Staking Claims: A History of Programming Language Design Claims and Evidence

A Positional Work in Progress

Shane Markstrum
Bucknell University
shane.markstrum@bucknell.edu

Abstract

While still a relatively young field, computer science has a vast body of knowledge in the domain of programming languages. When a new language is introduced, its designers make claims which distinguish their language from previous languages. However, it often feels like language designers do not feel a pressing need to back these claims with evidence beyond personal anecdotes. Peer reviewers are likely to agree.

In this paper, we present preliminary work which revisits the history of such claims by examining a number of language design papers which span the history of programming language development. We focus on the issue of *claim-evidence correspondence*, or determining how often claims are or are not backed by evidence. These preliminary results confirm that unsupported claims have been around since the inception of higher level programming in the 1950s. We stake a position that this behavior is unacceptable for the health of the research community. We should be more aware of valiant and effective efforts for supplying evidence to support language design claims.

1. Introduction

A glance at the Wikipedia page that boasts of listing all the major programming languages shows that there are thousands of programming languages and language variants in existence [1]. This list is not stagnant, but continues to grow as members of the software development community get the urge to fix the issues they have with their favorite languages. In other words, since developers can always find some slight flaw in any existing language, we are bound to witness the creation of new languages until the end of time.

With such a plethora of languages to choose from, each new language needs to stake some claim to justify its existence. These claims tend to fall into three categories: novel features, incremental improvement on existing features, and desirable language properties. These categories are easily distinguishable. A novelty claim states that the language provides a feature that has never been seen before. An incremental improvement claim states that, while the feature is not novel, the designers have found a way to improve its effectiveness. The third claims are the broadest, encompassing properties such as naturalness, readability, practicality, and efficiency.

We expect, as a rational audience, that such claims will be proven over the course of some introductory paper or papers. However, we often take such claims at face value and never demand that the authors provide the evidence or arguments that would truly prove such claims. The justification behind the programming language community's indifference to this missing *claim-evidence correspondence* is that it is difficult to prove these claims, especially in the category of desirable language properties, and therefore inferred evidence is adequate. As a result, language designers are free to make claims that they have no intention of proving true in the hopes that their peer reviewers are sympathetic and will make the logical leap-of-faith that the claims are valid.

In this paper, we begin exploring the historical claims of language designers in relation to their claim-evidence correspondence. This trending data is absent from the research literature. We believe this is part of the reason why language designers do not make more of an effort to actually evaluate their language with provable metrics. Such metrics should go beyond simple, and ambiguous, measurements such as lines of code for a program and appeals to the readers' intuitions on desirable language properties.

While our goal is to provide a comprehensive picture of historical claim-evidence correlations, we are still only at the beginning stages of this work. As a result, this work-in-progress paper focuses on just a few papers that span the era of modern programming, with an emphasis on early language papers.

The next two sections present findings and excerpts from classic and modern language design papers[1]. These papers have been selected to cover a variety of recognizable languages from a number of sources, such as journal articles, conference articles, technical reports, and white papers. The fourth section discusses trends found in these documents and their implications on how we can improve claim-evidence correlations. We argue in this section that the level of claim-evidence correspondence has historically been poor and that the community as a whole should take a more active role in encouraging higher levels of correspondence. We conclude with a brief summary and discussion of future work.

2. Claim-Evidence Correspondence in the Golden Age

To fully understand the state of claim-evidence correspondence, we need to start by looking at the claims made by early language design papers, when there were only a few languages against which designers could compare their work. Early papers on languages such as Fortran, ALGOL, Lisp, COBOL, PL/I, and APL fall into this category. As our research is focused on third-generation or higher level languages, machine and assembly languages are explicitly not considered.

Nearly all of this work makes claim to novel features, and given that very few programming languages existed prior to 1960, these claims are almost unarguably valid. What remains to be gleaned from this early work is how the authors handled claims of desirable language properties, or if they really needed to make such claims at all. Higher level programming languages and their associated compilers were certainly created to remove the burden of dealing with particular architectures in the software process. But would the desirable properties claims be rigorously argued or would they be cursorily addressed via proof-of-concept?

There is a large body of published work on these early languages, but we wanted to find some introductory papers with canonical status. As a result, we chose to start with Backus' first paper on Fortran [2], two papers representing the ACM-GAMM committee's work on ALGOL [3, 10], and McCarthy's first paper on Lisp [8].

The earliest of these papers presented the IBM 701 Speedcoding system [2]. The claims in the paper focus on the practical aspects that the higher level language provides over machine code. As there were relatively few such languages in existence at the time, these claims primarily fall under novelty as the paper itself demonstrated a proof-of-concept that higher level languages were possible. In a modern paper, such claims would invariably be considered desirable language properties. The evidence for the claims are largely anecdotal, although they do go beyond the authors' experience. For example, to support their claim of ease of use, the authors describe an experience involving a customer who had no previous knowledge of the Speedcoding system but nonetheless created a correct program using only the manual. In comparison to standard assembly language programs of the time, this was considered a big achievement. More typical is the following claim.

> This feature often enables one to reduce the number of instructions in a loop by a factor of 1/2.

The key word in this claim is "often." However, there is no explicit evidence to indicate the frequency with which the instructions were reduced. Given that this is one of the earliest papers on higher level languages, it is discouraging to find this kind of hand waving in an attempt to boost the significance of the achievement.

The two ALGOL papers are more descriptive of the language than the Fortran paper, and make fewer overall claims as a result. Beyond the novelty of the syntax structure, the first preliminary report makes four claims about naturalness and readability [10]. None of these claims are substantiated by any evidence, although they should be considered results of the ACM-GAMM committee's consensus. The report starts with the ultimate of these claims.

> Even so, it provides a natural and simple medium for the expression of a large class of algorithms.

This claim is predicated on a universal understanding of what natural and simple mean–in this case, it presumes a strong mathematical or algorithmic background and agreement that Fortran (at that time) did not possess either of these traits. It provides no argument in support of the claim because agreement was assumed to be implicit: everyone who worked on the committee agreed that these properties were achieved. Later in the same document, the objectives for the creation of ALGOL are stated. One of them is that the language should be "readable with little further explanation." Since the words natural and simple only occur in the original quote, it appears that the authors intended for the adjective readable to be in harmony with these other two descriptors. However, it is possible for a language to be both natural and unreadable by a large audience (e.g., any written natural language).

The follow up paper to this preliminary report [3] provides only one new claim.

> The proposed language is intended to provide convenient and concise means for expressing virtually all procedures of numerical computation. . .

The syntax that is presented throughout the rest of the paper is an attempt to justify this claim. However, without explicit comparison to programs in other languages, the presentation of the syntax can only be considered an appeal to the reader for tacit approval. The term "concise" here is especially problematic as it is not clear whether the author means that

[1] The application of the classic and modern adjectives to describe programming languages may be subject to reader's opinion

the syntax of the language is concise; the programs created by the language are concise; or both.

The fourth paper we examined from this era is John McCarthy's classic introduction to LISP [8]. The paper makes claims of novelty in regards to an early denotational semantics for functional programming with recursion. These claims are justified by consensus agreement in the programming language community. However, there are also two other claims which fall into the desirable language properties category that are presented without evidence. The first is demoted to an opinion through the phrase "[w]e believe." Hedging the claim in this way allows for a discourse in the community on whether such a claim is true even when evidence is not provided by the authors. In our opinion, this is an appropriate way for authors to introduce unsubstantiated claims to their readers. However, the second major claim of the paper is the following.

> This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular.

While we will ignore the claims of writability in this paper[2], there is no proof beyond personal opinion that the notation introduced here consisting of nested matched parentheses is particularly readable. A standard response to such evidence is just more personal opinion which claims the opposite. For example, anecdotal evidence gathered by the authors of this position paper suggests that code which heavily utilizes nested balanced parentheses (and other matched delimiters) is less readable to some programmers. A user study which looked at cognitive effects of reading standard ALGOL or Fortran syntax versus this functional style would have been useful evidence for such a claim.

3. Claim-Evidence Correspondence in the Modern Age

In contrast with the classic languages, modern languages need to quickly distinguish themselves from other languages. As a result, the language design papers do not have the same luxury of letting the syntax and program examples speak for themselves. These modern papers tend to have more claims than classic papers and seem to rely even more on a sympathetic reader. There are many possible reasons for this, which we will discuss in the next section.

Since the number of currently used languages is so large, we chose to focus on a couple of widely used languages (C++ and Java), one successful modern research language (Scala), and one brand new language (Ur). Finding an adequate introductory paper for C++ proved to be a challenge, as we wanted to avoid using documentation which shipped with a language distribution or a book-length tome. As a result, we chose to use the press release introducing the world to C++ from the Bell Lab News newsletter [7]. For Java and Scala, we chose to examine the overview technical reports provided by the developers [6, 9]. The Ur paper is from the PLDI 2010 proceedings [4].

The newsletter announcement of C++ [7] is, unsurprisingly, mostly a series of claims intended to pique the interest of the software community. These claims fall into all three categories mentioned earlier in this paper. Novelty is claimed twice: for developing an efficient compiler of class-based code, and for mixing C-style memory manipulation with object-oriented programming. As C already had some forms of data abstraction via structs and unions, the integration of data abstraction with the new class-based syntax can be considered an incremental improvement. All other claims fall into the desirable language properties category. Given the short nature of the news blurb, they are mostly unsubstantiated. Evidence is provided for practicality in the form of a list of groups already using C++. As with all press release claims, this is merely a smoke-and-mirrors psychological trick. There is no mention of whether these groups actually found C++ an improvement over other languages or why they initially started using the language. Without even anecdotal evidence, the reader can only infer that the groups must like C++, thereby predisposing the reader to focus on the positive aspects of the language.

The Java Language Environment white paper [6] provides a very comprehensive overview of the features of Java and also acts as its own press release. There are a panoply of claims which feature strongly hyperbolic language throughout the document. Some of the claims are genuine. Java can certainly lay claim to some novel features, such as built-in multi-platform threading primitives. The designers also knew that they were primarily assembling a greatest hits of language features and do not shy away from acknowledging that many of the technologies are simply incremental improvements. But the paper is set on selling Java and is littered with descriptors which have no evidential backing such as "simple," "familiar," "small," "robust," and "faster."

Some of these claimed properties are measurable. Familiarity can be measured through surveys of users who are just starting to use Java and have been exposed to previous languages. Speed of development can be measured by tracking the time to complete projects in a controlled user study[3].

Most of the terms employed by the Java designers, such as simple and small, must be put into a context in which they can be objectively measured. For example, compared to the syntax of the untyped lambda calculus, there is no practical Java grammar which can be considered simple. However, certain features particular to C and C++ were simplified or eliminated in the design of Java. At the same time, new features were added. The paper makes no effort to show that the new features do not overly complicate the previous

[2] Perhaps the authors intended for writeability to mean easy to write, but this is not mirrored in the standard definition of readable.

[3] Later studies such as [11] did attempt to quantify this, although such studies are notoriously difficult to implement correctly.

simplifications. Given the current Java specification, it seems unlikely that anyone would claim Java is a simple language at this point[4].

The Scala overview [9] is scaled back from the Java overview and focuses primarily on providing code examples to familiarize readers with its uncommon syntactic style. The Scala team was primarily interested in developing a "scalable" language and its claims revolve around this idea. The novelty claim of the paper is that it implements the νObj type system, which provides path-dependent types, and thus supports a unique combination of mixin-composition, family-polymorphism, and first-class functions. As with the other languages, the designers begin to appeal to the reader when they make claims about desirable language properties. For example, when discussing flexibility provided by variance annotations on types, they state the following.

> We found that in practice it was quite difficult to achieve consistency of usage-site type annotations, so that type errors were not uncommon.

The justification to remove this feature leaves a lot unspecified, such as who found it difficult in practice, to what problems the usage-site type annotations were applied, and whether the type errors provided insight to the developers beyond being an annoyance. The most bold of these claims is that Scala's approach to mixin-composition enables "smooth incremental software evolution." While this is certainly one of the design goals of the language creators, without either a user study or a corpus analysis, such a claim cannot be proven. A composable system is not guaranteed to make development smooth if the programmers find the language itself difficult, for example.

Spurred on by a boom in cloud computing and web services, a large number of languages with XML and database integration have recently surfaced. As such a hot field of work, it seemed fitting to choose a language targeting this area to include in this early analysis. The Ur language [4] ambitiously attempts to introduce dependent types into a metaprogramming language for web page generation. A large portion of the paper is spent describing in detail the types of useful functions and the semantics of the language. As a kind of turnabout to the standard language introduction, such types are rarely seen by the developers as the language designers believe the types are too complicated for the normal developer. However, the designers consistently provide no support to their language property claims beyond their own experience. For example, the following quote implies that the designers' experience using their language supersedes any other developer.

> Our experience writing programs in Ur/Web suggests that the feature set we have chosen is more than sufficient for our application domain.

Using personal anecdotal evidence to justify a general claim is not sufficient to prove that claim. The authors also mistakenly attribute a self-produced case study as evidence of general practicality. There are also unsubstantiated claims of user-friendliness and readability, although readability is noted as being a subjective claim.

4. Discussion

The results of this analysis turn up very few surprises. Claims of novelty and incremental improvement are generally agreed upon by the research community. Thus, presentation of a set of ideas, or a formal syntax and semantics, is enough evidence to justify these claims given they have been peer-reviewed. Claims of desirable language properties, on the other hand, are rarely rigorously proven, even when obtaining evidence is possible.

While we have only analyzed a handful of papers at this point, it seems likely that findings will be similar for most language design papers. It is easier for a language designer to throw together some convincing looking examples and skirt the effort involved in administering a user study or a long-term development study. The most consistent measured metric in this category was a comparison of the lines of code for selected examples. However, it is not clear what properties of language this metric actually implies as it is used to indicate a wide variety of properties including practicality, simplicity, familiarity, readability, and scalability.

One aspect of the studied papers was consistent. The designers believe that their own opinions weigh as much as, if not more than, any user study. When discussing their work, designers are likely to ascribe their opinions to the reader in the belief that the readers will respond sympathetically. There are both positive and negative aspects to this design hubris. On the positive side, novelty derives from independent thinkers, and a strong impression of ones own work is necessary to see an initial thought through to a finished product. On the negative side, this discourages any truly rational discourse with designers on what aspects or features of a language actually lead to desired language properties. For example, no individual or group would seek to publish a language design that did not feel natural in their opinion[5]. If every language is natural, though, then the "natural" descriptor has no objective use when comparing languages. This seems counter to the objective goals we seek as computer scientists and software engineers.

While the egoistic aspect of language development accounts for some of the increased dependence on reader sympathy, we do not believe that it is the sole reason. As the lan-

[4] Ironically, the Java developers quote a pessimistic Robert Heinlein passage to indicate the form of bloated technology evolution they will avoid but at this point the quote is a pretty fitting description of Java's development history.

[5] The exception to this rule is when the language was specifically designed to be unnatural, but this is a rare exception.

guage design and software development communities have grown larger, they have fragmented into factions which advocate that certain language properties are more valuable than others. Papers that describe new languages are submitted primarily to the conferences whose members are predisposed to agree with the authors' claims. As a result, extra effort to show that the languages actually have the properties would largely be ignored.

We think that this continued lack of claim-evidence correspondence is a poor comment on the state of language development. Has there really been so little movement on developing objective language measurements that our assessment techniques are essentially the same as when higher level languages existed? How can we know that new languages are actually moving us in a positive developmental direction when there are no useful objective metrics for programming language properties? Researchers are still making progress systematically answering the open questions of computability, but are ignoring rigor when they transfer their personal opinion and experience onto the general audience. This is akin to a car company that focuses on constantly improving the gas mileage of their vehicle while ignoring the ergonomics by claiming that humans can still fit inside.

We believe that it is time for language designers to own up to their claims and either hedge them as opinions or set out to prove that the claims are true. We, as a research community, should commend fellow language designers when they attempt to validate their claims even when the claims turn out to be false. For example, John Backus prefaced his early Fortran paper [2] with a statement that code written using the Speedcoding system probably will not be faster than hand-tuned machine code, but overall time is saved since development time is reduced dramatically. This claim would be much stronger if it was backed up with a user study indicating these claims were true, even if the user study itself was flawed.

This position is not meant to imply the extreme approach of only publishing language design papers that are statistics-driven. User studies, corpus analysis, and related statistical tools are heavyweight and should be reserved for measuring specific design objectives of the language creators. However, even a reasoned argument that clearly delineated the context in which the claims are made would be an improvement over the purely anecdotal claims which have been made since the earliest higher-level language papers. A careful definition of the terms used to describe a language–including the usual suspects of natural, readable, and simple–would help reviewers and programmers understand the level of success achieved by the language designers and prevent those same readers from projecting their own opinions onto these overused terms. Language designers should not bank the acceptance of their papers on finding reviewers predisposed to interpreting language descriptors in the same way the authors interpret them.

You may argue that since the scope of this paper is limited to discussion of claim-evidence correspondence, we have missed the point of the papers. Language design papers are largely devoted to explaining the syntax and semantics of the language or on walking the reader through case study examples. Such content is as much of an advertisement for the language as any authorial claim in the same way that a car is an advertisement for its brand when seen on the street. However, this content is definitional; the language would not exist if it did not have these features. Beyond serving as evidence for novelty or incremental improvement claims, any positive effect that it would have on the reader is better measured in sociological or psychological contexts.

5. Conclusion

In this paper, we have analyzed a number of language design papers in the context of finding a correspondence between designer claims and the evidence to prove the claims. This is a preliminary step in studying the history of such claim-evidence correspondence. Results were largely as we expected them to be: early language design papers do not make many claims to distinguish from existing work, while modern papers often make overstated claims in the hopes that readers will be sympathetic. In all of these papers, claims of novelty and incremental improvement are served well by case studies and comparison with the existing research literature. However, claims regarding desirable language properties are given little context and are mostly unproven.

It is our hope that by continuing this work, we will gain a more accurate picture of claim-evidence correspondence and be able to provide a comprehensive picture of the way in which language designers present their work. In the process, we hope to uncover a number of interesting and useful ways that language designers have validated their claims. We hope to distill these findings into a more codified set of language design metrics for objective comparison of languages. While we do not believe that such metrics should replace the role of the subjective in choosing a language, we do believe that they would be beneficial to the programming language and software development communities on the whole.

References

[1] Wikipedia - list of programming languages. `http://en.wikipedia.org/wiki/List_of_programming_languages`.

[2] J. W. Backus. The IBM 701 speedcoding system. *J. ACM*, 1 (1):4–6, 1954.

[3] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, pages 125–131, 1959.

[4] A. J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 122–133. ACM, 2010. ISBN 978-1-4503-0019-3.

[5] J. Favreau. Elf, 2003. New Line Cinema.

[6] J. Gosling and H. McGilton. The Java Language Environment. `http://java.sun.com/docs/white/langenv/`, 1996.

[7] T. Griggs. New C++ language extends c programming capabilities. *Bell Lab News*, 24(51), 1984.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4): 184–195, 1960.

[9] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[10] A. J. Perlis and K. Samelson. Preliminary report-international algebraic language. *Commun. ACM*, 1(12):8–22, 1958.

[11] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.