Open substrates for transparent, accessible programming

Tomas Petricek
Charles University, Prague
tomas@tomasp.net

1. Why substrates

I believe that the most important computer science research is not one that answers hard questions, but one that provides a new perspective that lets us ask new more revealing questions. The idea of a programming substrate has the potential to do this for programming.

For a long time, programming has been conceptualized as the act of writing code. This view comes with a number of assumptions. Code is written, compiled into an executable and then used by a user who is distinct from the programmer writing the code. Thinking about programming as interacting with (or operating within) a substrate makes it possible to rethink basic assumptions about programming.

Eliminate user vs. programmer divide. If we think of both programming and using as interaction with a substrate, the strict distinction between a user and a programmer disappears. This is a step towards making programming more accessible and giving users the freedom to use and modify their programs as they wish.

Making programs transparent. If programs exist as structures constructed within a programmable substrate that the user can also manipulate, the user can also look inside the programs they use – in order to understand how programs work, to learn and to modify them.

Focus on interaction. Thinking of programming as interacting with a substrate shifts attention from textual code to other forms of interaction. This makes it possible to think of new, more interactive ways of programming, advancing ideas such as programming by demonstration [3].

Bringing human to focus. When we think of programming as interacting with a substrate, we also need to think about who interacts with the substrate and how. In other words, the view brings the human into the focus, in a fruitful way [6] that the focus on static code does not.

Reconnect with our glorious past. Many bold visions of computing and programming of the past have been lost as the result of the shift to thinking about programming languages [2]. By focusing on programming substrates, we are able to continue – and advance this pioneering work.

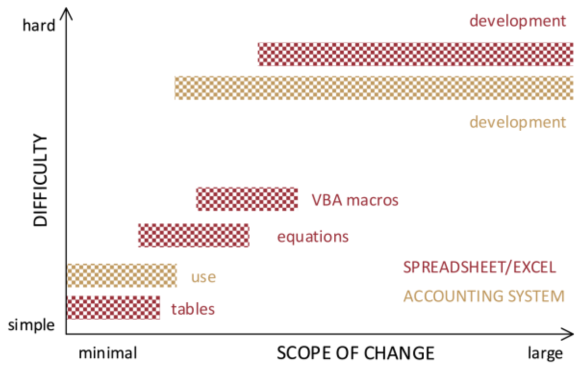

Figure 1. Substrates in current software systems [7]. Most applications today, such as a hypothetical accounting system, offer one substrate for development (code) and one for the user (user interface). Spreadsheets provide multiple substrates with different capabilities (macros provide more capabilities than equations), but the substrates are strictly separate making it difficult to gradually progress.

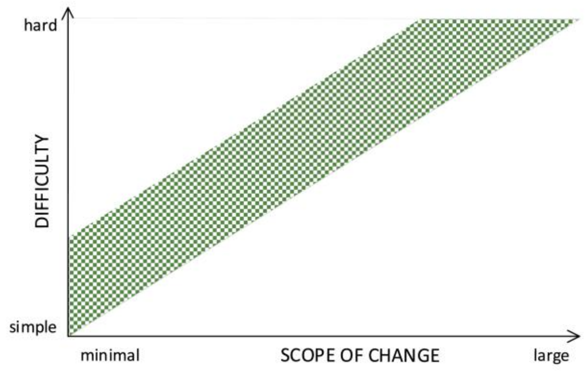

Figure 2. Ideal hypothetical substrate [7]. An ideal hypothetical substrate would provide one uniform way of interaction. This would be easy to use when merely using existing systems, but it would make it possible to make larger changes – modify the system itself – through the same substrate, with difficulty proportional to the scope of the change.

2. Research vision

I believe that software should be more transparent and more accessible. When using a program, users should be able to understand why they are seeing the results that they are seeing. They should be able to ask questions about their outputs and, potentially, modify the software they work with to better suit their needs. The complexity of making a change should be proportional to the scale of the change, i.e., an easy change should be easy to make, but a complex change of behavior may require greater effort. I believe that this can only be achieved if we think about the software substrate on top of which programs are constructed.

However, this kind of research vision requires not just new technical development, but also (a) new research methods that make it possible to study aspects of a system that are hard to formalize using adequately simplified models and (b) better understanding of the past, which reveals why the many past efforts to make programming more accessible and transparent have failed.

2. What is a substrate

In the text above, I used the term substrate without defining it. The examples suggest what I mean by the idea, but finding a precise definition should be one of the tasks for the emerging research field. Tentatively, I understand substrate as a mechanism that can be used for structuring information and computation. A substrate integrates the capability to represent data and logic with the capability of performing computations.

Arguably, this definition is very permissive. A spreadsheet document (with data tables, formulas and live evaluation) is a substrate. UNIX (with files, executables and process invocation) is a substrate. But also, many not yet invented systems can be thought of as substrates. For example, my recent work has been focused on computational documents (structured documents that combine data and formulas with an evaluation mechanism).

The idea of a substrate is very closely related to two other existing notions. The first is personal dynamic media introduced by Kay and Goldberg [5]. Personal dynamic media was envisioned as system for managing user's all information related needs. This is very similar to my understanding of computational substrates, with the difference that computational substrates do not have to be personal – an alternative explored by Webstrates [8]. The second is the notion of interactive, stateful programming systems [1, 2], which also makes it possible to see programming as interaction, rather than as writing of code.

The notion of a computational substrate may let us revisit those past ideas, look at them from a modern perspective and combine them with other novel research directions.

3. Research methods

Perhaps the most valuable aspect of focusing our research efforts around substrates is that it shifts attention – from formal reasoning about code and tools for manipulating code to interaction between the human and the system and capabilities that a programmable system can offer.

However, decades of research on programming that have been focused on code mean that our methods for studying code are significantly more advanced than our methods for studying substrates and interaction. We thus need to complement our research methods with new ones. In this section, I suggest three directions worth exploring.

3.1 Making demos rigorous

Software demos have long been used in the context of live and interactive programming to showcase the capabilities of new research systems. Demos make it possible to show interactive capabilities that are difficult to understand from a textual description. However, demos lack the rigor of mature computer science.

To study substrates, we need to find a way of turning demos from a presentation device into a rigorous epistemic object. To do this, we may take inspiration from the close reading method of critical code studies [10] where a specific aspect of a system is studied in detail from multiple perspectives, as well as from interactive web-based essays that allow the reader to experience particular interaction, albeit in a limited context. The key issue is establishing norms for what aspects of the system need to be explored in depth and what aspects can be ignored as unnecessary technical detail.

3.2 Complementary science

There are multiple past programming systems that are built around interestingly structured substrates. One example is the Boxer system [12] that is built around documents that contain data and computations and follow the design principle of naïve realism (what you see is all there is). An important part of our research on substrates should be recovering those innovative ideas, many of which have been lost or forgotten (or became difficult to understand) due to the paradigm shift from programming systems to programming languages [2].

The notion of complementary science, proposed by the historian Hasok Chang [11] provides a potential direction. It proposes to study history of science, looking at not just the winning but also the losing side of the history and use the recovered knowledge to critically asses contemporary scientific knowledge, further develop forgotten ideas and see if they can offer new perspectives today.

3.3 History of science

Finally, the study of substrates should not be merely technical. For example, the idea of making programming more open has been proposed repeatedly in the past. The Smalltalk vision was of an open system [5], yet the object-oriented programming paradigm that we today remember from Smalltalk does not specifically support openness (you cannot open an object browser and modify your application written in Java or C#).

Similarly, the vision of free software [4] advocated for users to be able to understand and modify their programs. In modern open-source software, this is the reality hypothetically at best – because of the increasing software complexity, most users (including programmers) are practically unable to modify their programs. In both cases, the original vision dissolved for a mix of (perhaps inevitable) social and technical (or engineering) reasons. If we want to design substrates that will offer new capabilities to users, we need to understand the mechanisms through which the desirable capabilities of past systems got lost – both for the sake of understanding history and for the sake of preventing such loss from our future substrates.

4. Conclusions

The notion of a substrate lets us talk about programming in a way that shifts focus from questions focused on formal properties of programming languages to interaction with a system and capabilities provided to the user. This is more human-centric perspective that, arguably, focuses on what actually matters about programming. The idea is not new and has predecessors in work on computational media and programming systems, but reinventing those in the modern research context in the form of computational substrates gives us an opportunity to explore fresh ideas and research directions on programming. An essential part of this new direction will be new research methodologies – we need methods that let us understand interaction with substrates, but also methods that let us understand capabilities of past systems and methods that let us understand how appealing ideas on programming fail.

Acknowledgements. The ideas in this vision statements are heavily influenced by numerous discussions with Joel Jakubovic and Jonathan Edwards and draw from previous work with Stephen Kell and Luke Church.

References

[1] Jakubovic, J., Edwards, J., & Petricek, T. (2023). Technical Dimensions of Programming Systems. The Art, Science, and Engineering of Programming, 7(3), 13-1.

[2] Gabriel, R. P. (2012, October). 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 (pp. 195-214).

[3] Cypher, A., & Halbert, D. C. (Eds.). (1993). Watch what I do: programming by demonstration. MIT Press.

[4] Stallman, R. (1984). The free software definition. GNU's Bulletin, Vol. 1, no. 1: https://www.gnu.org/bulletins/bull1.txt

[5] Kay, A., & Goldberg, A. (1977). Personal dynamic media. Computer, 10(3), 31-41.

[6] Chasins, S. E., Glassman, E. L., & Sunshine, J. (2021). PL and HCI: better together. Communications of the ACM, 64(8), 98-106.

[7] Jakubovic, J., & Petricek, T. (2025) On the Limits of Making Programming Easy. To appear.

[8] Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W., & Beaudouin-Lafon, M. (2015, November). Webstrates: shareable dynamic media. In Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology.

[9] Edwards, J., Kell, S., Petricek, T., & Church, L. (2019). Evaluating programming systems design. In PPIG 2019.

[10] Marino, M. C. (2020). *Critical code studies*. MIT Press.

[11] Chang, H. (2004). Inventing temperature: Measurement and scientific progress. Oxford University Press.

[12] diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859-868.

[13] Petricek, T. & Jakubovic, J. (2021). Complementary science of interactive programming. Presented at HaPoC. Available at: https://tomasp.net/academic/drafts/complementary/