On the Limits of Making Programming Easy

Tomas Petricek[1] and Joel Jakubovic[1,2]

[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 tractable 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 not only 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

A cold winter morning, 1st December 2023, outside the Cambridge Computer Laboratory; two computer scientists arrive in the area known as The Street in front of the lecture theaters; they came to attend the MycroftFest symposium: Tomas Petricek—a former PhD student of Alan—and Joel Jakubovic, Tomas' PhD student. The elder and younger researchers discuss threads of programming study as different from each other as The Street and the Cambridge Computer Laboratory are from Trinity Street and Peterhouse.

Let's eavesdrop.

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 [35] 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 [34]. With coeffects (Figure 1a), the type would tell you not just what your functions take and return, but also what resources they require. Coeffects would tell you that some part of your code can run only on some of the sites.

JOEL: I thought coeffects were about tracking how programs use variables. The two examples I saw involved dataflow and liveness. There was no mention of client-server web programming!

TOMAS: Well, yes, there is some irony in the fact that the original motivation did not actually fit all that well with the formal model that we developed. Dataflow and liveness were much better examples.

JOEL: And did coeffects make programming easier in the end?

TOMAS: I only ever built a prototype implementation of a language with coeffect support,[3] but if someone implemented a proper language based on the theory, then 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).

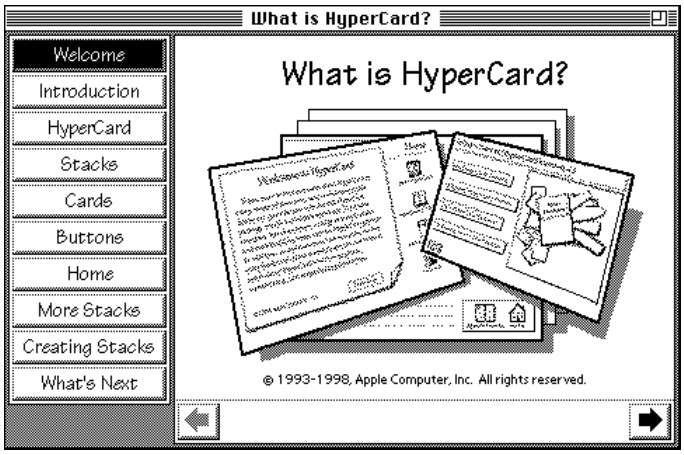
TOMAS: What is the difference? What is a programming system?

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

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

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

(a) Type and coeffect system for tracking context requirements [34].



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

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

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 and a compiler or an interpreter, but it also consists of graphical interfaces used to edit code, debug code and so on.

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 still 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" [16], where he points out that there was a paradigm shift through the 1990s from thinking about systems to thinking about languages.

JOEL: The paradigm shift encouraged more formal academic work on programming, but Gabriel also explains how some good ideas about programming became unthinkable because of this shift.

TOMAS: Does he say unthinkable, or unpublishable in OOPSLA?

JOEL: The latter, but in practice, the two co-evolve and end up being the same thing! What can be published tends to shape people's research agendas and their way of thinking about problems...

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 [37] 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 [21,20]. It would be useful to understand highly malleable 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...

JOEL: Exactly, and there is much more than the interface. When you think about Smalltalk-the-system or Lisp-the-system, you also need to consider all the libraries that are readily available to you as parts of the system.

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 show that you cannot make programming an order of magnitude easier?

JOEL: Interesting, why did he think that?

TOMAS: Brooks [7] analysed the different kinds of complexity involved in building software. The *essential complexity* is the inherent complexity of the concepts that software works with. It arises from the data sets, interconnections, algorithms and operations that the software is to perform. The *accidental complexity* comes from imperfection in our tools and languages.

JOEL: This would make more sense to me if the essential complexity came from the fact that our software has to encode the many (perhaps ad-hoc) rules of the human world. Algorithms and data structures sound less essential to me.

TOMAS: I like your interpretation, but maybe the algorithms and data structures that Brooks talks about are reflections of the real world? In any case, 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 assumes that you have to tackle the essential complexity all at once, but that is only the case if you are designing a system from scratch. Did Brooks say anything about open source that was emerging at the time and avoids this problem by letting anyone use and adapt software as they wish?

TOMAS: No, Brooks does not mention open source. He talks about buying off-the-shelf packages as an alternative to building software. Even with open source, I think Brooks' basic idea still holds. No matter if you use open source or replace the Waterfall of the 1980s with Agile methodologies of the 2020s, you still have to put together the entire system with all its essential complexity.

JOEL: Well, I know you like to relate software development to architecture [32], so perhaps that can help me explain what I'm thinking.

TOMAS: Architecture needs upfront planning even more than software. You first have to resolve all the design problems that the design of the house involves. There are different ways of doing this. You can rely on modernist analysis or the traditional "unselfconscious" way of building that the Christopher Alexander favoured in his early work [1], but you still need a detailed plan before you start building.

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 [8]. 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, being structured information, ought to be much more malleable than the rigid materials of buildings; accordingly, it evolves much faster. 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 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 [27], would be an

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] (photo: MIT Museum).

interesting example of that. Contemporary no-code and low-code systems go even further in that they blur the distinction between code and configuration.

JOEL: This is not quite what I had in mind, but fair enough! My point is that 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! Aren't we getting back to open source and the free software movement? 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*."[14] This sounds very much like building software through a sequence of small adaptations.

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, the freedom to change a program as you wish 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: 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 [9]! 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 it on a whiteboard with their colleagues, 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 [13] 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" [28], 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?

[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 [29].

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. Bret Victor demonstrated what this could look like [39], but I'm unaware of any follow-up work to realise his vision—which raises the question of how he himself got his demo to work, something that does occasionally keep me up at night.

TOMAS: I think I can imagine the technical characteristics such a system would need to work: it 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 [11]. We used a similar idea recently to automatically link data visualizations by analysing code used to produce them [31].

JOEL: This is useful technical background, but we should also look to interesting *real* programming systems—not just demos—for inspiration about interfaces that would make such program editing accessible.

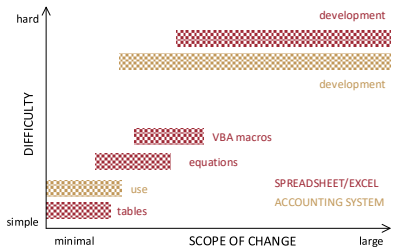
TOMAS: Perhaps the various live and exploratory programming systems [36] may be relevant here?

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 which other cells were used to compute the result. You do have to backtrace the causal chain manually by looking up the cells one by one. But it is the best example I can think of.

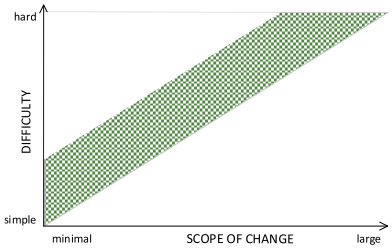
TOMAS: Excel has surprisingly good backtracing support these days! But regardless of whether you trace manually or with the system's help, this only works with the relatively simple formula language. If you want to achieve some more complicated behaviour, you have to do that with VBA macros and give up this traceability.

5 Programming substrates and notations

JOEL: True, so 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.



(a) Excel offers multiple but disconnected substrates (data entry, formulas, macros).



(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: 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 at multiple difficulty levels, but it is not a perfect system of notations yet...

TOMAS: There is perhaps a nice characterization of the ideal notational continuum that we are looking for in the work of Pierre Depaz [10, 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, to be able to use the right tools for different jobs, you need to be free from a restriction to just one notation. Second, you should also be free from a restriction to plain text; non-textual notations should be on the menu, because those are sometimes the most appropriate fit for a problem.

TOMAS: Hmm, but wouldn't such a "notational pluralism" be impractical? In Excel, you may use multiple disjoint notations such as tables, formulas and macros—yet 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 C++ 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 par-

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.

ticular 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: This reminds me of a brief research programme focused on “extensible programming languages” in the late 1960s [33], which tried to solve the problem of designing a universal programming language by having a base language that can be extended by the user. I’m still trying to figure out what happened to this idea...

JOEL: Ah, but you are talking about languages again, rather than systems!

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: Something like this could work in spreadsheets. I can think of some parts of Excel-the-application that transfer well to a portion of a spreadsheet. One could press a button to “reflect” the key/value pairs in the preferences menu into two columns of cells. When edited, these cells would update Excel’s internal preferences that you normally edit in a dialog box. At a stretch, perhaps other parts of such a dialog could also be reflected—say, the list of options in a drop-down box. And then you could, for example, 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, this is a heavy constraint on the notations we can use for development! What would justify such an idea?

JOEL: I admit, it’s about as restrictive as one could get! The motivation is that the user is already familiar with the “notation for use” in the software—so to the extent that “development” can be massaged to use the same nota-

tion, they are able to adapt the software by re-using the same skills. They no longer have to invest in learning a completely different notation (like a programming language) before they can do that.

TOMAS: So, to take this idea to its limits, you're saying that Microsoft Paint should be written in the Piet esolang that we can see in Figure 4?!

JOEL: Fair enough. I wouldn't go so far; *given that* the use-notation in Paint is what it is, there aren't many changes to Paint that fit sensibly within it. That being said—the program's tool icons could be "reflected" in its canvas and re-drawn that way, but that's the only straw I can think to grasp!

JOEL: The principle will break down with absurd examples if you commit to it fully, but that doesn't mean we shouldn't explore its domain of validity. It is still promising, but you need to think about what is the essence of the substrate through which most people interact with computers. It is not pixels...

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

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] Such a substrate could even be the result of taking a 2D physics engine [15], stripping it of concepts like mass and rotations, and optimising it for this simpler domain.

JOEL: If we had that, then I think we would make a step towards the ideal substrate for programming. And the substrate would need to live within an actual running materialized system, rather than a static blueprint like a program in source code 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

[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 rendering the layout for the end-user than on developing a foundational underlying substrate. Additionally, individual constraints are too low-level for a human-friendly UI substrate, although they could well be what such a substrate "compiles" to in order to run. This was the subject of much research in the 1990s [30].

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: You remember correctly. There was a point in my PhD when I realized that all the things I could prove about my programming language were not that interesting, and all the things that I found interesting about programming were not particularly formalizable. I guess I was getting increasingly interested in what people sometimes call "programming 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 language researchers are so focused on formalization and proofs?

TOMAS: For one thing, there is a bias for theory over experiment in all branches of science. Ian Hacking [19] pointed this out in the case of physics, as did Hossenfelder [22], and the same applies to computer science. 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, empirical evaluation, but also system descriptions, user studies and perhaps even interactive artifacts [12].

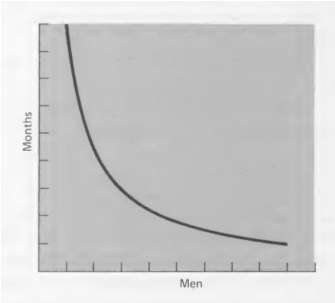
JOEL: I do not understand how you are supposed to make general and unifying claims if you do not first start by looking at the concrete and specific!

TOMAS: Fair enough, and there are two examples to support your perspective. For most of its history, particle physics consisted of two sub-cultures that Peter Galison refers to as the "image" and "logic" traditions [17]. While the latter focused on statistical analysis of observations, the former attempted to capture "golden events" that show the decay of a single particle. These concrete traces went on to be hugely influential in shaping the theory of particle physics. And in the case of city planning, Jane Jacobs made a similar argument [23] 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.

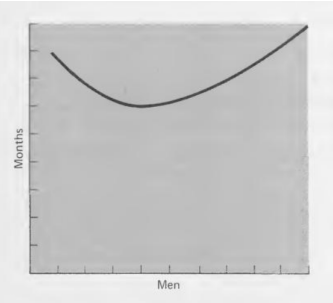
JOEL: OK, but on reflection, we need something in between. In order to do any formalisation or a user study, you need conceptual clarity on what you're studying and what you're trying to find out. I don't think we have any such clarity on many interesting aspects of programming systems.

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

[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 [6])

JOEL: You made a reference to Fred Brooks earlier. I think his other essay "The Mythical Man-Month" [6] 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 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 [38]. One of the arguments for why such systems cannot be reliably built was based on comparing the rate of change in the environment and the rate of change of the system. The idea was that the US cannot build a reliable defence system, because changing the software takes more work than changing some of the characteristics of the Soviet missiles. Ergo, their defences won't be able to keep pace with Soviet innovations in offensive capability. 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 *proto-theories*. I would say that we need more thinking like this! They can be later refined to more formal models, 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: Well, I would have loved to pursue proto-theories during my PhD—but I fear I would not have been able to defend it!

JOEL: Exactly—there needs to be a way for researchers to get credit for doing this sort of work; until then, it's disincentivised. The *Onward!* tracks at SPLASH provide this opportunity, but there ought to be more venues for it.

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

7 Technical dimensions of programming systems

JOEL: Well, a good place to start would be the published framework of "technical dimensions" [25] 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 [18] which was the main inspiration. However, the Technical Dimensions are concerned with the programming system as a whole, not just its notation.

TOMAS: I can see how you capture various properties of notations. They can be more or less dense, more or less consistent, more or less easy to modify etc. But what kind of properties can you capture as dimensions about programming systems in general?

JOEL: The list is quite long and I do not think it is complete. but it includes some of the characteristics of programming systems that we have talked about already. For example, *modes of interaction* captures whether you do everything in a single environment or whether "using" is distinct from "programming"; *notational structure* captures whether there is just a single adaptable notation or a range of different ones; *self-sustainability* captures the degree to which the system can be modified from within itself and so on.

TOMAS: Just to get back to our original question, how does such a framework contribute to making programming easier?

JOEL: Remember Figure 1, which illustrated that we know how to study programming languages, but not how to study programming systems? Well, many people came up with interesting programming systems that make programming easier in some way. But because we lacked a good way for talking about them, the ideas were never clearly described and nobody was able to adopt them and develop them further.

TOMAS: This is the incommensurability that Gabriel [16] warns us about! He would chuckle and reminisce about the glory days of Lisp systems in the 1980s, before shift to the mathematical paradigm made all the programming systems work incomprehensible.

JOEL: Alas, we cannot travel in time back to the 1980s! We should take time *today* to set up a framework for programming languages people to understand and appreciate work on programming systems.

TOMAS: I guess what you are saying is that technical dimensions of programming systems gives us a set of common points of reference that allow for apples-to-apples comparison of widely different programming systems? But is this meant to supersede the existing programming language research methods?

JOEL: Not at all. Since many programming systems contain a programming language, all existing programming language 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.

TOMAS: Didn't you link the change from thinking about programming languages to thinking about programming systems to the idea of Kuhnian scientific revolution before?

JOEL: Yes, it would be a corrective to the shift that Gabriel observed in the opposite direction [16]; not by *fighting* PL research, but by simply making space to do the more general thing. To truly establish a new paradigm, the return to thinking about systems would put more emphasis on research methods that can talk about them in their full complexity. It would not *invalidate* programming language theory, but it could make it more of a speciality instead of the default. However, we are quite far from what Kuhn called "normal science". It is still early days for programming systems research!

TOMAS: What do you think still needs to be done? I have to say that, compared to established programming language research methods, the work on technical dimensions seems quite informal!

JOEL: Right. 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 a set of dimensions that come with qualitative descriptions. Future work, as set out towards the end of my dissertation [24], involves making these descriptions more precise and possibly even quantifiable.

TOMAS: Quantifiable? Does this risk the problem we mentioned about focusing on what is formalizable rather than on what is interesting?

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: It is funny to hear us discussing the theoretical framework of programming systems, after we complained about the scientific bias in favour of theoretical work! You have just outlined the future work for interested "theoreticians", but the future work for "experimentalists" should be at least as

important. I guess the experimental work should be filling the database of how different systems fit into the framework.

JOEL: That is certainly one task for the experimentalists. But they can also explore the design space to find new designs. To get back to our original motivation of making programming easier, my thesis explores programming systems that have three properties: *notational freedom*, which would give you the kind of adaptable substrate we talked about, *explicit structure*, which is a pre-condition for building such a substrate, and *self-sustainability*, which would make it possible to evolve the system from within itself, enabling a virtuous cycle of self-improvement.

TOMAS: So your thesis not only proposes a proto-theory, but it also showcases an application of it. Is that something that you'll be able to defend?

JOEL: Let's hope so! I do feel prepared to defend it; we'll find out for sure in one week's time![7]

8 Conclusions

ALAN (*arrives in his festive Christmas jumper*): HO HO HO! It is nice to see you! What have you two been chatting about here?

TOMAS: Joel was practicing the elevator pitch for his PhD thesis, but we got a bit distracted. We ended up talking about whether it is possible for programming to be easier, whether there are any fundamental limits to that, and also what kind of research methodology can get us there!

ALAN: And what did you conclude?

JOEL: On a more general level, we agreed that we need more thinking about interactive, stateful programming systems and their characteristics. Programming can only become easier if we start thinking about how programming systems are used. In other words, think more about *programming systems* than about *programming languages*. Today, this is tricky, because we lack the right vocabulary for talking about programming systems, but the technical dimensions are a useful *proto-theory* that serves as the starting point!

ALAN: And on the more specific level?

JOEL: We do not have a concrete system yet, but we believe it will have a number of general properties. It needs to be built around a programming substrate that can grow with the user. It should make it easy to make small changes to the system, but also allow making more complex and difficult changes. We also need to think about programming as gradual adaptation, to overcome Brooks' "doom prophecy" of essential complexity. But I remain optimistic.

TOMAS: My tongue-in-cheek suggestion is that your *Concepts in Programming Languages* module, which I very much enjoyed supervising back in the day,

[7] It turned out such a thing did work out after all.

should become *Concepts in Programming Systems*! We need a new generation of researchers who will adopt the programming systems perspective and come up with great new theories and experimental systems!

ALAN: So, what do you think the next generation of Computer Lab programming systems PhD students should be doing?

JOEL: I would love to see more research that looks at how programming languages are embedded in broader stateful interactive systems. This is something where we are lacking both theory and new interesting design ideas. I also think programming researchers should look at things that we do not really see as programming languages. What is it that makes systems like Excel, but also Flash, Hypercard or game engines work well as programming tools? We also need to be more fearless in looking at problems that are interesting, but informal. If we are to study informal properties that are not simply true or false, we will also need more focus on finding consensus among researchers working on programming tools, such as agreeing on the right proto-theories on which to further advance our work.[8]

Acknowledgments. The events portrayed in this essay are entirely fictional. Any resemblance to actual persons, living or dead, is purely coincidental. But we are grateful to the non-fictional people who participated in a lively discussion at Mycroftfest and helped to shape this essay, as well as to the anonymous reviewers who may find some of their suggestions embedded in the dialogue.

The authors were supported by the Charles University grant "Type systems for data-centric programming" (PRIMUS24/SCI/021) and the first author also beneifted from the support of the Charles University-funded research centre "Language, Image, and Gesture: Forms of Discursivity" (UNCE24/SSH/026).

References

1. Alexander, C.: Notes on the synthesis of form. 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), presented at Boxer Salon, Porto
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.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Professional (1975)

[8] A recent seminar focused on the problem of developing new "theories of programming" [26]. However, we believe that a theory needs to start as an informal proto-theory and that *premature* scientific framing in terms of hypotheses may inhibit progress.

7. 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
8. Cohen, S.: Physical context/cultural context: Including it all. OPPOSITIONS **2**, 1–40 (January 1974)
9. 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
10. Depaz, P.: The role of aesthetics in understanding source code. Ph.D. thesis, Université Sorbonne Nouvelle, ED120 - THALIM (2023)
11. 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
12. 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)
13. 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
14. Free Software Foundation: What is free software (version 1.1) (2001), https://www.gnu.org/philosophy/free-sw.en.html, accessed on: 5 February 2024
15. Gabriel, R.P.: Exploratory research proposal: Virtual-world-inspired programming language design
https://www.dreamsongs.com/Files/VirtualWorldsForCodeProposalPure.pdf
16. 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
17. Galison, P.: Image and Logic: A Material Culture of Microphysics. University of Chicago Press (1997)
18. 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)
19. Hacking, I.: Representing and Intervening: Introductory Topics in the Philosophy of Natural Science. Cambridge University Press, Cambridge (1983)
20. 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
21. 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
22. Hossenfelder, S.: Lost in Math: How Beauty Leads Physics Astray. Basic Books (2018)
23. Jacobs, J.: The Death and Life of Great American Cities. Random House (1961)
24. Jakubovic, J.: Achieving Self-Sustainability in Interactive Graphical Programming Systems. Ph.D. thesis, University of Kent (March 2024), https://kar.kent.ac.uk/105537/
25. Jakubovic, J., Edwards, J., Petricek, T.: Technical dimensions of programming systems. The Art, Science, and Engineering of Programming **7**(3), 13–1 (2023)

26. LaToza, T.D., Ko, A., Shepherd, D.C., Sjøberg, D., Xie, B.: Theories of programming (dagstuhl seminar 22231). In: Dagstuhl Reports. vol. 12. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2023)
27. 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
28. Lynch, K.: The Image of the City. MIT Press, Cambridge, MA (1960)
29. MacKenzie, D.: Mechanizing Proof: Computing, Risk, and Trust. The MIT Press (09 2001). https://doi.org/10.7551/mitpress/4529.001.0001
30. Myers, B.A. (ed.): Languages for developing user interfaces. A. K. Peters, Ltd., USA (1992)
31. 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
32. 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
33. Petricek, T.: The rise and fall of extensible programming languages (2023), https://tomasp.net/academic/drafts/extensible-rise/hapoc-2023.pdf, presented at HaPoP
34. 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
35. 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
36. 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)
37. 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
38. Slayton, R.: Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012. MIT Press (2013). https://doi.org/10.7551/mitpress/9234.001.0001
39. Victor, B.: Learnable programming: Designing a programming system for understanding programs (2012), http://worrydream.com/LearnableProgramming