducible tests, with corresponding evidence. For example, in psychology, scientists have created the DSM-5 [1], a rigorous collection of assessment procedures for a variety of psychological conditions. With programming languages, creating a standard collection of all assessments ever done on type systems, syntax, or other features and documents on how to easily replicate them, could go a long way toward giving language designers reliable, replicable, scientific information about impact. This leads us to our next question:

RQ	Can we create a catalog, using the DSM5 [1] as an exemplar, for reliable tests on language impact?

3. Stewardship and Responsibilities

"Now naturally, like many of us, I have a reluctance to change too much of the old ways."

- Kazuo Ishiguro, The Remains of the Day

Given that language designers are the stewards of computation in the sense that our community develops foundational products for which the world's software is built upon, it makes sense to ask what our responsibilities are. In this section, we look at the programming language wars from a different point of view. While recognizing that this problem may be one of the most challenging in computer science, which does not have an easy or straightforward solution, we discuss here a possible path forward. In this context, we discuss how various communities can contribute to alleviating aspects of the language wars over the next century.

Thus, we look now at individuals and communities, discussing their role in the language wars, including where they have succeeded or failed. Throughout this process, we provide our view on possible activities they could undertake that may help in the long run. While it should be obvious, we do not have all the answers, nor do we pretend to have some type of guarantee that our suggestions will be beneficial. We point out, however, that many communities are doing little to investigate or fix a major social ill in our discipline.

3.1 Responsibilities of Individual Researchers

Individual academic scholars have an array of responsibilities at modern universities, including their duties in working toward solutions to genuinely hard problems. We argue in this section that individual scholars and researchers have a crucial role to play in the language wars. In this section, we provide tangible suggestions that we think may help individual scholars contribute.

3.1.1 Provide Evidence of a Problem's Impact

"Tell me one last thing," said Harry. "Is this real? Or has this been happening inside my head?" [...] "Of course it is happening inside your head, Harry, but why on earth should that mean that it is not real?"

- J. K. Rowling, Harry Potter and the Deathly Hallows

Computer scientists and programming language researchers are adept at constructing examples that illustrate a situation that could be considered as a problem. Typically, a problem statement is used as a springboard for explaining a potential solution. As an example, consider the first paper about aspect-oriented programming [24], which stemmed from a background in programming language design, formal methods, meta-object protocols, gray box testing, compiler construction, etc. While there was a history to why AOP was developed, we can find no evidence in the literature that the original problem was based on documented evidence. Of course, we understand all too well why this is often the first step in science. However, while thinking about a potential problem is an important *first* step, modern science in other disciplines, when publishing at major journals, usually requires authors go deeper, a step sometimes missed in our discipline. This brings us to the first responsibility of an individual scholar in language design:

Resp.	Problem statements should include reproducible, verifiable, evidence that the stated problem has impact.

There are different ways to identify whether a problem has impact, including at least: 1) controlled experiments, 2) field studies, 3) code repository analyses, 4) surveys, and other techniques. All of these methods permit us to give evidence that a particular situation has enough of an impact that it can be detected and replicated by other research groups. Such techniques can often confirm, and importantly refute, that a problem matters, which can be tested by trying to detect harm. For example, if scholars identify loss of performance in runtime behavior of computer systems, loss of productivity in human software developers, decreased learnability with students, loss of maintainability, or other factors, these would qualify. Generally, while it should go without saying, the greater the language impact, seemingly the more important the problem may be.

3.1.2 Provide Evidence that a Solution Helps

"I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."

- Deep Thought in Douglas Adam's Hitchhiker's Guide to the Galaxy

If we continue with the AOP example, we find that aspect-orientation is a technique that is now commonly discussed in the literature. Some in the research community have argued that its success is paradoxical [45]. On the one hand, it seems clear cross-cutting concerns are real—we really can come up with examples, like logging, where an AOP solution feels cleaner or nicer. Or, we can discuss large code bases and the potential need to weave code throughout them, which feels appealing to some scholars.

However, consider empirical investigations into aspect-oriented programming conducted by scholars that are independent from the original research group [2, 11, 12, 17]. In such a case, when the situation is considered more carefully and actual developers are asked to program using the new techniques, the claimed benefits of AOP are not supported. Careful measurements refute the AOP community's opinions that such techniques are helpful, which should give pause to any scholar thinking carefully about whether the original problem was impactful or that the proposed solutions help. Oddly, however, the first at scale experiment from an independent research group on AOP appeared to take place more than a decade after the original paper. By that point, young scholars working toward tenure and promotion, if they were involved in such a community, would have a vested interest to ignore results showing the research direction does not solve the original problems—changing research directions can be difficult and risky.

Consider another example, in this case the use of generics. The study by Parnin et al. shows how frequently generic types in Java are used [34]. While the technical aspects of generics were published at a large variety of venues (see e.g., [3, 21, 33] among many others), the results of this study indicate that generics are used relatively little—although it should be mentioned that in an extended version of the study, the authors see at least some upward trend [35].[3] Other examples of such studies are the one by Gil and Lenz [15] where the authors study the frequency of method overloading in class definitions or the study by Callau et al. [5], where the authors study the use of dynamic language features in Smalltalk.

One interesting characteristic of the previous studies is that language constructs are being tested long after they were invented. Yet, even before these studies, many claimed that the constructs were helpful and in some cases they had world-wide impact. To give an example from today, many programming languages are now integrating features from functional programming. Yet, when we analyze every paper ever written at ICFP, which we have done internally and coded formally, this community consistently argues that functional programming is beneficial, yet also provides no evidence, other than purely technical definitions—necessary, obviously, but not sufficient. Tichy appears to be correct; our communities are integrating features without evidence. This leads us to our second responsibility:

Resp.	Individual scholars need to significantly increase their evidence standards.

3.1.3 If it Ain't Broke, Don't Fix it

"So, I guess, like, now we just have to start over and start rebuilding everything like our houses, but ... I was thinking maybe instead of houses we could live in teepees, cause it's better in a lot of ways."

- Richie Norris in Mars Attacks

In the previous section, we discussed that individual scholars should challenge themselves to provide an increasingly larger amount of data and evidence in academic papers, especially in regard to evidence with whether a language solution has impact, works, and actually helps the community as a whole. In this section, we turn to a different problem that must be considered by individual programming language researchers and scholars, namely: only change those language features for which there is direct observable evidence that they should be.

Consider a hypothetical example. Suppose a language designer has just invented a new programming language feature. This feature, it is claimed, is beneficial. Assume as

[3] In terms of human impact, one study showed that generics help developers who use an API that uses generics in comparison to raw types in the interface description (see [19]).

well, for the sake of argument, that this scholar followed our guidelines from the previous sections. They have provided a formal proof that the feature worked, conducted a randomized controlled trial with human beings showing the feature had a positive impact, and conducted surveys with industry partners, which collectively provided a solid foundation of evidence. From this, we would conclude that the researcher has done their due diligence, plausibly obtaining several publications and doing their job as a scholar adequately.

Now suppose, however, that this scholar expands their work, embedding it into a new language with their own compiler. For building such a compiler, the scholar has at least two options: 1) recognizing that it is not always possible, build the feature into an existing language with the feature embedded, or 2) to construct a new language that includes the feature. Given our observations of the research literature, and programming languages in the field, we harbor doubts that scholars are making sensible choices here, which we exemplify by starting with a discussion of Java and Boo.

Let us first consider Java, where we find a number of different ways to iterate over a collection (e.g., for-loop, while loop over an iterator, foreach syntax since Java 5). These methods harbor the same semantic goals, to iterate, but are syntactically different. When the language designers change the language, features like this seem to appear when new versions arise. Although it seems commonly accepted that the main goal of Java 5 was the introduction of generic types, for example, new syntax constructs appeared with no backing evidence for or against their use [19].

Consider as another example the language Boo and its blueprint Python. First, Boo explicitly advertises differences such as the static type system and type inference. As is now known, there is some evidence that static type systems, at least those with a declarative type system, help developers be more productive under a wide variety of conditions (e.g., with an IDE, without an IDE, with documentation, without documentation) [12, 25, 31, 37, 42, 46–48]. Hence, while the inventors of Boo may or may not have known the evidence, this decision appears to be supported.

Second, while the Boo authors, so far as we can ascertain, provide no evidence, Boo also introduces a number of accidental or spontaneous changes such as different usages of imports. [4] Or, Boo has what it calls Generator Expressions, a "for in" loop, to use the author's terminology from the Boo Manifesto. Further, Boo allows the programmer to turn off the static type system (so-called Duck Typing), a decision not supported by the literature on type systems. The Boo author describes this as follows:

> *Sometimes it is appropriate to give up the safety net provided by static typing. Maybe you just want to explore an API without worrying too much about method signatures or maybe you're creating code that talks to external components such as COM objects. Either way, the choice is yours not mine.*

Taking into account that the previously mentioned studies show that static typing helps when exploring an API, and taking into account that no other studies show the opposite, the previous statement about the appropriateness of giving up the safety of static typing for exploring an API must be called factually incorrect. While evidence exists in this case, where there is none, claims should be checked to see if they hold up to scientific tests of correctness. More crucially, for authors of programming languages, the spontaneous change of programming languages features without evidence should be avoided. While we have no doubt that some will find such changes harmless exploration, we feel that non-evidence based language divergence, such as this, are of debatable utility and may impact software maintenance in the long-term. Thus, we are led to our next responsibility:

Resp.	Do not change language features without backing evidence.

3.2 Responsibilities of the Programming Language Community

"Start by doing what's necessary; then do what's possible; and suddenly you are doing the impossible."

- Francis of Assisi

While these responsibilities were directed to single scholars, researchers, or language designers, like those that invent new constructs, we think it is important to discuss the responsibilities of the programming language community. By programming language community, we mean those scholars, authors, or designers that accept new languages or advertise new languages to be used in teaching and research. In this section, we are also serving our discussion to programming language review boards at conferences, journals, and scholars at funding agencies. While we admit readily that academic scholars, let alone individual authors, can only do so much to attack a problem as complex as the language wars, it's naive to think that a community cannot make progress over long periods of time. Thus, we describe here our thoughts on what can be done, focusing on taking ownership of the problem and community outreach.

3.2.1 Impose Science on Chaos

"We impose order on the chaos of organic evolution. You exist because we allow it and you will end because we demand it."

- Sovereign, Mass Effect

Both in an industrial and educational context, little academic work is conducted or funded to evaluate the language wars.

[4] See `https://github.com/bamboo/boo/wiki/Gotchas-for-Python-Users` for a more complete description of differences between Boo and Python

Entire communities at the National Science Foundation are dedicated to educational computer science, but such programs generally do not discuss this issue, despite the fact that this can lead to peculiar decisions caused by the state-of-the-practice. For example, the excellent high school curriculum, Exploring Computer Science, includes within it a variety of programming languages. This decision, to use more than one programming language in a curriculum for relatively young children, is reflective of the discipline, for good or ill. In this section, we discuss the types of research we think the funding agencies, academic program committees, and journals in language design, should be encouraging.

First, while an outsider to the computer science community might rightfully ponder why we have not yet done it, we need significantly more competitive analysis amongst language tools. At conferences such as ECOOP, OOPSLA, POPL, etc. the languages C, C#, Java, JavaScript, Python, Haskell, Scala, OCAML, F#, Scheme and Lisp are commonly discussed. While using a particular language is always fine for scholars exploring the world around them, from our perspective, the language wars have been at least partially caused by a lack of formal tests comparing aspects of language products. While we understand some in these communities are mathematicians, or designers, decisions related to language impact still require evidence. This leads us to our first responsibility for the community:

Resp.	Investigate whether language features benefit programmers in practice.

Second, committees need to encourage more investigation of the mystery and implications of the language wars. This includes analyzing how divergence of languages is impacting other disciplines. As just one example, we think it would be good to know how language divergence is impacting chemists or physicists. Additionally, we know very little about how the divergence of language products is impacting education or the software industry generally. While we think the result of the language wars is probably negative, we would argue that no scholar really knows—computer scientists do not investigate one of the most impactful problems in all of computer science. To be clear, we have heard many scholars in our community argue, if the reader will excuse a generalization, that the free market will work it out on its own. We reject this argument, finding it naive. If the free market could solve the language wars, it probably would have already. This leads us to our next responsibility:

Resp:	Investigate the divergence of programming languages.

To give an idea of what an investigation of the language wars might look like, studies could analyze both languages as a whole and language features. A good starting point for analyzing language features would be those that are already available in most languages (e.g., loops, side-effects, functions or procedures, parameter passing, etc.) and the techniques for these studies would probably include controlled experiments. However, the previously mentioned studies by Parnin et al, Gil and Lenz, Callau et al, and Souza [5, 15, 34, 41] that study use of language features in existing code repositories may give us usage information, although determining causality is typically easier with the former for well known reasons.

Finally, language conferences and journals spend too much of their time investigating issues that, to our eyes, are of lesser importance. For example, type soundness proofs are common. We never seem to ask, however, "Does type soundness matter?" While we do not doubt the proofs, as they are self-apparent, we harbor doubts that continued publication of type soundness proofs will push forward the state of the art in a meaningful and important way, while also recognizing that they are sometimes necessary to make a point. This is true in our view, especially, because many of the most popular programming languages are not type sound. While type soundness is just one example, we should say that it is obviously true that mathematics is crucial in language design, but in many disciplines (e.g., physics) mathematics is coupled with rigorous empirical data and observations.

3.2.2 Conduct Outreach Work to Disseminate Verifiable Claims about Language Design

"Help will always be given at Hogwarts to those who ask for it."
– J.K. Rowling, Harry Potter and the Chamber of Secrets

Even if scientists work diligently on the language wars for decades in a scholarly setting, actually making an impact in practice seems unlikely unless we work with existing language designers outside the discipline. We think this dissemination work should take two forms with the community: 1) we need to convince corporate language designers of what is true and false, and 2) more work needs to be presented on how language design impacts humans or society in the broad, given that this *is a fundamental principle of programming languages*.

On the first point, take as an example our previous discussion of Boo or Java. On the one hand, new features of such languages are generally not built on a foundation of evidence that they are important. On the other, unless the research community, after it has conducted such studies, conducts outreach to teach designers why their features succeed or fail, and what tangible designs fix the problem, little will change.

From our perspective, large conference venues could help by conducting workshops or other outreach work where corporate language designers are welcomed and taught about language impact. We do not mean, in the strictest sense, that we should teach how to run experiments, as individual designers like the author of Boo probably have little mecha-

nism for running them even if they wanted to. We do mean, however, that language designers, and students in the classroom, need to be taught what the best available evidence shows in regard to language design.

Second, the research community needs far more work studying the impact of language design. As a consequence, the research community must make sure that human-centered methods are heavily applied in publications about programming languages. Presentations at conferences or workshops should be encouraged to contain elements about human-centered studies or language impact broadly and this information should be shared beyond academia. This is important, as we imagine some scholars claiming that it is enough if a new language construct is studied by using human-centered methods and that it is not necessary to conduct outreach work. However, in practice, many large corporations own the language products that are used by the community in practice—they are financially dependent. If the scholarly community were to conduct the studies, but not conduct outreach, it seems plausible corporate partners will ignore it. From both previous statements we conclude the following responsibility of the research community:

Resp.	Teach, communicate, and encourage human-centered methods in programming language design to a wide variety of audiences.

3.3 Responsibilities of the Software Industry

Now that we have discussed the responsibilities of individual language designers or scholars, and the language community, we feel it is important to note that problems as daunting as the language wars cannot, and will not, be solved by academia alone. Users of programming language technologies, including everyone from working scientists in other disciplines to video game designers at Electronic Arts, also have responsibilities in regard to this problem. We discuss these responsibilities in this section, focusing on the 1) need for more or better evidence on how languages are used in industry, and 2) industry's responsibility to demand better evidence from language designers.

3.3.1 Tell the World How you Use Your Languages

Probably one of the most crucial responsibilities of the software industry is to tell us more exactly what languages are in common use and how they are used. While it may seem obvious, garnering a picture of the real problems with the language wars can only be achieved if we have a greater understanding of it. Some may believe that we already have answers to this question, but we argue here that our answers are weaker than we need.

For example, one of the more common ways of evaluating the languages used is a resource such as the Tiobe index (as, for example, being used in [18, 32, 41]). Such an index is often used as an indicator of how languages play a role on the market, although there are serious critiques that we feel should not be overlooked. [5] In other words, we do not know enough about how languages are used in practice, which makes it difficult for our community to guide research priorities to meet the market needs. Without such information, scholars are often in the dark, leading us to our first responsibility:

Resp.	Make data regarding language use public.

3.3.2 Demand Evidence

"Don't pay the ferryman. Don't even fix a price. Don't pay the ferryman until he gets you to the other side."

- Chris de Burgh, Don't pay the ferryman

We harbor little doubt that the software industry itself may not, or cannot, realistically conduct the kind of scholarly evidence gathering that would be required to know the impact of language design in practice. Mathematical proofs are usually solved by scholars and randomized controlled trials often need to be conducted by groups independent from economic self-interest. Thus, while the software industry may not conduct studies on its own, before adopting language products, we urge the software industry to demand more evidence and to abandon language products/features that are not living up to reasonable scientific expectations.

As an example, consider the new specification for C++11, or the up and coming C++14. In this standard, the C++ ISO committee made a number of changes to a language used world-wide. Even a quick scan of the C++ 11 wikipedia page has a section labeled "Core language usability enhancements," a surprisingly specific claim from writers of the page. Given that Wikipedia is hardly a perfect source, if we look instead at the draft specification by ISO, the word "usability" is used only once, [6] and in its context of usage, they likely did not mean human usability.

What we find interesting about the changes to C++, however, is that the ISO committee appears to have made changes without evidence. It is unclear whether the standard is better or worse from a human-centric point of view. Consider, for example, the language addition of lambda functions. On the one hand, it might appear that adding this language feature is positive, given that other languages have added this feature recently, perhaps most notably Java JDK 8. However, we find no record of conducting experiments, have not found any data, and so far as we can tell, find no record of any evidence gathering at all. We have not pub-

[5] see for example `http://lambda-diode.com/programming/the-tiobe-index-is-meaningless`, `http://blog.timbunce.org/2009/05/17/tiobe-index-is-being-gamed/`

[6] The free working draft can be found at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`

lished data on this language feature, but think it is unclear whether it has much of an impact in practice. This leads us to another responsibility for the software industry:

Resp.	Groups like ISO, and companies that support them, should base their decisions on evidence.

We want to be clear in saying that while we recognize that such committees often have to rely on expert judgment, C++ has been established for decades. When it was first designed, it is understandable that Stroustrup was not conducting randomized controlled trials. Decades later, however, it is unfortunate that groups as large and important as ISO are not using scientific evidence. While some of the changes in the C++ 11 specification "feel" harmless to us (e.g., adding a hash table to the standard library), others seem confusing or of questionable value. Ultimately, it's unfortunate that the evidence standards were not higher for something as obviously important as changing C++.

Our point though, committees or features aside, is that industry has a role to play in this process. Groups like Microsoft or IBM that build C++ compilers should be asking ISO for evidence and otherwise holding its feet to the fire because the productivity of its developers depend on good design. Google showed this quite clearly in their latest article at ICSE [40], showing evidence that individual compiler errors were impacting developer productivity in the company. We argue that the C++14 committee is being pseudo-scientific if they are not using the scientific method in making the next version. Running user studies is easy enough to accomplish that we see few acceptable excuses.

3.4 Responsibilities of Educational Institutions

We think educational institutions also have an important role to play in the language wars. Academics ultimately decide which language products the next generation of computer scientists will learn and should educate students to use evidence when thinking about them. Thus, educators play a significant role in the language wars and have important responsibilities. We discuss these here.

3.4.1 Analyze Language Impact with Students

While the computer science education community has a number of important roles (e.g., engagement, work with people with disabilities, computing in primary schools, the impact of gender) studying the impact of programming language design is somewhat rare in this community. This is interesting because, while it's clear many are interested in visual tools, in our mini-review of colleges in the mid-west, we saw few universities actually using visual tools to teach college students, almost always favoring more general purpose programming languages.

First, even in regard to visual programming languages, while we know that syntax directed editors [49], or more modern tools like Alice [6, 36], Scratch [29, 38], or perhaps end user programming systems [27] have been around for some time, the evidence for their benefits is not as convincing as some might think. For example, it is crucial to recognize that visual tools may help novices initially (although not for the blind), specifically in promoting transfer-of-learning from visual systems to text-based programming languages. This has been confirmed independently and using different methodologies with at least the tool ALVIS [20] and Alice [7]. On the other hand, a study by Garlick and Cankaya compared using Alice in an introductory course to a control group that started with pseudo code, finding that the Alice group had a statistically significant drop in grades [14]; a very different kind of study with a very different outcome from the one discussed by Dann et al. [7]. This is interesting, we think, in part because the Garlick and Cankaya study was the first randomized controlled trial conducted by a group that was, to our knowledge, independent from the CMU team.

With a tool like Scratch, there appears to be little rigorous evidence gathering on use of the tool compared to others that existed before it. We find this odd given its considerable funding from the National Science Foundation for promoting engagement activities [4, 13, 29, 38, 51]. While we make no claims about Scratch either way, we were surprised to learn that most published papers we investigated contained highly limited evidence gathering, although many, despite this, contained praise for the tool. On the other hand, one study looked at 536 Scratch projects, finding students learned programming concepts using Scratch over time. This might seem acceptable until we realize that the study did not have a control group [28]. We remind the reader, as Kaptchuk describes, that the use of control groups helps rule out important issues like fraud or more mundane issues like incorrectness [22]. More crucially, given that students spend most of their academic career using more general purpose products than Alice or Scratch, it is unfortunate that so little research is conducted investigating the products people use the most in the classroom or in the field.

This leads us to our first responsibility for the educational community:

Resp.	Conduct competitive analysis of programming languages with students at all levels of the academic pipeline.

Second, we need to get a better picture of how programming languages are used in academia internationally. As it stands now, we know surprisingly little about how language products are used in practice. Work on engagement has focused on high school using Alice or Scratch, mostly, but it is unclear how much these tools are used in college or for how long. While some schools use them in an introductory course, we have little data to support what choices are even being made, let alone the consequences of those choices. This leads us to a responsibility:

Resp.	Carefully map out language usage in academia.

From our perspective, we need such a map to understand the language wars. We think it would provide at least the following: 1) a more accurate estimate of usage in academia and K-12 schools, and 2) by identifying schools and universities that use particular language products, we can then work together to conduct controlled trials on programming language learnability as best as we can. For example, it would be good to know whether schools that begin with visual programming languages have higher performing students, on average, than those that do not, by the end of their degree program. Such studies might help us determine whether the positive impact documented by Dann et al. [7] or the negative impact documented by Garlick and Cankaya [14] has implications beyond the first course in college.

3.4.2 Teach Empirical Methods to Programming Language Students

The current generation of computer science students in academia rarely, if ever, is engaged with evidence on language impact, with the exception that some universities may provide a human computer interaction course. While we are not calling for more HCI, current courses on programming languages typically follow the approach where students try many languages to give them a rough overview or introduction to what is out there. Other programming language courses teach formal characteristics of certain programming language constructs.

From our perspective, this model of teaching programming languages is unfortunate for a world where the number of languages and libraries is so daunting. We call for a re-thinking of our approach to teaching programming languages to students—one which acknowledges that the impact of language design on people is a fundamental principle of programming languages.

Specifically, we propose that programming language courses should be more stringent on teaching what actually works and should dispel the myth that language does not matter for productivity. Put another way, scholars teaching language courses should inform students that static type systems improve developer productivity under a wide variety of conditions (e.g., with or without an IDE, without documentation, with documentation), thread based parallelism systems lead to more bugs than software transactional memory, and that syntax can cause an order of magnitude difference in accuracy amongst students, results for which we have already cited randomized controlled trials as evidence. Authors of programming language textbooks, then, should try to give students a well-rounded view of the evidence, so that the next generation of computer scientists makes language decisions on an increasingly larger evidence base.

Resp.	Teach students about the language wars, providing evidence based arguments on the debate.

4. Summary and Discussion

"Throughout history, every mystery, ever solved, has turned out to be ... not magic."

- Tim Minchin, Storm

This essay is a first attempt to frame the debate about what is often colloquially termed the programming language wars. This struggle was natural for the computer science community in its inception, when we had no experience, but this is no longer true. At the beginning of the 21^{st} century, we already know how to do a great deal in regard to designing languages and we need to use this as an objective filter of what already exists. Further, we feel that we are duplicating effort at too large a scale, which needs to be carefully investigated and considered.

Perhaps the recent extensions of C++ are a good exemplar for what we feel has gone wrong. In the case of C++, new constructs continue to be invented. While few would doubt that innovation can be positive, these constructs have no evidence foundation. Even in academia, scholars are expressing less skepticism than we feel they should toward anecdotes and claims, while simultaneously showing too much skepticism toward experimentation—an ironic truth, given that techniques like randomized controlled trials are the gold standard in almost all other scientific disciplines. With that said, we imagine our email boxes saying, "But yes, this is how it's done." Our argument is that the way it is done makes no sense. We are scientists, not bricklayers.

The lack of a reliable evidence foundation is at the core of the language wars and we may have to make changes to our discipline to fix the problem. This may include alterations to peer review policies, adjustments to education, and an increasing unwillingness to accept claims made by language designers that do not base their design decisions on verifiable and replicable scientific data. This is important because the language wars may be part of the most massive duplication of effort in history. As such, it is poor stewardship if we continue to ignore the problem in perpetuity.

We imagine a large number of participants in the language wars, no matter what we have said in this essay nor how we have said it, will object using anecdotes or personal experience. We imagine researchers will tell us they cannot provide evidence for all their steps or that they find evidence gathering banal or control groups to be unimportant. We imagine other scholars will tell us that we should not worry—empirical evidence is just an unhealthy trend in language design research and it will go away soon. Given the history of science, which trends toward increasingly reliable empirical measurements, we have our doubts.

It seems plausible that arguments from the language community will be similar. Developers of language products that are not scholars, have no training in study design, nor any mechanism to conduct experiments, will rightfully say they have no way to follow our suggestions. Of course, this is justifiable, but developers that cannot contribute to evidence gathering should at least educate themselves on what the evidence shows. In other words, if a language designer cannot gather evidence on syntax, they should follow existing data. If they have no training in studies on type systems, they should again collaborate or follow the evidence.

The software industry may also object. Perhaps industry will argue that they cannot wait for evidence because they require quick solutions to their problems. Language owners might tell us that studies will not save money and therefore they do not care. Both of these arguments are naive. What we see in industry currently is that development houses use a polyglot approach to software development. It is perfectly normal for a professional to make a claim like, "We use PHP on the backend, with a Javascript front end, generating HTML and CSS3, using SQL for our database and a JSON intermediate layer." Further, companies like Microsoft, Google, IBM, or Oracle seem to have no problem reinventing their own standard libraries. We have no evidence, but would be hard pressed to believe that this is inexpensive or that these companies do not care about the cost. An evidence-based language industry could, if we play the long-game, make such efforts increasingly less necessary.

We imagine educational institutions will tell us that the request for evidence is counterproductive and that conducting research studies in the classroom is too difficult. We also suspect many educational researchers, and rightfully so, imagine that even if they conducted such studies, language designers would probably ignore the evidence anyway (a claim the authors have heard many times). Further, while we recognize that in classroom studies can be difficult (see e.g., Enbody and Punch [10]), the medical sciences have it dramatically harder than us—if our studies fail, no one dies. Yet, while obviously imperfect for many reasons, other disciplines make tangible progress over time using empirical data, a fact our community should not ignore.

Finally, we are all too aware that most of our suggestions will not be taken seriously by many scholars in our community. Many will tell us that they do not need to use evidence to get tenure and promotion—and they are probably right, given that such procedures are sometimes based on counting papers instead of reading them. We have even heard, unfortunately, some younger scholars make arguments like, "I am sympathetic that we need to use evidence, but if I do that, I will not be able to publish my papers, because POPL will never take it seriously." That any scholar in our history has thought such a claim might be true is unfortunate. If the claim is actually true, it is a travesty. We imagine some of our suggestions could take years of introspection by the computer science community, while others may never be fulfilled. We implore the community, however, to take action. Potential solutions to the language wars are not magic.

5. Conclusion

"I am and always will be the optimist, the hoper of far-flung hopes, and the dreamer of improbable dreams."

- Doctor Who, Eleventh Doctor, Season 6, Episode 6

As we write the concluding portion of this essay, harboring no misconceived delusions that it will impact the world in any substantive way, we do add one final responsibility, which we say with all sincerity and complete seriousness:

Resp.	We need to think more deeply about the programming language wars, before we leave a mess for the next generation to clean up.

The programming language wars is a major social ill causing serious problems in our discipline. Authors of these languages, we believe, genuinely want to improve how the world computes, but our arrogance, ignorance, and unwillingness to compromise on some of the most basic issues of our day has left our community in significant disarray. We duplicate effort massively. We reinvent the wheel constantly. We leave scientists in other disciplines in bewilderment at our evidence practices and standards. We need to do something about the programming language wars. The last few hundred years have seen tremendous innovation, largely due to empirical observations of nature. For problems like the language wars, which will not be solved by yet one more proof or the magic of the free market, we need to join the scientific community at large, challenging ourselves to not base our decisions on faith, hope, and love; acting on what in our view is the elephant in the room.

Acknowledgments

We would like to thank Bruce Horn for his feedback on drafts of this work, in addition to all anonymous reviewers. The detailed comments we received were extraordinarily helpful in revising and improving this essay.

References

[1] A. P. Association. *Diagnostic and Statistical Manual of Mental Disorders DSM-V-TR*. American Psychiatric Publishing, Arlington, VA, fifth edition, 2013.

[2] M. Bartsch and R. Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control*, 16(1):23–44, 2008.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver,*

British Columbia, Canada, October 18-22, 1998, pages 183–200. ACM, 1998.

[4] K. Brennan and M. Resnick. Stories from the scratch community: Connecting with ideas, interests, and people. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 463–464, New York, NY, USA, 2013. ACM.

[5] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6):1156–1194, 2013.

[6] S. Cooper. The design of alice. *Trans. Comput. Educ.*, 10(4):15:1–15:16, Nov. 2010.

[7] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 141–146, New York, NY, USA, 2012. ACM.

[8] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[9] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 208–212, New York, NY, USA, 2011. ACM.

[10] R. J. Enbody and W. F. Punch. Performance of python cs1 students in mid-level non-python cs courses. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 520–523, New York, NY, USA, 2010. ACM.

[11] S. Endrikat and S. Hanenberg. Is aspect-oriented programming a rewarding investment into future code changes? A socio-technical study on development and maintenance time. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 51–60, Kingston, CA, 2011. IEEE Computer Society.

[12] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do api documentation and static typing affect api usability? In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 632–642. ACM, 2014.

[13] L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick. Designing scratchjr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*, IDC '13, pages 1–10, New York, NY, USA, 2013. ACM.

[14] R. Garlick and E. C. Cankaya. Using Alice in CS1: A quantitative experiment. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 165–168, New York, NY, 2010. ACM.

[15] J. Gil and K. Lenz. The use of overloading in java programs. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 529–551, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] 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, New York, NY, 2010. ACM.

[17] S. Hanenberg, S. Kleinschmager, and M. Josupeit-Walter. Does aspect-oriented programming increase the development speed for crosscutting code? an empirical study. In *Proceedings of Empirical Software Engineering and Measurement (ESEM) 2009*, pages 156–167, 2009.

[18] M. Hills, P. Klint, and J. Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335, New York, NY, USA, 2013. ACM.

[19] M. Hoppe and S. Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.

[20] C. D. Hundhausen, S. F. Farley, and J. L. Brown. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Transactions on Computer Human Interaction*, 16(3):1–40, 2009.

[21] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*, pages 132–146. ACM, 1999.

[22] T. J. Kaptchuk. Intentional ignorance: A history of blind assessment and placebo controls in medicine. *Bulletin of the History of Medicine*, 72(3):389–433, 1998.

[23] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[25] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *IEEE 20th International Conference on Program Comprehension, Passau, Germany, June 11-13, 2012*, ICPC'12, pages 153–162. IEEE Computer Society, 2012.

[26] A. Ko, T. LaToza, and M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, pages 1–32, 2013.

[27] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.

[28] J. Maloney, M. Resnick, N. Rusk, K. A. Peppler, and Y. B. Kafai. Media designs with scratch: What urban youth can learn about programming in a computer clubhouse. In *Proceedings of the 8th International Conference on International Conference for the Learning Sciences - Volume 3*, ICLS'08, pages 81–82. International Society of the Learning Sciences, 2008.

[29] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, Nov. 2010.

[30] 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*, PLATEAU '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM.

[31] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tucson, AZ, USA, October 21-25, 2012*, OOPSLA'12, pages 683–702. ACM, 2012.

[32] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18, Oct. 2013.

[33] M. Naftalin and P. Wadler. *Java generics and collections - speed up the Java development process*. O'Reilly, 2006.

[34] C. Parnin, C. Bird, and E. R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 3–12. IEEE, 2011.

[35] C. Parnin, C. Bird, and E. R. Murphy-Hill. Adoption and use of java generics. *Empirical Software Engineering*, 18(6):1047–1089, 2013.

[36] R. Pausch. Alice: A dying man's passion. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 1–1, New York, NY, 2008. ACM.

[37] P. Petersen, S. Hanenberg, and R. Robbes. An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 212–222. ACM, 2014.

[38] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[39] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? *SIGPLAN Not.*, 45(5):47–56, Jan. 2010.

[40] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 724–734, New York, NY, USA, 2014. ACM.

[41] C. Souza and E. Figueiredo. How do programmers use optional typing?: An empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 109–120, New York, NY, USA, 2014. ACM.

[42] S. Spiza and S. Hanenberg. Type names without static type checking already improve the usability of apis (as long as the type names are correct): an empirical study. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 99–108. ACM, 2014.

[43] A. Stefik, S. Hanenberg, M. McKenney, A. A. Andrews, S. K. Yellanki, and S. Siebert. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In *Proceedings of the 2014 IEEE 20th International Conference on Program Comprehension*, ICPC '14, pages 223–231. IEEE Computer Society, 2014.

[44] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, Nov. 2013.

[45] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, 2006. ACM Press.

[46] M. Steinberg. *What is the impact of static type systems on maintenance tasks? An empirical study of differences in debugging time using statically and dynamically typed languages*. Master Thesis, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen, January, 2011.

[47] M. Steinberg and S. Hanenberg. What is the impact of static type systems on debugging type errors and semantic errors? An empirical study of differences in debugging time using statically and dynamically typed languages. unpublished.

[48] A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, Portland, Oregon, USA, 2011. ACM.

[49] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sept. 1981.

[50] W. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[51] I. Utting, S. Cooper, M. Kölling, J. Maloney, and M. Resnick. Alice, greenfoot, and scratch – a discussion. *Trans. Comput. Educ.*, 10(4):17:1–17:11, Nov. 2010.