

A Rhetorical Framework for Programming Language Evaluation

Stefan K. Muller
smuller@cs.cmu.edu
Computer Science Department
Carnegie Mellon University
USA

Hannah Ringler
hringler@andrew.cmu.edu
Department of English
Carnegie Mellon University
USA

Abstract

Programming languages researchers make a variety of different kinds of claims about the design of languages and related tools and calculi. Each type of claim requires different kinds of reasons and evidence to justify. Claims regarding the aesthetics or elegance of a design, or its effects on people, are especially tricky to justify because they are less strictly defined and thus are subject to change depending on the exact audience. In this essay, we take an interdisciplinary approach to this problem by drawing on the fields of argument theory and rhetorical analysis to develop a framework for justifying audience-dependent claims. In particular, we argue that researchers should provide descriptions of specific features of their systems that connect to effects on audience in order to justify these claims. To demonstrate this framework, we show several examples of how this is already being practiced in some programming languages research, and conclude by calling for authors to provide descriptive evidence to bolster such claims and to frame and strengthen other evaluation methods such as user studies.

CCS Concepts: • Software and its engineering → General programming languages; • Social and professional topics → History of programming languages.

Keywords: programming language evaluation, justifying claims, rhetoric

ACM Reference Format:

Stefan K. Muller and Hannah Ringler. 2020. A Rhetorical Framework for Programming Language Evaluation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '20)*, November 18–20, 2020, Virtual, USA. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3426428.3426927>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

Onward! '20, November 18–20, 2020, Virtual, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8178-9/20/11.

<https://doi.org/10.1145/3426428.3426927>

1 Introduction

Programming language research makes a variety of different types of claims about languages, tools, techniques and models, which prompt a variety of different types of justifications. For example, research might claim that a debugging algorithm “scales much more gracefully” than past work [14], that a static checker has a “simpler annotation language” than others [8], or that a memory model tries to achieve “sufficient ease of use” [15]. The justifications of these claims take many different forms, depending on the type of claim: for example, a claim about type safety would be justified by a proof of progress and preservation, while a claim about scalability would be justified by evaluating it on inputs of many different sizes.

Some claims, however, do not have explicit justifications, which might not appear problematic until those claims are presented to different audiences. For example, at the 1978 History of Programming Languages conference (HOPL), John Backus called the source language of the the Laning and Zierler compiler “elegant by comparison with the earlier ones” [20]. But this is an algebraic language with no looping constructs, which most programming language researchers in 2020 would not assess as being elegant. As another example, Walter Isaacson (an author of books on technology for the general public) described the lambda calculus as “a different and less beautiful solution [relative to the Turing machine] to the *Entscheidungsproblem*” [11], a statement which would likely strike most programming language researchers as off base.

Claims of this type about work being “elegant,” “usable,” “simpler,” and so forth are evaluative claims, referring to properties that researchers believe their work has. Though there may be some level of implicit agreement among the programming languages community as to what is elegant or not, this is a subjective evaluation in that it does not have a formal definition and can and does vary across time and audience. Even seemingly objective claims, such as “type-safe” and “efficient”, which have generally well-accepted justifications, vary across time and culture to some degree, as illustrated by then-Captain Grace Murray Hopper’s keynote address, also from HOPL in 1978:

In the early years of programming languages, the most frequent phrase we heard was that

the only way to program a computer was in octal. Of course a few years later a few people admitted that maybe you could use assembly language. But the entire establishment was firmly convinced that the only way to write an efficient program was in octal. [20]

But how and why do these shifts in meaning occur? How are we to understand why Backus’s claim about elegance made sense in 1978 or the establishment’s claim of inefficiency made sense in the early days of programming languages, but both strike modern programming language researchers (and, in the case of Hopper’s anecdote, programmers in 1978) as incorrect? And how are we to guard against this kind of shifting in meaning confusing different or future audiences when they read publications from today’s programming languages research?

One idea might be to shy away from making subjective claims: for example, instead of describing the lambda calculus as “more beautiful than a Turing Machine”, say it “has fewer linguistic constructs”. But this approach doesn’t account for the fact that most claims have some degree of subjectivity or contingency to them depending on context. Further, even claims that appear more explicitly subjective can be useful and informative as long as they are well-justified: it may be that the best reason to prefer the untyped lambda calculus over the Turing Machine is its beauty.

Instead, in this essay, we aim to provide programming languages researchers with a framework to make subjective claims more convincing, accessible and stable by considering the kinds of evidence appropriate to justify different types of claims. We do so by drawing on research in the humanities, which have long embraced non-empirical and human-centric claims in their scholarship. In particular, drawing on the fields of argument theory and rhetorical analysis, we argue that one can claim that a language produces certain effects through detailed descriptions of specific features of the language or design, possibly bolstered by additional evidence to support the link between the features and claims.

In Section 2, we lay out a categorization of claims and the types of evidence currently used to justify them. In Section 3, we introduce argument theory and rhetorical analysis and explore how they can provide a framework for justifying more subjective types of claims. In Section 4, we develop this into a framework more specific to claims about programming languages, by way of several examples of how these types of justifications and evidence are already being used in programming languages research. Finally, we conclude by calling for increased justification of these claims in future research papers.

2 Types of Claims and Evidence

Before discussing how to justify evaluative claims made in programming languages research, it is necessary to describe

the sorts of claims we are discussing. Because there are many such claims, and each requires different sorts of evidence or justification, it is useful to categorize and describe a number of the sorts of claims we are considering in this work, although this essay in no way seeks to be an exhaustive or definitive list of such claims.

Coblenz et al. [7] identify three broad types of claims, or what they call “quality attributes”:

1. *formal properties* such as type soundness or other correctness guarantees;
2. *observational properties* such as performance, which relate to the compilation and execution of programs written in the language; and
3. *effects on programmers* which cover attributes related to the user experience of programmers using a language (such as “usability”, “ease of reasoning” and “modifiability”).

This listing and categorization was developed primarily in relation to the design of programming languages. In this essay, we consider a broader set of programming languages publications including analysis algorithms and tools, metalanguages, calculi, and so on. As such, we must slightly broaden the three categories above, though they remain largely valuable (for example, an analysis tool might include “manual annotation burden” as a category 3 claim above, so categorized because this is a human-focused effect on the user of the tool). We also add a fourth category, which is more relevant in papers concerning calculi or other formalisms not meant for immediate use by programmers or end users:

4. *aesthetic properties* include those that are related to the human-centered aesthetic appeal of a language, formalism or technique but which are not directly concerned with ease of (re)use by others. Examples include “elegant”, “natural” and “intuitive”, although this category is perhaps better defined in part by the oft-used (in programming language writing) antonym of the above terms, “ad-hoc”.

Aesthetic properties were not considered “quality attributes” in the original formulation because that work viewed programming languages as “utilitarian artifacts” and considered properties of languages only insofar as they would affect programmers or users [6]. We note that these categories are not necessarily clear-cut or mutually exclusive. For example, by describing a language as “intuitive”, one might mean that it is easy for programmers to learn (an *effect on programmers*) or be referring to its intrinsic *aesthetic properties*. The line may not always be clear, but as a rule of thumb, we will consider a claim to be category 3 when it (implicitly or explicitly) refers to a user or programmer (this is, for example, likely to be the case if one is referring to an IDE, which is a tool designed for ease of programmer use, as “intuitive”) and as a category 4 claim when it does not (this is more likely to be

the case, for example, if one is referring to a calculus that is not intended for writing actual programs).

The four categories of claims are evaluated in different ways, as partially detailed in the original presentation of the first three categories above [7]. Generally, formal (category 1) properties are justified with a mathematical proof, or at least a sketch of one. Observational (category 2) properties are generally justified with quantitative experiments, such as performance evaluations. Coblenz et al. argue for various forms of user studies as the similarly definitive justification of category 3 properties. We agree that user studies are valuable tools to justify specific claims such as the following hypothetical “maintainability” claim:

Professional Language X programmers spent on average 34% less time fixing a particular bug in a Language X program than professional Language Y programmers spent fixing the same bug in a Language Y program.

We note, however, that a user study such as the hypothetical one above is subject to the cultural biases of the sampled population and the time at which the study was done: the same study done in a different country, or using undergraduates instead of professionals, or conducted 20 years later, might yield substantially different results. Well-done user studies will generally acknowledge these threats to validity, but might not recognize the cultural implications of the concept of “maintainability” itself: we don’t know who will be maintaining a particular piece of software, or for what purpose, or even what will constitute a bug in another community or in the future. This doesn’t mean that user studies can’t or shouldn’t be used to justify such claims, it just means that careful thought is required about the exact meaning of the claim being made and the nature of the justification. Category 4 (aesthetic) properties frequently go unevaluated in the literature, although these often accompany formalisms that arose from a particular mathematical process (such as the Curry-Howard isomorphism) or are particularly parsimonious. We will discuss the justification of category 4 claims in more detail later.

It is important to note that, although the meanings of formal and observational properties might be somewhat less subject to change or might change more slowly, they are not immune to the social processes we describe for “maintainability” above. Hopper’s anecdote from Section 1 illustrates how the seemingly objective notion of efficiency has changed drastically as computing hardware has developed. Claims such as “type safe” and “secure” are generally well-defined within the contexts of papers that make these claims, but what constitutes a “formal” pen-and-paper proof of such a claim is hardly standardized, and it is also largely determined by the audience what claims can be left unproven, claimed as trivial or attributed to folklore.

Nonetheless, as formal and observational claims generally have better-understood definitions and more agreed-upon justifications within the programming languages community, we will focus our attention in this essay on categories 3 and 4. Because such claims are not as well-defined and stable, we argue that there is a need for new ways to justify these claims. In the next section, we explore ideas in this direction, drawing on the field of rhetoric. Rhetoric engages heavily with studying constructs in natural language and understanding the impacts that texts have on audiences, in a similar way that the field of programming languages deals with the study of programming language constructs. Hence, we believe it is profitable to explore how ideas might be exchanged between the two disciplines.

3 An Interdisciplinary Approach to Justifying Claims

Regarding evaluative claims in programming languages, especially those in the 3rd and 4th categories of *effects on programmers* and *aesthetic properties*, we are faced with a challenge of how to justify evaluative claims that are necessarily audience-dependent. To offer a solution to this problem, we draw inspiration from the field of rhetoric and communication studies. As a field, rhetoric understands language as socially and historically contingent, which shapes rhetorical analysis that investigates how texts relate to and impact audiences. We describe the practice of rhetorical analysis briefly here, and then use this practice to inform our thinking about justifying category 3 and 4 claims.

3.1 The Contingency of Language

Natural language theorists would explain that all natural language is incredibly socially and historically contingent and constantly shifting in meaning. That is, words and phrases frequently change meaning, even in subtle and nuanced ways. For example, the phrase “family values” developed a different shade of meaning after it was used by members of the RNC in 1992 to frame themselves as opposed to certain LGBT and gender rights reforms, much as the term “fake news” draws up different images for us now in the Trump era than it did before 2016. In these cases, the actual meaning of terms changed after certain heavily publicized and politicized usages.

Mikhail Bakhtin, an early 20th century language philosopher, coined the terms *dialogic* and *heteroglossic* to explain this contingency in language [2]. He explains that when we speak, the words or phrases that we use are *dialogic* in that they have “taken meaning and shape at a particular historical moment in a socially specific environment” and thus are already “shot through” with meaning from all of the ways that they have been used before. For example, when Backus used the term “elegant” to describe an input language in 1978, he did so only having heard the ways that “elegant” had

been used to describe languages previously, which appears to have been applied to languages with different properties than the ones researchers today mark as elegant. And when researchers in 2020 use the term “elegant” to describe a language, their evaluation of what warrants the designation is based on the languages that they have seen the term applied to in the past.

However, the languages that researchers have heard the term “elegant” applied to even today will all be slightly different, based on the classes each researcher has taken, the other researchers they have collaborated with, the conferences they have attended, and so forth. In this sense, every person will have a slightly different understanding of what it means to be “elegant,” giving language a *heteroglossic* property of having multiple different shades of meanings for the same words or terms, which are socially and historically dependent. This explains why Isaacson is able to describe the lambda calculus as a “less beautiful solution” and leave programming language researchers baffled: he may be unaware of (or purposefully leaving out) the ways in which the programming language community labels things as beautiful and would understand the properties of the lambda calculus that make it so, but is instead basing his evaluation on a definition of beauty that he has acquired from other types of sources. If the evaluations that researchers use on their work are natural language terms and have the dialogic and heteroglossic qualities that Bakhtin theorizes about, then what exactly is meant by them is unstable in its dependence on time and community. That is, claims about a programming language may not hold up in 30 years. And perhaps more important, they may not hold up to those in other communities: an ethical problem arises when evaluative qualities are assessed mostly by those within the academy, rather than by those in other communities who either use or want to use these languages.

3.2 Rhetorical Analysis and Argument Theory

Because language has the property of being socially and historically contingent in its meanings, studying the effects of language on people must be grounded in these kinds of contexts. Rhetorical analysis is a broad category of techniques that try to understand how people influence each other through language in particular historical and social contexts [18]. When Aristotle defined rhetoric as “the art of discovering in any given case the available means of persuasion” [1], he was defining rhetorical analysis as a method which tries to describe how a text achieves certain effects on an audience. In this sense, rhetorical analysis is incredibly *descriptive* of the texts it analyzes, and grounds that description in context.

As an illustration, two foundational treatises in rhetorical studies have large sections dedicated entirely to various techniques of argumentation like making arguments from authority, creating analogies, and drawing on emotions, and

so forth [1, 17]. Someone conducting a rhetorical analysis of a text will describe different elements of the text and relate them to effect: for example, an analyst might describe how a series of historical events creates a certain exigence for speech, called a “rhetorical situation,” much like a mass shooting seems to historically prompt reflective speeches from politicians [3]; or an analyst might describe how a political memoir uses a large amount of first-person and narrative language in order to create a more personal identity and relate more to potential voters [13].

In rhetorical analysis then, evaluative claims about effects on audiences are justified by in-depth descriptions of the strategies that a text uses to create effects for certain audiences. Doing so requires a vocabulary for strategies and an understanding of particular audiences, and these allow for grounded justifications of claims about impacts on audiences. Argument theory, a subfield of rhetoric and communication studies, offers a framework we can adapt for thinking about how arguments are constructed wherein argumentative *claims* are justified with *reasons*, and reasons can then be explained with *evidence* [19]. When applying this argument framework to rhetorical analysis, argumentative claims about texts and what they do can be backed up with reasons in terms of descriptions of the rhetorical strategies they employ, which can then be evidenced with descriptions of the sociohistorical contexts of the audience that link effect and strategy. Justifying claims about effect in this way allows for critical thought about *why* certain effects are achieved and *how*, rather than simply *whether* the effect was achieved at that particular moment in time.

4 Application to Programming Languages

Just as rhetorical analysis uses descriptions of rhetorical strategies that texts use to produce certain effects on readers, we believe that programming languages researchers can describe features of programming languages or tools in order to justify that those features produce certain effects on programmers or users. In the argument theory framework described above, these features would serve as *reasons* to justify the claim made by the researcher. By making these reasons explicit, the researcher provides a clearer categorization of what is meant by the claim (e.g., the features of a language that the particular researcher believes to be elegant) and gives it a more stable justification.

This gives us the beginning of a framework for programming languages researchers to bolster their category 3 and 4 claims. In this section, we expand upon this framework in the hope that programming language researchers will find it useful. Because every paper and every claim is different, it is impossible for us to give a specific blueprint for authors to follow in every case. Instead, we will illustrate our framework through hypothetical examples as well as examples

drawn from authors already applying some of the techniques we suggest.

The first step in justifying a claim is to give *reasons* for it. As we introduced above, we suggest that these reasons should take the form of a description of the language or tool’s features. To illustrate, we will return to the example from the introduction of the claim that the lambda calculus is (or isn’t) “beautiful”. Suppose we wish to justify that the lambda calculus is indeed beautiful. We would begin by selecting features of the language that we believe contribute to its beauty. Choosing such features requires an understanding of audience. That is, descriptive features can only be argued to contribute to an effect insofar as an audience believes the two are related. Programming languages researchers might believe that the untyped lambda calculus is beautiful because of the simplicity of its definition and its closeness to mathematics, but an audience must believe that simplicity and closeness to mathematics are related to beauty for the lambda calculus to be described as “beautiful” using these reasons. If the audience of a description is other programming languages researchers in the current era trained in a similar tradition as the author, then these reasons will likely be convincing and the audience will believe the claim. A different audience (for example, readers of Isaacson’s book [11], or researchers in another field, or perhaps even programming languages researchers decades from now) might have very different notions of beauty and might not consider it clear why closeness to mathematics makes the lambda calculus beautiful. Then, although an argument that uses this as a reason to claim that the lambda calculus is beautiful might not convince such an audience, by making the reasons and assumptions explicit, the audience at least understands that this claim is being made on the basis of the language’s concision or other related features.

If writing for a programming languages audience then, we might write that “the untyped lambda calculus is a beautiful solution to the *Entscheidungsproblem* because through it, all of the complexity of Turing-completeness arises from just two constructs—lambdas and applications.” This simple sentence sets up our justification of the beauty of the lambda calculus: it is beautiful because (1) its axioms are simple, (2) its resulting behavior is complex and (3) formalisms with simple axioms and complex resulting behavior are beautiful. For audiences who believe that (1) and (2) are true of the lambda calculus and (3) is true in general, this reasoning should suffice. Of course, such views might not be universal, as we will revisit further in our discussion of *evidence*.

As a more concrete example drawn from a programming languages publication, the authors of the Cilk-5 parallel language [9] repeatedly highlight the “simplicity” of Cilk-5 relative to previous versions of the language.

[T]he language has been simplified considerably.
It employs call/return semantics for parallelism...

Merely having call/return semantics may not strike programming languages researchers as a particularly strong selling point for a language, but this does stand in contrast to earlier versions of the language (e.g., [5]). In these versions, functions spawned as parallel threads did not return but instead needed to be passed an explicit continuation, and instead of returning a value like a typical function, they would “send” their return value to the continuation using the `send_argument` keyword. The authors here claim that Cilk-5 is simpler because parallel threads are spawned in the same way as normal C functions: they are called with their arguments and return their return values. The comparison to C, which is implicit here, is made explicit later in the paper.

The basic Cilk language can be understood from an example. Figure 1 shows a Cilk program that computes the n th Fibonacci number. Observe that the program would be an ordinary C program if the three keywords `cilk`, `spawn`, and `sync` are elided... The semantics of a `spawn` differs from a C function call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C. [9]

This excerpt follows the basic claims and reasons framework by making a claim about the simplicity of the language, and giving reasons for it by emphasizing Cilk’s similarity to C and giving a code example to show that the code is fairly simple and resembles C. By providing the code example as a reason for simplicity (a common practice already in many papers describing new languages), the researchers allow the readers to decide for themselves if they believe the claim given the complexity of the task described (computing Fibonacci numbers, for example, is a fairly straightforward process) and the syntactic complexity of the code. In this case, the authors go further and justify why they believe the code shows the language to be simple (or, at least, simpler than earlier editions of Cilk): it is similar to C and thus, implicitly, understandable by audiences familiar with C.

There are a whole variety of possibilities for description, which can function as reasons for various evaluative claims. Consider this excerpt from the abstract of a PhD dissertation:

To argue that this system is elegant, I present a modal logic formulated for this purpose and then prove its global soundness and completeness and its equivalence to known logics. I then show how a small programming language can be derived from the logic, and how it can be implemented, proving properties of this abstract compilation procedure. All of these theorems are formalized in Twelf and can be checked by computer. [16]

This excerpt is in part interesting because the author explicitly marks an argument for claims about elegance (“To argue

that this system is elegant, I...”), but also because it demonstrates multiple techniques for backing up the evaluative claims made in this work: proving equivalence to known logics, being able to derive a small (small as opposed to complex with many orthogonal features) language, and that the system is simple enough to formalize in Twelf. With this level of description and reasoning, broader audiences are able to understand why the author considered the system elegant and what features of the system they should recognize as especially important to this quality (regardless of whether they agree with the author, or have similar definitions).

Thus far, the excerpts provide only a claim and reasons, but no evidence. This brings us to the second step of justifying a claim about a programming language or tool. In some cases, if it should be clear to the audience why the reasons justify the claim, evidence may not be necessary. Again, such reasoning depends on the audience and context. In the Cilk example, it is logical that similarity to a programming language with which a user is already familiar would make a new language easier to use. For an audience that one would expect to be familiar with C (and believe C can be described as “simple”), perhaps no explicit evidence is necessary to justify that Cilk-5 is simple because it is similar to C. Such evidence might be more necessary if the claim were targeted at non-programmers or at functional programmers, two groups that might not view C as “simple” (on the other hand, if the reason provided was similarity not to C but to Standard ML, such a justification would be persuasive if the audience for the claim was the first author of this essay, but might perhaps not be as convincing to, for example, the broader audience of SPLASH). Evidence might also be necessary if there were more reason to doubt the link between similarity and understandability.

When evidence is necessary to support reasons, the source of this evidence gets trickier in the case of programming languages research but should focus on the relationships between effects and audiences. For example, an agglomeration of several user studies across a substantial span of time and many demographics may be able to justify a broad preference for certain linguistic features over others. As an alternative, researchers could rely on studies from psychology of the effects of programming languages on programmers. While extensive research in this field remains to be done, the “cognitive dimensions” framework of Green et al. [4, 10] lays out a number of orthogonal dimensions along which a “notation” (a broad categorization that could include user interfaces, programming languages and language formalisms) can be evaluated for its psychological effects on users. In many cases, the inclusion and evaluation of these dimensions is backed by psychology research and user studies.

As it happens, some researchers are already relying on such psychology frameworks implicitly. The following excerpt justifies that a class of constraint logic programming languages is “intuitive.”

Intuition in the reasoning about programs is enhanced as a result of working directly in the intended domain of discourse. This contrasts with working in the Herbrand Universe wherein every semantic object has to be explicitly coded into a Herbrand term thus enforcing reasoning at a primitive level. [12]

The reason behind the intuition claim here is that, in the author’s semantics, representations more directly relate to the semantic objects one has in mind. It is not outlandish to claim that this would make a semantics more intuitive, and perhaps the authors felt that no additional evidence was necessary to support it. If the authors wished to provide such evidence, however, they could reference Green’s “closeness of mapping” cognitive dimension [10], which is motivated as follows:

Programming requires mapping between a problem world and a program world. The closer the programming world is to the problem world, the easier the problem-solving ought to be.

More thought may be required to determine the proper forms of evidence for justifying certain claims in programming languages. We feel that, at the very least, researchers can clarify their arguments for their audiences by making the reasons explicit through detailed description of their language or design and thinking carefully about whether evidence is possible or necessary to link the reasons and claims for the particular audience and context. As we have shown through the programming languages examples in this section, some researchers are already putting thought into crafting their arguments in this way, by providing descriptions of features of their work that back up the evaluative claims. Common practices such as listing code examples in PL design papers also fall within this framework, and so many authors are likely already putting our suggestions into practice without thinking explicitly about the framework of claims, reasons and evidence—by doing so explicitly, they could likely offer even clearer and more cogent arguments.

5 Conclusion

Like researchers in any discipline, programming languages researchers make claims in their publications, and these claims take many forms. Following Coblenz et al. [7], we have identified four categories of claims frequently made in papers about programming languages, analysis techniques, calculi and related concepts. While two of these (*formal* and *observational*, to borrow Coblenz et al.’s terminology) have well-accepted forms of justification, it is less obvious how to justify claims about effects on programmers, such as “usability” and effects about the aesthetics, such as “elegance” or “intuitiveness”, of a language.

In this essay, we have taken an interdisciplinary approach to look for ways to justify these latter sorts of claims. To do

so, we have drawn on a framework from argument theory of justifying *claims* using *reasons* and linking the two with *evidence*. For inspiration as to how to use this framework to justify claims in programming languages, we have looked to rhetorical analysis, in which claims about effects of texts on audiences are justified using thick descriptions of certain features of those texts, based on context. We believe that programming languages researchers can similarly describe the features of a language that they believe contribute to a desired effect and, where necessary, lean on evidence showing that those features indeed are linked to the effect.

The techniques presented here are in no way intended to replace other sorts of justifications where others are applicable: rather, we are looking to supplement other justifications, especially for sorts of claims that cannot be justified purely using existing empirical methods. In particular, we encourage researchers to use empirical methods where they exist and to continue to develop new empirical evaluation methods, but we believe there will continue to be claims (such as “beauty”) which cannot be justified empirically and yet still warrant scrutiny. Furthermore, we believe other evaluation methods and the framework we present here can complement each other: well-done user studies and other evaluations can give evidence to link features and effects in specific populations, and thinking carefully about features and audience can help put such studies in the proper context and help researchers recognize their limitations and fill in gaps in the justification of claims.

We hope that readers of this essay will have a heightened awareness of the claims they make in their own research, and in particular the level of subjectivity that comes with those claims and how the claims might be justified. For human-centric and aesthetic claims (Category 3 and 4 in our list) especially, we hope we have inspired researchers to recognize the subjective nature of these claims, to consider using description as a way to give reasons to their claims, and to consider what evidence, if any, might be necessary to link these reasons to the claimed effects. As evidenced by the examples in Section 4, many authors are already implicitly using reasons that fall within this framework; we believe that they could strengthen such claims by considering the claims-reasons-evidence framework more explicitly in their thinking, and encourage authors not already practicing these techniques to consider adopting them. We additionally hope that others in the field will join us in considering more carefully what kinds of evidence are possible to give broader and more useful justification to claims in the field of programming languages.

Acknowledgments

The authors would like to thank Michael Coblenz for helpful discussions on some of the concepts explored in this essay, as well as the anonymous reviewers for their useful feedback.

References

- [1] Aristotle. 1991. *On Rhetoric: A Theory of Civic Discourse*. Oxford University Press.
- [2] Mikhail Mikhailovich Bakhtin. 2010. *The Dialogic Imagination: Four Essays*. University of Texas Press. Original work published 1975.
- [3] Lloyd F Bitzer. 1992. The Rhetorical Situation. *Philosophy & Rhetoric* 1 (1992), 1–14.
- [4] Alan F. Blackwell and Thomas R.G. Green. 2003. Notational Systems – the Cognitive Dimensions of Notations framework. In *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, John M. Carroll (Ed.), 103–134. https://doi.org/10.1007/978-3-540-87730-1_4
- [5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (August 25 1996), 55–69. <ftp://theory.lcs.mit.edu/pub/cilk/cilkjpc96.ps.gz>
- [6] Michael Coblenz. 2020. Personal Communication.
- [7] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Boston, MA, USA) (*Onward! '18*). Association for Computing Machinery, New York, NY, USA, 133–146. <https://doi.org/10.1145/3276954.3276965>
- [8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) (*PLDI '02*). Association for Computing Machinery, New York, NY, USA, 234–245. <https://doi.org/10.1145/512529.512558>
- [9] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) (*PLDI '98*). Association for Computing Machinery, New York, NY, USA, 212–223. <https://doi.org/10.1145/277650.277725>
- [10] T.R.G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages & Computing* 7, 2 (1996), 131 – 174. <https://doi.org/10.1006/jvlc.1996.0009>
- [11] Walter Isaacson. 2014. *The Innovators: How a Group of Hackers, Geniuses, and Geeks Created the Digital Revolution*. Simon & Schuster, New York, NY, USA. <https://doi.org/10.17077/0003-4827.12228>
- [12] J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) (*POPL '87*). Association for Computing Machinery, New York, NY, USA, 111–119. <https://doi.org/10.1145/41625.41635>
- [13] David S. Kaufer and Shawn J. Parry-Giles. 2017. Hillary Clinton’s Presidential Campaign Memoirs: A Study in Contrasting Identities. *Quarterly Journal of Speech* 103, 1–2 (2017), 7–32. <https://doi.org/10.1080/00335630.2016.1221529>
- [14] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (*PLDI '05*). Association for Computing Machinery, New York, NY, USA, 15–26. <https://doi.org/10.1145/1065010.1065014>
- [15] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) (*POPL '05*). Association for Computing Machinery, New York, NY, USA, 378–391. <https://doi.org/10.1145/1040305.1040336>

- [16] Tom Murphy, VII. 2008. *Modal Types for Mobile Code*. Ph.D. Dissertation. Carnegie Mellon. <http://tom7.org/papers/> Available as technical report CMU-CS-08-126.
- [17] Chaïm Perelman and Lucie Olbrechts-Tyteca. 1971. *The New Rhetoric: A Treatise on Argumentation*. University of Notre Dame Press.
- [18] Jack Selzer. 2003. Rhetorical Analysis: Understanding How Texts Persuade Readers. In *What Writing Does and How It Does It*, Charles Bazerman and Paul A. Prior (Eds.). Lawrence Erlbaum Associates, Inc., Chapter 10, 279–307. <https://doi.org/10.4324/9781410609526>
- [19] Stephen Toulmin (Ed.). 1958. *The Uses of Argument*. Cambridge University Press, Cambridge, UK.
- [20] Richard L. Wexelblat (Ed.). 1978. *History of Programming Languages*. Association for Computing Machinery, New York, NY, USA.