The Hidden Program State Hurts Everyone

Sverrir Thorgeirsson
sverrir.thorgeirsson@inf.ethz.ch
ETH Zürich
Switzerland

Oliver Graf
olgraf@ethz.ch
ETH Zürich
Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zürich
Switzerland

Abstract

While visual scaffolding, live programming, and direct manipulation of the program state are considered useful programming paradigms for novices, they might not always offer the same benefits to experienced software developers. In this essay, we will use chess as a proxy for exploring how these paradigms can also support those who have an intuitive understanding of the program state and its connection with textual code. We will consider the visual programming language Algot and recent user studies conducted on the language to uncover insights into how direct manipulation and programming by demonstration can benefit everyone.

CCS Concepts

• **Human-centered computing** → **Visualization theory, concepts and paradigms**; • **General and reference** → *Experimentation.*

Keywords

visual programming, live programming, programming by demonstration, low-code, no-code, direct manipulation, program comprehension

1 Introduction

Many chess grandmasters are well-known for staring into empty space when playing the game. Like experienced computer programmers who can visualise flows of data without visual aids, strong chess players do not need the crutch of the chessboard when planning their moves. Thanks to their visual memory and visual perception, chess masters can traverse relevant branches of the game tree with ease while keeping track of the entire game state in their mind. This ability allows them to play entire games without looking at the board, and the strongest ones can even play multiple games blindfolded while maintaining a high quality of play. For such players, the mental engagement with an imagined board during blindfold play is merely an extension rather than a deviation of what they do consistently during ordinary gameplay.

Given this profound ability of chess masters to visualise and manipulate the game state mentally, to what extent is the presence of the game board helpful? Does it make any difference at all? The tools that masters use for training and game preparation offer a clue. Professional players, for instance world champion Garry Kasparov [8], typically use sophisticated software such as ChessBase or Chess Assistant,[1] which includes, for example, large databases of tournament games, integrated opening tree views with move popularity and success statistics, engine analysis, and endgame tablebases, with the chess games that are under review represented textually in algebraic notation. An example of an online tool with these capabilities can be see in Figure 4. Such tools are comparable to modern integrated development environments (IDEs) in that the tools used for the analysis and understanding of chess games resemble the ones used by software developers to understand and debug code. There is, however, one key difference: chess preparation software contains a digital representation of the chess board. No matter the skill level, chess players are accustomed to not only constantly seeing the game state, but also to directly manipulate it when exploring new moves or variations. Unfortunately, similar affordances are typically not available to developers, whose primary method of modifying the program state is to edit an opaque textual representation of the state under a dual-coding mindset where one must constantly maintain a mental model of the program's execution alongside the actual code.

Figure 1: Grandmaster Vasyl Ivanchuk playing chess without looking at the board. Photo copyright: David Llada, 2019.

In the essay *Learnable Programming* from 2012, interaction designer Bret Victor states that programming novices need tools that allow them to see the program state in real time [25]. To demonstrate the point, Victor shows how the program context can be

[1] See the products on chessbase.com and chessok.com, respectively.

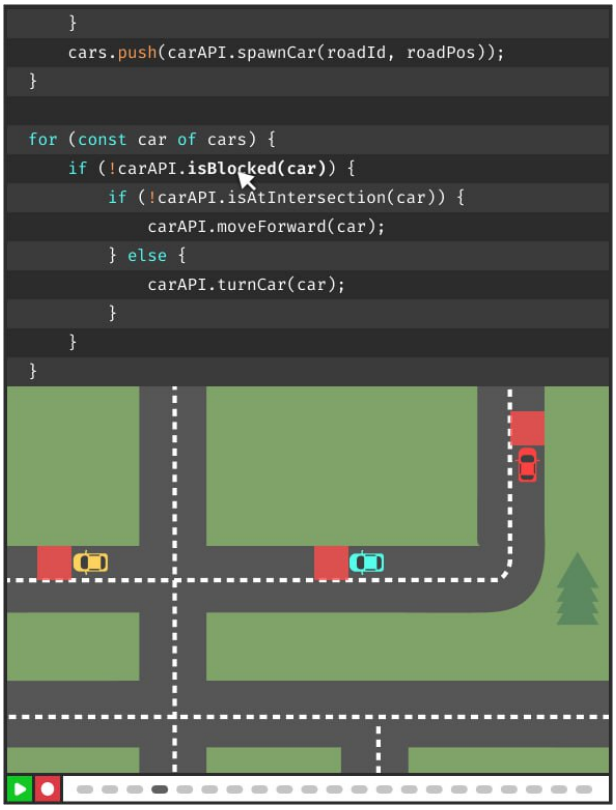

Figure 2: Our rendition of visual scaffolding of the program state of a simple traffic simulation program, with the code on top and the state representation on the bottom. When the programmer hovers over the line `isBlocked(car)`, the tile in front of every car that is being iterated over is highlighted in red, indicating whether it can move forward or not.

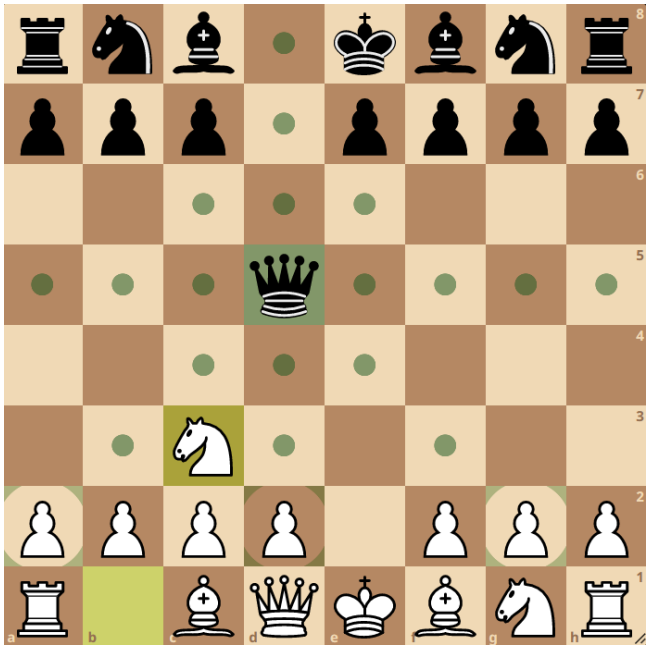

Figure 3: A screenshot from lichess.org showing a scaffolded representation of the chess board. The tiles that the black queen (center) can access are marked by green dots. Experienced chess players may find this presentation distracting.

embedded in the code, for example by highlighting relevant parts of the program state when the matching lines of code are selected. For instance, a model of traffic flow could visually present how vehicles move through a network of roads, with the current speed or positions of vehicles highlighted as the corresponding lines of code are selected or edited (see Figure 2). This approach is one form of scaffolding, which means supporting the learner's development by providing structures to guide their progression. While Victor's essay has influenced various systems in the twelve years since it was published [13, 17, 24], some have found it too limited on the grounds that excessive visual aids would be redundant or even distracting for experts, contributing to higher cognitive load. For example, in a comment on the essay on the *Hacker News* forum, software developer Scott Schneider writes that "chess masters don't need to be shown the legal moves on a board for a bishop; their understanding of the problem is so innate at that point that they no longer play the game in such elementary terms" [18] (see Figure 3). However, we believe that this line of criticism might be too narrow; software meant for chess preparation, medical imaging, financial modelling, and geographic analysis all contain examples of how even experts can benefit from visualisations when carrying out complex cognitive activities, with the implementation details determining their effectiveness.

In this essay, we explore the question: if the visual scaffolding advocated by Victor and others is the equivalent of a naive, digital chess board with distracting piece visualisations suitable only for beginners, what would be the software development equivalent of sophisticated chess software like ChessBase that has transformed the way chess is studied and played? In other words, how can state visualisation and interactivity in software development tools go beyond the needs of novices and elevate the experience of seasoned programmers? To answer the question, we consider Algot [21, 27], a visual programming language inspired by Victor's essay that implements programming by demonstration, direct manipulation, and live programming. Four studies from this year indicate that the language is effective for helping learners at the secondary and tertiary education level when compared against the textual language Python and the block-based visual language Scratch [5, 20, 22, 23]. However, it is less clear whether a language like Algot can meet the demands of professional developers engaged in tasks with a complex program state. We will consider how the principles used in Algot, namely direct manipulation and programming by demonstration, could be extended to support more mature software developers in their work.

2 Background

Don Norman defines the core of interaction design as the bridging of two gulfs: the *Gulf of Evaluation* and the *Gulf of Execution* [14]. The former represents the challenge users face in understanding the

Figure 4: The game analysis interface of the chess website lichess.org. The left part of the screenshot shows a standard, digital visual representation of the game state (the chess position) which can be changed using direct manipulation. The right part (from top to bottom) shows an engine evaluation of the position, an interactive algebraic representation of the moves played in the game, an ECO code that describes the opening played, an integrated opening tree views with move popularity and success statistics, and a short list of top-rated games that that have reached the given position.

outcomes of their interactions with a system, essentially measuring the gap between the system's response to user actions and the user's expectations regarding these actions. The latter refers to a user's ability to enact their intentions within a system, focusing on the difference between the user's goals and the means provided by the system to achieve these goals. For example, in a software environment, the gulf of execution might involve how to manipulate code to achieve their intended outcomes, and in chess, it could mean how to help the user understand why a piece could perform this move, and the effects of that action. Good interaction design needs to bridge both these gulfs and support the user in comprehending them both at the same time.

The current state of computer programming does not achieve this. Currently, the programming process more or less works as follows: the user edits a text document, compiles it to a machine-executable form, observes its execution, analyses the execution for potential flaws and errors, and then edits the document accordingly. This is what we call the debug-compile-execute cycle. While such processes are not unique to programming, they are still onerous and inefficient, and are generally avoided when possible. For example, if one tried to find a recipe for a chocolate cake only by trial-and-error–carefully adding and removing ingredients, adjusting temperatures and mixing times–it might eventually result in something useful, but it would be considered both frustrating and wasteful.

To build something like chess preparation software for computer programming, consider first how it differs from traditional programming environments in terms of visual representation and user interaction. First, chess software excels in using the visual dimension, directly representing the game state through the chess board and piece positions. This way, users can get an immediate and intuitive response as they explore existing games. In contrast, traditional programming environments have historically been less visually oriented. Modern Integrated Development Environments (IDEs) have made strides in this direction, incorporating features like syntax highlighting and real-time syntax checking. These features introduce a form of "liveness," providing immediate feedback to users as they edit code. However, the state itself is seldom visualised. More importantly, chess software allows users to directly interact with the object of interest—the game board, representing the game state—by using it to explore chess moves. This approach to interface development, combining incremental, reversible actions and immediately visible effects, is known as *direct manipulation*, an idea described by Ben Shneiderman in 1993 [2, 19]. In programming, this can be likened to the effect of pointing a garden hose: the programmer guides the changes to their program until they

correspond to the correct solution, as they would guide the stream from the garden hose. Systems that offer sufficiently rapid feedback can give users the impression that they are acting on the objects themselves rather than an intermediary, which in turn lowers cognitive load by reducing the relevant information that the user must keep track of [10].

Direct manipulation is a steep requirement for a programming environment, but some strides have been made by the use of *programming by demonstration* (PbD), a methodology or a paradigm in which the users demonstrate the actions of their programs instead of writing down abstract instructions. PbD is prevalent in robotics, where the user can move the robot by hand or drag its limbs into the correct position using a visual interface instead of by programming the rotation degrees of its joints. However, for general programming, the most significant drawback of this approach is that attempts to generalize a working program based on a demonstration may not always capture the user's intent [9]; in other words, attempts to synthesize a program based on a demonstration can be difficult or impossible due to a lack of information. *Programming by example*, a closely related idea in which the programmer supplies input-output examples, can suffer from the same problem.

Some existing educational tools implement some of these paradigms, at least in part. For example, the block-based programming language Scratch [16] visualises the program (and, to some degree, its state) and does away with the cognitive overhead imposed by language syntax by having the programmer create programs by arranging and lightly modifying blocks of code. Another example from computer science education, Python Tutor [7], is a popular state visualisation tool for Python and other languages that also features a reverse debugger. Like Scratch, it is not a direct manipulation system, but has shown positive results in programming education [12]. Lastly, *AlgoTouch* [1] is a direct-manipulation programming tool which creates programs via the user performing some concrete executions on a visualised program state. While it is also associated with positive learning outcomes, it currently appears to be primarily intended for simple array algorithms.

Outside of education, *low-code* and *no-code* development tools have gained popularity in recent years [9]. These tools are primarily intended for *citizen developers*, which are people who typically possess domain expertise but lack formal experience with computer programming or computer science. These platforms often incorporate visual programming elements and drag-and-drop interfaces, allowing users to create applications with minimal traditional coding. While these tools have found success in certain domains, particularly in business process automation and simple web application development, they still face limitations in terms of flexibility and scalability for more complex software projects.

3 Reflections on Algot

Algot is a visual, graph-based programming language that has been under development since 2019. It was first described in a 2021 research paper [21] and a new version was described at Onward! in 2022 [27]. The research on Algot is ongoing and the language is subject to continuous evaluation in experimental user studies, some of which we will discuss in this section.

Unlike text-based and block-based programming languages, Algot programs are composed without writing or manipulating code. This is achieved by what we consider programming by demonstration; first, one specifies input for a program, and second, one performs actions on this input in the same order that they are meant to be executed when the function is called. During demonstrations, operations can be performed conditionally and repeatedly. Conditionals are realised via queries, which are binary questions that can be asked about the program state (e.g., whether two nodes share the same value). Repeated operations are not realised via iteration but recursion, i.e., by calling the same operation that is currently being defined.

In Algot, the state is represented as a (directed) graph. It can consist of many different connected cyclic or acyclic components, or specific instances of those such as linked lists, trees and individual nodes. Pattern matching can be used to perform operations or queries on nodes belonging to the same connected components as the input nodes. For example, to define a map-like operation f that computes the same operation g on every node in a linked list, one would call g on the head of a list and call f recursively on the child of the list head. When the end of the list is reached and no child is found, the operation will terminate without throwing an error.

Figure 5 shows an example of checking a binary search tree for existence of a value in Algot using pre-defined atomic operations. To do so, the programmer may do the following:

(1) Create a query comparing the root of the BST Root and the input value v. If the values are the same, set the result of the custom query to true.
(2) If Root is greater than v, recursively call BST Search using the left child as the new root, otherwise run it with the right child.

To understand better how this works, the reader is encouraged to test this out for themselves at algot.org.

A similar program in Python would resemble the one given in Listing 1. There are, however, a few differences. First, Algot does not process a return statement, meaning that queries handle computations via side effects only. In custom queries, the boolean return value is implied by treating the query result element like a mutable variable. Second, the Algot program does not need to contain an explicit base case; once the current root does not have children anymore, the recursive call will not have a full set of (existing) arguments anymore and the operation will terminate.

Many computer scientists are skeptical of the notion of allowing state mutations at all, so a programming language that contains no other method of computation might be considered particularly ineffective. While this does make it harder to make mathematical arguments about Algot programs or to prove their correctness, we believe that any cognitive arguments against mutations, namely that they make programs harder to understand, are at least partially addressed by making the state visible throughout the program execution. We also believe that, by avoiding explicit return statements, Algot can eliminate or reduce the likelihood of novices developing certain misconceptions about recursion. For example, students often misunderstand the mechanics of active and passive control in recursive function flow [3, 4], but in Algot, students are unlikely to be confused where a function is supposed to terminate or which

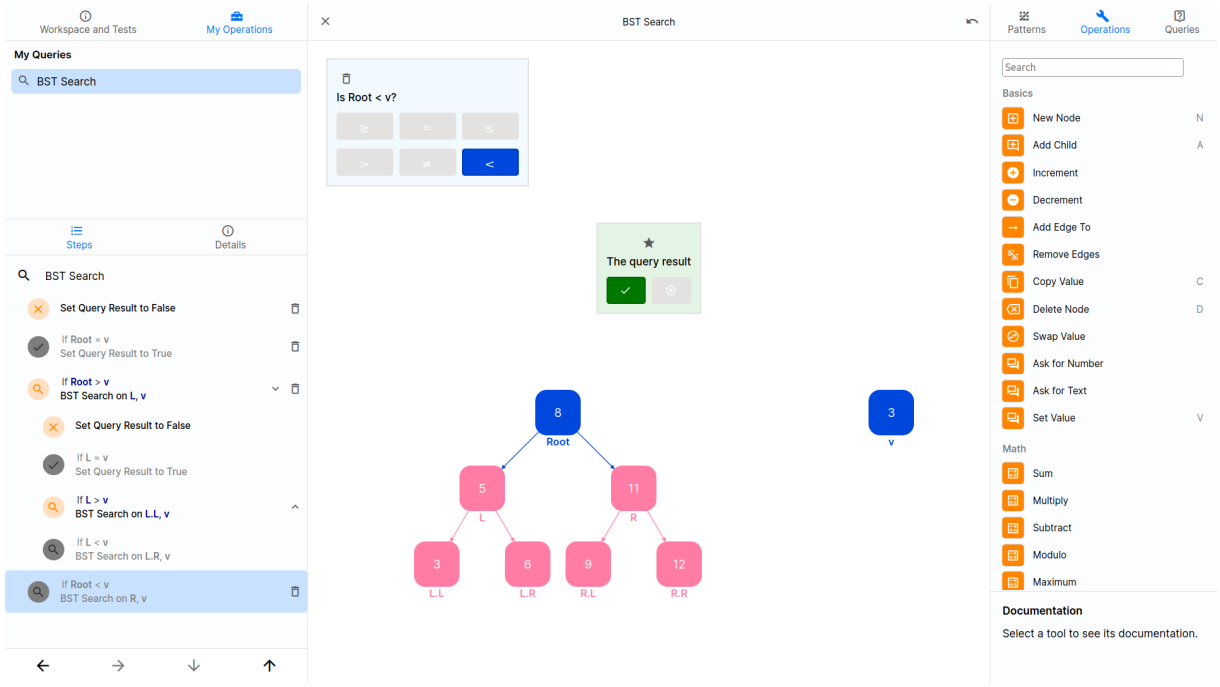

Figure 5: A screenshot of an Algot operation that checks whether the input value v exists in the binary search tree with root Root. Recent additions to the system include support for example-based programming (all nodes have concrete values), an interactive semantic representation of the program (see the sidebar on the bottom left), and a system for users to define their own queries.

variables a function under a recursive call can access. This is because state changes in Algot are observable, explicit, and presented linearly (see bottom left of Figure 5) instead of following the more typical interruption-based mechanism of intermittent return values in the base case and the recursive calls.

```
def bst_search(root, v):
    if root is None:
        return False
    if root.value == v:
        return True
    if v < root.value:
        return bst_search(root.left, v)
    return bst_search(root.right, v)
```

Listing 1: A Python implementation of a function which checks if a value exists in a given binary search tree.

Four controlled, experimental studies on Algot published this year support the notion that this approach is helpful for novices [5, 20, 22, 23]. For instance, in one study, 23 secondary-school students were taught recursion in Algot (the experimental group) and Scratch (the control group), a block-based programming language that most students were familiar with. They were then asked to compute certain recursive functions and to take a post-test measuring their conceptual and procedural understanding of recursion. Despite the low number of participants, there was strong and statistically significant evidence that students using Algot did better on the assigned tasks and the post-test [20]. Our hypothesis, based on the textual feedback from the students, is that the difference in results has mostly to do with students being able to visualise the state when programming, but the other properties of the language may also be a contributing factor.

However, it is less clear to us how a programming language based on direct manipulation and programming by demonstration could support more experienced programmers. For instance, the program in Listing 1 is trivial for an expert to implement. It is not obvious why someone would prefer to do so in Algot, having to do multiple tiring mouse clicks in the process instead of what could just as well be implemented in a Python one-liner. This is aligned with our experience that experienced developers are less accepting and less patient of the language than beginners, which may explain why they are no less likely to develop conceptual misconceptions about its mechanisms. To show experts the usefulness of the paradigms it implements, namely direct manipulation, live programming, and programming by demonstration, we have built small extensions to the language that may make it more relatable to those accustomed to textual programming and may improve on its expressiveness to make it easier to build more complex applications.

First, we note that there is nothing about the programming paradigms we used that precludes us from introducing a return statement. In fact, we have taken a step towards this direction with the introduction of a custom queries. Expanding on the original query system introduced at Onward! in 2022 [27], our extension support users in defining their own queries using the same demonstration mechanism that is used for operations. Figure 5 shows

an example of how the system can be used recursively to define a query that determines if a binary search tree contains a given value. The center of the screen shows a small window labelled "the query result" which represents the boolean return value of the query. The value, which is `false` by default, can be changed by direct manipulation of the two buttons inside of the window. This action is interpreted as any other Algot operation and can therefore be conditionalised; for instance, we may wish to change it to `true` if the input integer is zero, and otherwise decrease the value by two and then call the query recursively if the value is still positive. The code could be represented as the Python program shown in Listing 2, using the base query `compareNumbers`.[2]

```
queryResult = False
def BSTSearch(root, v):
    global queryResult
    if root == v:
        queryResult = True
    if root > v:
        BSTSearch(root.left, v)
    if root < v:
        BSTSearch(root.right, v)
```

Listing 2: An alternative implementation in Python of searching a binary search tree.

Here, `queryResult` is the implicit return value of function. The code is hence functionally equivalent to the more common and idiomatic approach shown in Listing 1.

From a cognitive standpoint, we think that one meaningful distinction between these two approaches is that the former one shows explicitly what the function is returning (the variable `queryResult`), allowing the programmer to follow how this value changes throughout the definition of the program, and to change it any time using direct manipulation. Note that while `queryResult` is a global variable that can be modified within each recursive call, it is inaccessible outside the query definition. We think that this is a balanced way to introduce higher state awareness without adding unnecessary visual distractions. This method could be further extended in Algot by allowing the programmer to set nodes or connected components as the global return value inside a query-value window equivalent, or by marking specific nodes as designated return values.

A second change that we have made is the introduction of example-based programming, which was first mentioned as future work in a 2022 paper on Algot [27]. As shown in the example in Figure 5, every node has a specific, concrete value that is chosen by the programmer when a new operation is first defined. When needed, additional sets of example values can be defined at any point during the operation composition. We find that this is a non-intrusive method of helping beginners and experts visualise state changes throughout the program execution. This addition should also help the programmer assess the correctness of the programs by borrowing same principle as test-driven development, namely to write tests prior to defining functions.

Third, we have introduced a proper program editor, also visible in Figure 5 on the bottom left. We agree with Victor that the programmer should be able to "follow the program execution over time" as opposed to "only seeing a single point in time at any instant" [25], so by using the arrow buttons, it is possible to view the program state after any given operation was executed. Furthermore, it is possible to edit programs by rearranging operations via drag-and-drop or by stepping into function calls like in visual debuggers. We did find, however, that the visual scaffolding can become excessive when trying to intertwine the visual and textual representation. For example, we tested highlighting the relevant nodes and operation when the programmer hovers over the matching syntax (e.g., in the example in Figure 5, to highlight the nodes `d`, `f` and `c` and the operation `Dot Product` in the right sidebar when hovering over the fifth syntactic element). We found this distracting and likely to contribute to the programmer's extraneous cognitive load, similarly to how the visual scaffolding of chess piece movement (see Figure 3).

Last, we have begun implementing more advanced data representation in Algot to facilitate development at scale, starting with explaining data in context. Algot nodes can contain two types of data: numbers and strings. The existing base operations are type-aware; for example, when `increment` has been selected, nodes with a string value fade out of view, becoming unselectable. We consider this to be what Repenning might call a "pragmatic affordance" of a visual programming language [15]. Our intention is to take this one step further by supporting abstract data types in which a single node can represent any arbitrary connected component together with its own operations, and the developer can at any time switch between an abstract, high-level view of the data or drill down to concrete details when necessary. For instance, an Algot list could contain nodes representing hospital patients, each containing hierarchical data in a tree structure such as medical history, current treatments, and upcoming appointments, each of which could in turn represent a more complex data structure in its own node encapsulation. This way, developers could work with large, complex data, as opposed to toy examples, while still working with direct manipulation, live programming, and programming by demonstration.

4 What does this all mean?

In the previous section, we reflected on our experience and vision of how Algot implements the kind of visual scaffolding and state awareness that we think developers at every skill level could benefit from. The question remains how all of this could be brought together into a cohesive system.

Graphs were chosen as the underlying data structure of Algot due to their versatility and modelling power [27]. For example, a chess board could be expressed as a graph with 64 vertices and edges representing adjacent squares or possible piece movements. The simpler example in Figure 6 shows how a classical chess problem could be easier to solve and understand with a graph representation. We note that while all the examples that we have shown in this essay have represented graphs in the same textbook style with circular nodes connected by linear edges, there is nothing stopping us from letting the programmer programmatically customize or transform the visual representation of graphs according to their specific needs or preferences.

[2]For more information on how custom queries can be be implemented, see a video demonstration here of a simple Algot program: https://algot.org/AlgotIsEven.mp4.

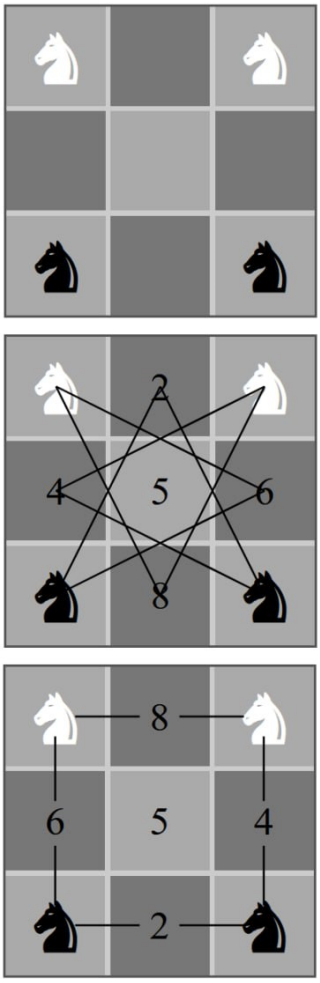

Figure 6: Top: Guarini's problem (first published in 1512), which is the earliest chessboard puzzle in the world [26]. The goal is to find a sequence of moves such that the black and white knights switch positions. Middle: The chess board represented as a graph where the edges indicate the legal moves of each knight. Bottom: An untangled copy of the same graph, demonstrating how easy it is to solve the problem after a different state representation has been found.

We believe that with the proper auxiliary tools and extensions, some of which we discussed in the last section, Algot or a language built on the same principles can serve as a general-purpose language that aids developers in the same sense that domain-specific visual systems, such as the ones we discussed in the introduction, can help specialists with narrow subjects. One line of criticism of Victor's *Learnable Programming* essay is that it is "problem-specific" and that "[t]he control flow visualisation works great for toy problems when learning programming, but quickly breaks down in the real world [...]. You're working with billions of seemingly random integers, or with strings, or even more complex data structures. How are you going to visualize that over time?" [11]. But we believe that by using graphs with nodes of nested, abstract data types, we can represent virtually anything, from the elementary algorithms we showed in the previous section, to the simple geometric shapes shown in Victor's essay (given the right visual transformation tools), to complex business applications with large amounts of data. While the entire state may not always be visualized at once, we propose that the state graph can be expanded or collapsed as needed, providing a scalable visualization that can adapt to the complexity and size of the data involved.

One struggle with getting developers to adopt such systems is path dependency. For example, Paul Graham has pointed out with what he called the "Blub Paradox" that programmers often judge the power of a programming language by what they already know, rather than its objective merits or potential for growth; a hypothetical "Blub" programmer is content with programming in Blub, thinks in Blub, and finds that other languages are weird and full of "other hairy stuff" [6]. The purpose of this essay is not just to share our vision of software development or to advocate for a specific visual programming language, but to call on others to move beyond "Blub" and participate in bringing to life the vision of scalable, general-purpose direct manipulation systems in which programmers can direct their faculties to other, more intellectually meaningful tasks than constantly maintaining a mental representation of the program state. This may require a shift in thinking about what programming entails and what that means is largely unknown to us, and we encourage those interested in exploring that vision to get in touch.

5 Acknowledgements

We thank Sandra Wiklander for helping create and find photos for this essay and David Llada for the permission to use the photo in Figure 1 (link: https://www.flickr.com/photos/davidllada/47517860951).

References

[1] Michel Adam, Moncef Daoud, and Patrice Frison. 2019. Direct manipulation versus text-based programming: An experiment report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 353–359. https://doi.org/10.1145/3304221.3319738

[2] David M Frohlich. 1993. The history and future of direct manipulation. *Behaviour & Information Technology* 12, 6 (1993), 315–329. https://doi.org/10.1080/01449299308924396

[3] Carlisle Eldwidge George. 1996. *Investigating the effectiveness of a software-reinforced approach to understanding recursion.* Ph. D. Dissertation. Goldsmiths College (University of London).

[4] Carlisle E George. 2000. EROSI—visualising recursion and discovering new errors. *ACM SIGCSE Bulletin* 32, 1 (2000), 305–309. https://doi.org/10.1145/331795.331875

[5] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. 2024. Assessing Live Programming for Program Comprehension. In *Proceedings of the 29th Innovation and Technology in Computer Science Education Conference (ITiCSE 2024)*. ACM, Milan, Italy. https://doi.org/10.1145/3649217.3653547

[6] Paul Graham. 2005. Beating the Averages. https://paulgraham.com/avg.html. Accessed: 2024-04-15.

[7] Philip J. Guo. 2013. Online Python tutor: embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[8] John Hartmann. 2008. Garry Kasparov Is a cyborg, or What ChessBase Teaches Us about Technology. (2008).

[9] Martin Hirzel. 2023. Low-Code Programming Models. *Commun. ACM* 66, 10 (sep 2023), 76–85. https://doi.org/10.1145/3587691

[10] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct manipulation interfaces. *Human–computer interaction* 1, 4 (1985), 311–338.

[11] Jules Jacobs. 2012. Comment on Bret Victor: Learnable Programming. https://news.ycombinator.com/item?id=4578339. Accessed: 2024-04-16.

[12] Oscar Karnalim and Mewati Ayub. 2017. The effectiveness of a program visualization tool on introductory programming: A case study with PythonTutor.

CommIT (Communication and Information Technology) Journal 11, 2 (2017), 67–76. https://doi.org/10.21512/commit.v11i2.3704

[13] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2481–2490. https://doi.org/10.1145/2556288.2557409

[14] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[15] Alexander Repenning. 2017. Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets. *J. Vis. Lang. Sentient Syst.* 3, 1 (2017), 68–91. https://doi.org/10.18293/VLSS2017-010

[16] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[17] Charles Roberts, Matthew Wright, and JoAnn Kuchera-Morin. 2015. Beyond editing: extended interaction with textual code fragments. In *NIME*. 126–131. https://doi.org/10.5555/2993778.2993812

[18] Scott Schneider. 2012. Comment on Bret Victor: Learnable Programming. https://news.ycombinator.com/item?id=4577609. Accessed: 2023-04-16.

[19] Ben Shneiderman. 1982. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology* 1, 3 (1982), 237–256.

[20] Sverrir Thorgeirsson, Lennart Lais, Theo Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon. https://doi.org/10.1145/3626252.3630916

[21] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576166

[22] Sverrir Thorgeirsson, Theo Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon. https://doi.org/10.1145/3626252.3630808

[23] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research. In press. (ICER '24)*. ACM, Melbourne, VIC, Australia. https://doi.org/10.1145/3632620.3671124

[24] Jasper Tran O'Leary, Gabrielle Benabdallah, and Nadya Peek. 2023. Imprimer: Computational Notebooks for CNC Milling. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–15. https://doi.org/doi.org/10.1145/3544548.3581334

[25] Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. *URL: http://worrydream. com/LearnableProgramming* (2012).

[26] John J Watkins. 2004. *Across the board: the mathematics of chessboard problems*. Princeton University Press.

[27] Theo B Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 80–94. https://doi.org/10.1145/3563835.3567668