On the limits of making programming easy

Tomas Petricek[1] and Joel Jakubovic[2]

[1] Charles University, Prague, Czechia
tomas@tomasp.net
[2] University of Kent, Canterbury, UK
joel.jakubovic@cantab.net

Abstract. A lot of programming research shares the same basic motivation: how can we make programming easier? Alas, this problem is difficult to tackle directly. Programming is a tangle of conceptual models, programming languages, user interfaces and more and we cannot advance all of these at the same time. Moreover, we have no good metric for measuring whether programming is easy. As a result, we usualy give up on the original motivation and pursue narrow tracktable research for which there is a rigorous methodology.

In this paper, we investigate the limits of making programming easy. We use a dialectic method to circumscribe the design space within which easier programming systems may exist. In doing so, we bring together ideas on open-source software, self-sustainable systems, visual programming languages, but also the analysis of limits by Fred Brooks in his classic "No Silver Bullet" essay. We sketch a possible path towards easier programming of the future, but more importantly, we argue for the importance of proto-theories as a method for tackling the original motivating basic research question.

Keywords: Programming systems · complexity · technical dimensions.

1 Introduction

In the Cambridge Computer Laboratory, in the area known as The Street in front of the lecture theaters, on the cold winter day of 1st December 2023, we encounter two computer scientists who arrived for the Mycroftfest symposium. They are Tomas Petricek, a former PhD student of Alan, and Joel Jakubovic, who is, at the time of our meeting, a PhD student of Tomas. Joel recalls how he enjoyed Alan's *Concepts in Programming Languages* module and the two start discussing their research.

2 Programming systems

TOMAS: I hope you are ready for your PhD defense next week, Joel! The examiners will surely ask you to give a brief summary of your work, so you'll need a good answer for that. Have you figured out how to summarize what your PhD work is about?

JOEL: I always like to introduce it as making programming "suck less!" I suppose I will need something that sounds more formal than that. Of course, I mean it in a very focused and specific way that is small enough to fit into a PhD.

TOMAS: My PhD was about making programming easier too. I worked on client-server web programming before [28] and found it tricky to keep track of the right context on the two sites. Some code requires resources that are only available on the server (e.g. a database), while some requires resources only available in the web browser (e.g., the user's location).

JOEL: So what did you do to make client-server web programming easier?

TOMAS: I tried to fix this by designing a context-aware programming language, that would track the context requirements, or *coeffects* in the type system [27]. With coeffects, the type would tell you not just what your functions take and return, but also that some part of your code can run only on some of the sites.

JOEL: And did that make programming easier?

TOMAS: Well, I ended up looking mostly at other use cases (dataflow, liveness) and only ever built a prototype implementation of a programming language with coeffect support.[3] But if someone implemented a proper programming language based on the theory, then I believe it would make programming client-server web applications easier...

JOEL: Oh dear... so your work was only concerned with improving programming *languages*.

TOMAS: Of course. Programming languages are the medium through which you program. But you make it sound like you have something else in mind.

JOEL: We both say that we are interested in making *programming* easier. Programming is about much more than programming languages and their formal properties. We should be thinking about programming *systems* more generally, not just about languages (see Figure 1).

TOMAS: What is the difference? What is a programming system?

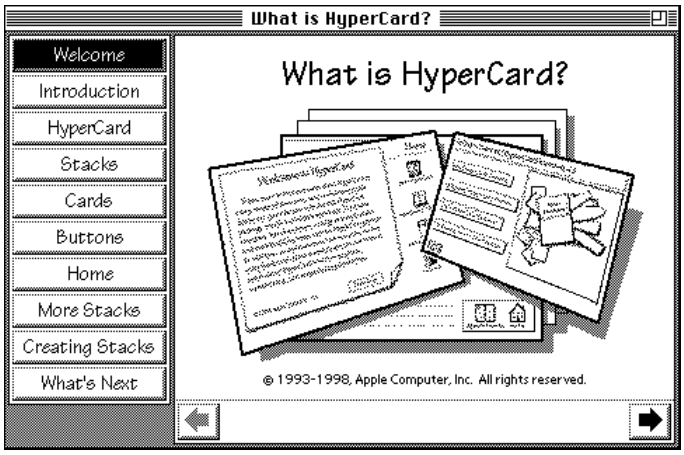
JOEL: A programming language is an abstract formal entity. A programming system is a running, interactive and stateful, collection of software. A system may implement a language (or multiple languages). It includes a runtime, a compiler or an interpreter, but it also consists of graphical interfaces used to edit code, debug code and so on.

[3] The prototype can be found, in the form of an interactive essay, at https://tomasp.net.

$$(var)\ \frac{x : \tau \in \Gamma}{C^{\mathbf{e}}\Gamma \vdash x : \tau} \qquad (app)\ \frac{C^r\Gamma \vdash e_1 : C^t\tau_1 \to \tau_2 \quad C^s\Gamma \vdash e_2 : \tau_1}{C^{r \vee (s \oplus t)}\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(sub)\ \frac{C^s\Gamma \vdash e : \tau}{C^r\Gamma \vdash e : \tau}\ (s \leq r) \qquad (abs)\ \frac{C^{r \wedge s}(\Gamma, x : \tau_1) \vdash e : \tau_2}{C^r\Gamma \vdash \lambda x.e : C^s\tau_1 \to \tau_2}$$

(a) Type and coeffect system for tracking context requirements [27].



(b) A screenshot of Hypercard, an early hyper-text system that allowed users to easily create their own interactive "decks".

Fig. 1: The contrast between programming langauges and programming systems. We know how to study *languages* (a), but not how to study *systems* (b).

TOMAS: I'm still not convinced. Of course, you need to write your code and invoke the compiler in some way. That's why you also have a terminal and Emacs to do programming.

JOEL: You are thinking about the problem through the lens of the *languages* paradigm. The point is that the *systems* paradigm gives you a different perspective, different problems and different research methods. Gabriel wrote about this in his "Structure of a programming language revolution" [14], where he points out that there was a paradigm shift through the 1990s from thinking about systems to thinking about languages. This encouraged more formal academic work on programming, but Gabriel also explains how some good ideas about programming became unthinkable because of this shift.

TOMAS: You can say that a programming language plus all the extra things it needs is a programming system, but what other good ideas about programming does this view enable?

JOEL: First, the different view lets us consider ways to make programming easier which are hard to think about as languages. For example, visual programming, block-based editing [30] or systems like Flash [2] and Hypercard. These are programming systems we should learn from. Second, the view lets us talk about interactivity and self-sustainable systems that can be modified from within themselves [19,18]. It would be useful to understand image-based programming systems like Lisp and Smalltalk, because they can in principle be used to make their *own* programming easier.

TOMAS: But Lisp and Smalltalk are languages ... ah, I see what you're going to say. Smalltalk-the-system *contains* Smalltalk-the-language, but a lot of other important things like its interface! And similarly for systems built on Lisp...

TOMAS: Those are very good points. I'm starting to believe that this programming systems view is actually useful. But let's return to it later, because I wanted to return to one more point you made...

3 Essential complexity

TOMAS: You said that the goal of your work was to make programming easier. I can see how thinking about programming systems rather than programming languages is a better way to approach the problem. Yet, I worry that this may not be as much help as you hope. Didn't Fred Brooks came up with an argument showing that you cannot make programming an order of magnitude easier?

JOEL: Interesting, why did he think that?

TOMAS: Brooks [6] analysed the different kinds of complexity that is involved in building software. The *essential complexity* is inherent in the real-world. It comes from the fact that our software has to encode the many (perhaps ad-hoc) rules of the human world. The *accidental compexity* comes from imperfection in our tools and languages.

TOMAS: Now, Brooks argues that we can reduce the accidental complexity through better tools, but the essential complexity will remain. He then points out that we will not be able to get an order of magnitude improvement in overall complexity of software unless the accidental complexity in software today is more than 10x that of the essential complexity.

JOEL: This may have been a valid point in the 1980s, but it assumes a very Waterfall view of software development. You only have to tackle the essential complexity all at once if you are designing a system from scratch. Not all programming is about that.

TOMAS: Programming methodologies have certainly evolved since Brooks' time. Today, you might use Agile methods rather than Waterfall, but I think the basic idea holds. You still have to construct the entire system with all its essential complexity.

Fig. 2: MIT Building 20 was a temporary timber building, erected for radar research during the World War II. It continued to exist for over 50 years and was highly popular for its infinite adaptability [5] (foto: MIT Museum).

JOEL: Well, I know you like to relate software development to architecture [26], so perhaps that can help me explain what I'm thinking.

TOMAS: Even more than in software, in architecture, you have to manage all the complexity and build the whole house! You can rely on modernist analysis or the traditional "unselfconscious" way of building that the early Christopher Alexander favoured [1], but you need to solve all the design problems the house involves.

JOEL: I think you can look at the problem very differently though. As argued by Stewart Brand [5], buildings also evolve. Old power stations become galleries, a garage becomes a co-working space, while a formal city plaza suddenly gains new use as an ice-skating rink [7]. Even when the spaces are not designed for their future use, they often end up fitting it remarkably well.

TOMAS: You are probably right. Buildings seem like the most static things, yet they evolve. Software is more dynamic, so it must surely evolve too. But how can this solve the problem of software complexity?

JOEL: Like buildings, I think we often build software by gradual adaptation. We may not always think about it that way, but we should and we should embrace this view. Very often, you have software that does *almost* what you want, but not quite.

TOMAS: I guess systems that you can buy and configure, like SAP that was slowly rising to prominence when Brooks was writing his paper [22], would be an interesting example of that.

JOEL: This is not quite what I had in mind, but fair enough! The point is, I think we can sidestep Brooks' "doom prophecy" by focusing on how software is built by making small changes to existing software. To make software development easy, we should make it easy to make *changes!*

TOMAS: Wait a second! Isn't building software in this way, through a sequence of small adaptations, something that the free software community already thought of in the 1980s? Having access to code and freedom to change it was the whole point of free software. The free software manifesto says you should have "The freedom to study how the program works, and *change it so it does your computing as you wish*."[13]

JOEL: In theory, yes, but does access to source code really give you the freedom to change a program as you wish for any moderately complex open-source software today?

TOMAS: I think you are right. The complexity of most software that we are dealing with today means that, even as a hacker myself, I cannot hope to understand and modify software with a reasonable amount of effort, even if I do have the source code.

JOEL: In fact, this probably worked in the 1980s when the GNU utilities were a manageable size and all the users were hackers who could modify and compile the code!

TOMAS: Isn't this inevitable? The kind of sofware we need today is just more complex, because computers get used for more complex problems. We are facing Brooks' essential complexity again. To make a small change to a large piece of software, you may not need to understand all of it, but you need to understand a significant part of it and even that is prohibitive.

4 Programming that scales

JOEL: Isn't it inevitable that static source code simply doesn't scale for understanding complex dynamic systems? The change we want to make is to the dynamic behaviour, but we have to trace the causality backwards and figure out which change to the *initial* state will produce it. At today's scales, this is like being a surgeon forced to operate on the DNA and then regrow the entire patient. Who would expect this to work?

TOMAS: Isn't the whole point of programming to tackle this kind of complexity through abstractions? Going back to my own PhD, a key idea in it was that you can hide multiple fairly complex specific instances of context-dependence behind a simple unified mathematical structure.

JOEL: Many computer scientists like to see programming as a formal mathematical activity, but this is just a metaphor and it has its limits. De Millo, Lipton and Perlis pointed this out in 1979 [8]! They believed that formal verification of software is bound to fail, because the complexity of the formal

entities *computer scientists* work with is different than the complexity of the formal entities *mathematicians* work with. A mathematical proof is probably correct because mathematicians excitedly go over them with their colleagues on a whiteboard, but no such social process accompanies formal proofs of software.[4]

TOMAS: If seeing programs as formal mathematical entities is not the right perspective, then what is the alternative?

JOEL: In the early 2000s, the Software Engineering Institute at Carnegie Mellon University asked this very question as part of their project on Ultra-large-scale systems [12] They said that we need to shift how we think about problems we face and that new perspectives will by inspired by work looking at disciplines such as microeconomics, biology, city planning, and anthropology.

TOMAS: I do like the city planning metaphor for thinking about software. If you look how people understand a city, as documented in the classic book "The image of the city" [23], I think much of that applies to source code. If you want to get from one place to another in a city, you do not need to understand the whole city. There are paths through the city that take you there like, for example, the metro line. Is this what you are thinking about?

JOEL: You can surely alleviate some of the scaling problems with source code by organising it better, but it is still wrong to have to search through the source code and look for names that sound like the right thing, in order to find the relevant code for the change we want to achieve.

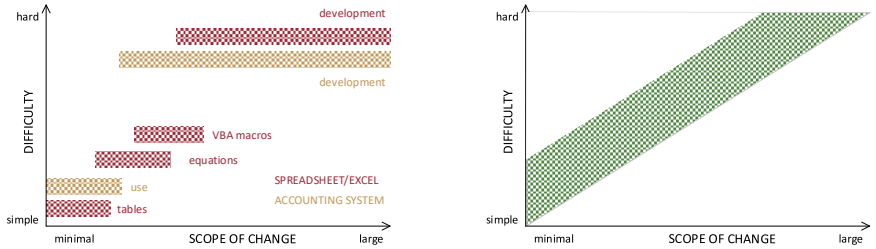
TOMAS: What do you imagine, then?

JOEL: Well, a programming system should model the chain of cause to effect in programs that you create. Then, when you want to make a change to something that you can see, or to something that produces visible outputs or effects on the screen, it can point you back to the code that you need to change.

TOMAS: Well, I'm not sure what the programmer interaction for this could look like, but I think I can imagine the technical characteristics of a system necessary to make it work. The system would need to record information about its execution and make it acessible. Basman [3] refers to this idea as "materialized execution" and traces it to the 1980s Boxer project [10]. A similar idea was also used recently to automatically link data visualizations by analysing code used to produce them [25].

JOEL: This is useful technical background, but we also need to look to interesting programming systems for inspirations about the interface through which we can make such program editing accessible.

[4] For a detailed and a refined account of the debate that followed the publication of the paper, as well as the historical context, see the book written by MacKenzie [24].



(a) Excel offers multiple but disconnected substrates (data entry, formulas, macros) than a reguler end-user application.

(b) An ideal system would have one adaptable substrate that makes minimal changes simple, but allows harder larger chages.

Fig. 3: Scope of change vs. difficulty of use of multiple programming substrates.

TOMAS: Perhaps the various live and exploratory programming systems [29] may be relevant here?

JOEL: Yes, those tools give you rapid feedback. Perhaps a surprisingly good example here is Excel, which also, to some extent, keeps track of the chain of cause to effect in that you can view the source of the formula and see what other cells were used to compute the result. The editing is somewhat limited in that you can only edit the current formula. There is no easy way to follow the chain of causes. But it is the best example I can think of.

TOMAS: Ah, but this only works with the relatively simple formula language. If you want to achieve something more complicated, you have to do that with VBA macros, which is as complicated as regular programming.

5 Programming substrates and notations

JOEL: Let's stop to think about notations for a bit. We can understand them better by charting the scope of change that can be achieved through a particular notation against the difficulty of using the notation. Figure 3a shows this for a hypothetical accounting system and a spreadsheet system.

TOMAS: I see. For an accounting system, you have the graphical interface that provides a notation for using the system and a notation (say, Java or C++) with which the system is implemented. For a spreadsheet, you have tables where you enter your data, formulas, VBA macros and then the notation used to implement Excel itself.

JOEL: Right. This is already better than regular software, because there are appropriate notations for changes of multiple levels of difficulties, but it is not perfect...

TOMAS: There is perhaps a nice characterization of the ideal notational system that we are looking for in the work of Pierre Depaz [9, p.123] on aesthetics of code. He says that *"simplicity is found in source code when the syntax and the ontologies used are an exact fit to the problem: simple code is code that is neither too precise, nor too generic, displaying an understanding of and a focus on the problem domain, rather than the applied tools."*

JOEL: I think this really boils down to two things. First, you need notational pluralism to be able to use the right tools for different jobs, and second, non-textual notations should be on the menu, because those are sometimes the most appropriate fit for a problem.

TOMAS: Hmm, but isn't notational pluralism impractical? In Excel, you have multiple disconnected notations such as tables, formulas and macros. But this also means that there is no gradual progression between them. If you learn how to use formulas, it does not help you at all with learning how to write macros (or even become a developer working on Excel at Microsoft...).

JOEL: Yes, the right approach seems to be to have an adaptable substrate. Something that has a basic form, but can be adapted in various ways to suit particular domain-specific problems. See the illustration in Figure 3b where the same (adaptable) substrate can be used for both small and simple changes, but also large and (inevitably) difficult changes.

TOMAS: But how would this work? You would need a substrate where an end-user can make a small change and gradually progress. The starting point then has to be something they are already familiar with. To take a spreadsheet as an example, do you think the substrate could somehow "grow" from filling numbers in a table to modifying some simple aspects of the system that users may care about, like making the interface color scheme colour blind friendly or making the wording of menu items clearer?

JOEL: I imagine it could work in spreadsheets. You could have more things in Excel modifiable in a table. The key/value pairs in the preferences menu could, in addition to their normal UI, be brought into the table substrate. More of the interface could also be defined in a table. And then you could also have your preferences computed with formulas!

TOMAS: Hmm, are you getting at some general principle with this idea?

JOEL: The notation for development should resemble the notation for use!

TOMAS: Well, to take this idea to its limits, are you saying that MS Paint should be written in the Piet esolang that we can see in Figure 4?!

JOEL: Fair enough. The principle breaks down with absurd examples if you take it literally, but that doesn't mean the principle will never work. It is still useful, but you need to think about what really is the essence of the substrate through which most people interact with computers today. It is not pixels...

TOMAS: Do you mean buttons, tabs, tables, lists, forms and such?

Fig. 4: “Hello world” program in the esoteric programming language Piet. The program instructions are encoded as colours and are executed as a pointer moves over the individual colour regions of the bitmap.

JOEL: I believe there is a more fundamental substrate behind all of those. You can model a lot of user interface widgets as rigid bodies that exist in some space, connected with rods and springs and perhaps forces such as attraction and repulsion between particular entities.

TOMAS: How do you imagine this would provide a programming substrate?

JOEL: I do not have a perfect answer to this question, but you can think of the basic substrate as a sort of “user interface physics” within which specific visual notations for particular problems could be defined.[5] If we had that, then I think we would make a step towards the ideal substrate for programming. And the substrate needs to be talking about actual running materialized system, rather than a static pre-scription of it that is a computer program in a static programming language today.

6 Research methodologies

TOMAS: You certainly have some very interesting ideas about the future of programming here, but let me ask an annoying practical question. How do you turn this into a realistic research project? Is there something you can analyse formally or measure about a prototype? To make this into a research agenda, there needs to be some scientific methodology behind it!

JOEL: Tomas, remind me, didn’t you run into a problem in your thesis with the intersection of provable and interesting things?

TOMAS: Well, you remember correctly. There was a point in my PhD when I realized that all the things I could prove about my programming langauge were not that interesting and all the things that I found interesting about programming were not particularly formalizable or provable. I guess I was getting increasingly interested in what people sometimes call “programming

[5] Constraint-based systems for specifying user interfaces, such as Cassowary [4], can be seen as related to this idea, although they focus more on relating elements for the end-user than on developing a foundational underlying substrate.

experience" research.[6] The experience also made me think much more about research methodology and led me to interest in history and philosophy of science.

JOEL: Does that shed any light on why programming langauge researchers are so focused on formalization and proofs?

TOMAS: Well, for one thing, there is a bias for theory over experiment in all branches of science. Ian Hacking [17] pointed this out in the case of physics and the same applies to computer science. However, finding a suitable methodology for programming experience research is an open problem that does not have a definite answer. It may need a range of qualitative and quantitative methods, including formalisms and empirical evaluation, but also system descriptions, user studies and perhaps even interactive artifacts [11].

JOEL: I do not understand how you are supposed to make general and unifying claims if you do no first start by looking at the concrete and specific!

TOMAS: Well, there are some counter-examples to this. For most of its history, particle physics consisted of two sub-cultures that Peter Galison refers to as the "image" and "logic" traditions [15]. While the latter focused on statistical analysis of observations, the former attempted to capture "golden events" that show the decay of a single particle. Such concrete trace was then hugely influential in shaping the theory of particle physics. And in the case of city planning, Jane Jacobs made a similar argument [20] in favor of looking at specific cases, which she calls "unaverage clues". An example is a chain of bookshops that always remain open until late, except for one branch in Brooklyn. The clue tells you something significant about that part of the city.

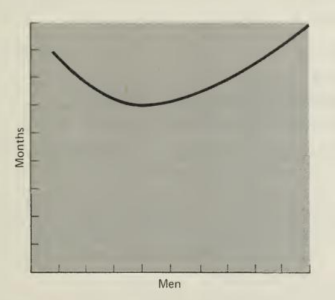
JOEL: You need something in between. In order to do any formalisation or a user study, you need conceptual clarity on what you're studying and what you're trying to find out. I don't think we have any such clarity on many interesting aspects of programming systems.

TOMAS: But what kind of knowledge structure can give you such conceptual clarity, without being as detailed as a formal model?

JOEL: You made a reference to Fred Brooks earlier. I think his other essay "The Mythical Man-Month" [21] is a good example. In the essay, Brooks explains why adding more workforce to a delayed project only makes the delay worse. The argument is based on analysing the amount of communication that needs to happen (Figure 5). For a perfectly partitionable programming task, adding people helps. But if the work requires communication, the overhead of the communication grows exponentially as people are added, eventually overtaking the gain you get from the linear growth of people who can do the programming. Brooks illustrates this with charts and equations. He does not

[6] The topic is the subject of the Programming Experience workshop (https://programming-experience.org) that has been running since 2016.

(a) Time versus number of workers: Perfectly partionable task



(b) Time vs. number of workers: Task with complex inter-relationships

Fig. 5: Two illustrations by Brooks showing how communication overhead means that adding workforce to a delayed project makes things worse (from [21])

have formal proofs or specific numbers, but the model provides a good theory that provides conceptual clarity.

TOMAS: Interesting, this kind of reasoning seems to be quite common in software engineering. I can think of another example from a book about the history of anti-ballistic missile systems [31]. One of the arguments for why such system cannot be reliably built was based on comparing the rate of change in the environment and the rate of change of the system. The argument was that the US cannot build a reliable defence system, because changing the software is more work than changing the characteristic of the Soviet ballistic missile. Again, this is a somewhat mathematical argument that provides conceptual clarity.

JOEL: Right. I think we can refer to such models that provide conceptual clarity protot-theories and I would say that we need more thinking like this! They can be later refined to more formal modles, but not prematurely. I would also add that mathematical formulations like those above are not the only kinds of proto-theories we should be creating.

TOMAS: So what would be a useful proto-theory for thinking about programming systems?

7 Technical dimensions of programming systems

JOEL: Well, a good place to start would be the published framework of "technical dimensions" which break down programming systems into narrower properties of which they can have more or less.

TOMAS: Dimensions? That rings a bell...

JOEL: You're thinking of the Cognitive Dimensions of Notations [16] which was the main inspiration. However, the Technical Dimensions are concerned with the programming system as a whole, not just its notation.

TOMAS: I see! How does such a framework contribute to making programming easier?

JOEL: By providing a way for us to actually learn something from the vast range of programming systems that aren't necessarily languages. It is a set of common points of reference that allow for an apples-to-apples comparison of widely differing software systems.

TOMAS: OK. Is this meant to supersede programming language research methods?

JOEL: Not at all. Since many programming systems contain a programming language, all existing PL research is applicable to their syntax, semantics, types, paradigms, and so on. The Technical Dimensions simply establish the analogues of those concepts for the other parts of those systems.

JOEL: Behold: clusters of technical dimensions down the side, and programming systems along the top. Each cell is a concise account of how the system measures against those dimensions.

TOMAS: This is so informal! How does this help?

JOEL: The full realisation of the aims of the framework is a long-term research program. What we have already is the seed to start it off. We have suggested a set of dimensions and described each one qualitatively. Future work, as set out towards the end of my dissertation, involves making these descriptions more precise and possibly even qualtifiable.

TOMAS: Quantifiable? Does this risk the problem we mentioned about formalizable vs. interesting properties?

JOEL: That spectre is always hanging over it, yes. But the solution is to refine the framework slowly and carefully, with input and critique from the research community. This is why the dimensions were published in their qualitative state; better to grit our teeth and endure the informality, than to prematurely formalise before we know what we're doing! Just the fact that we're breaking down programming systems into a common vocabulary of properties to focus on, helps to analyse programming systems and compare them with each other. It also means we can map the design space of possible systems and spot combinations of properties that haven't been tried yet.

TOMAS: Aha, so while you have been outlining the future work for interested "theoreticians", the future work for "experimentalists" would be filling the database of how different systems fit into the framework.

JOEL: Exactly - there is a lot of what Kuhn called "normal science" waiting to be done here.

8 Conclusions

Acknowledgments. Omitted for initial submission.
This study was supported by X (grant number Y).

References

1. Alexander, C.: Notes on the Synthesis of Form. A Harvard paperback, Harvard University Press
2. Ankerson, M.S.: Dot-com design: The rise of a usable, social, commercial web, vol. 15. NYU Press (2018)
3. Basman, A.: Boxer and the tradition of materialised programming (2022), conference talk presented at Boxer Salon
4. Borning, A., Marriott, K., Stuckey, P., Xiao, Y.: Solving linear arithmetic constraints for user interface applications. In: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology. p. 87–96. UIST '97, Association for Computing Machinery, New York, NY, USA (1997). https://doi.org/10.1145/263407.263518
5. Brand, S.: How Buildings Learn: What Happens After They're Built. Viking (1994)
6. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. Computer **20**(4), 10–19 (1987). https://doi.org/10.1109/MC.1987.1663532
7. Cohen, S.: Physical context/cultural context: Including it all. OPPOSITIONS **2**, 1–40 (January 1974)
8. De Millo, R.A., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. Commun. ACM **22**(5), 271–280 (may 1979). https://doi.org/10.1145/359104.359106
9. Depaz, P.: The role of aesthetics in understanding source code. Ph.D. thesis, Université Sorbonne Nouvelle, ED120 - THALIM (2023)
10. diSessa, A.A., Abelson, H.: Boxer: a reconstructible computational medium. Commun. ACM **29**(9), 859–868 (sep 1986). https://doi.org/10.1145/6592.6595
11. Edwards, J., Kell, S., Petricek, T., Church, L.: Evaluating programming systems design. In: Marasoiu, M., Church, L., Marshall, L. (eds.) Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2019). Newcastle University, UK (August 2019)
12. Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems: The Software Challenge of the Future (Jun 2006), https://insights.sei.cmu.edu/library/ultra-large-scale-systems-the-software-challenge-of-the-future/, accessed: 2024-Feb-5
13. Foundation, F.S.: What is free software (version 1.1) (2001), https://www.gnu.org/philosophy/free-sw.en.html, accessed on: 5 February 2024
14. Gabriel, R.P.: The structure of a programming language revolution. In: Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. p. 195–214. Onward! 2012, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2384592.2384611
15. Galison, P.: Image and Logic: A Material Culture of Microphysics. University of Chicago Press (1997)

16. Green, T.R.G.: Cognitive dimensions of notations. In: Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V. p. 443–460. Cambridge University Press, USA (1990)
17. Hacking, I.: Representing and Intervening: Introductory Topics in the Philosophy of Natural Science. Cambridge University Press, Cambridge (1983)
18. Hirschfeld, R., Masuhara, H., Rose, K. (eds.): Workshop on Self-Sustaining Systems, S3 2010, Tokyo, Japan, September 27-28, 2010. ACM (2010). https://doi.org/10.1145/1942793
19. Hirschfeld, R., Rose, K. (eds.): Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers, Lecture Notes in Computer Science, vol. 5146. Springer (2008). https://doi.org/10.1007/978-3-540-89275-5
20. Jacobs, J.: The Death and Life of Great American Cities. Random House (1961)
21. Jr., F.P.B.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Professional (1975)
22. Leimbach, T.: The sap story: Evolution of sap within the german software industry. IEEE Annals of the History of Computing **30**(4), 60–76 (2008). https://doi.org/10.1109/MAHC.2008.75
23. Lynch, K.: The Image of the City. MIT Press, Cambridge, MA (1960)
24. MacKenzie, D.: Mechanizing Proof: Computing, Risk, and Trust. The MIT Press (09 2001). https://doi.org/10.7551/mitpress/4529.001.0001
25. Perera, R., Nguyen, M., Petricek, T., Wang, M.: Linked visualisations via galois dependencies. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498668
26. Petricek, T.: Programming as architecture, design, and urban planning. In: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. p. 114–124. Onward! 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3486607.3486770
27. Petricek, T., Orchard, D.A., Mycroft, A.: Coeffects: Unified static analysis of context-dependence. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D. (eds.) Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II. Lecture Notes in Computer Science, vol. 7966, pp. 385–397. Springer (2013). https://doi.org/10.1007/978-3-642-39212-2_35
28. Petricek, T., Syme, D.: F# web tools: Rich client/server web applications in f#. Unpublished draft. Online at https://tomasp.net/academic/articles/fswebtools/fswebtools-ml.pdf
29. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding. The Art, Science, and Engineering of Programming **3**(1) (2018)
30. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y.: Scratch: programming for all. Commun. ACM **52**(11), 60–67 (nov 2009). https://doi.org/10.1145/1592761.1592779
31. Slayton, R.: Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012. MIT Press (2013). https://doi.org/10.7551/mitpress/9234.001.0001