Toward a Theory of Programming Language and Reasoning Assistant Design: Minimizing Cognitive Load

MICHAEL COBLENZ, University of Maryland, USA

Current approaches to making programming languages and reasoning assistants more effective for people focus on leveraging feedback from users and on evaluating the success of particular techniques. These approaches, although helpful, may not result in systems that are as usable as possible, and may not lead to general design principles. This paper advocates for leveraging theories from cognitive science, focusing on cognitive load theory, to design more effective programming languages and reasoning assistants. Development of these theories may enable designers to create more effective programming languages and reasoning assistants at lower cost.

1 INTRODUCTION

How can we design programming languages and reasoning assistants that make users as effective as possible? User-centered design techniques, which have recently been applied to programming languages [5], enable designers to iteratively improve their designs using feedback from users. Alternatively, when a designer has a design candidate that they believe may be effective, the designer can evaluate the tradeoffs in a user study. However, there are two problems with these approaches. First, perhaps the designs are beneficial relative to other design candidates, but alternative designs might be superior. That is, perhaps the process finds only a *local maximum* in the design space. Second, the approach does not provide *predictive power* regarding candidate designs, and in a search for effective designs, it is not cost-effective to evaluate large numbers of design candidates. The results of evaluating individual systems may not generalize to new systems that combine techniques from earlier evaluated designs because of potential interactions among design components. Of course, in particular situations, the designer might *hypothesize* that prior results *do* apply on the basis of the designer's experience or on the basis of various kinds of theory.

An alternative approach to design languages that are more effective for users would be to use a theory of usability of languages. If we had theories that provided predictive power regarding programming language and reasoning assistant (PL/RA) designs, designers might be able to create designs that are more effective overall, avoiding the *local maximum* challenges that arise with user-centered design iteration and the *high costs* of repeated evaluations with users. Even if a theory does not provide quantitative predictions, it may help a designer hypothesize which designs are likely to be more effective than others, guiding the design process at lower cost than an empirically-driven approach.

To illustrate the benefits of a user-centered theory-driven approach to PL/RA design, in this paper, I consider the relevance of one particular theory (*cognitive load theory*, CLT) to programming language designs. Many of these implications pertain to type systems or the design of reasoning assistants directly, though for the sake of completeness, I consider other implications as well. In the future, we might turn our efforts to developing *new* theories, potentially in collaboration with cognitive scientists. By showing how one theory from cognitive science may have implications on PL/RA design, I aim to promote the use and further development of human-related theory in PL/RA

Author's address: Michael Coblenz, mcoblenz@umd.edu, University of Maryland, 8125 Paint Branch Drive, College Park, Maryland, USA, 20742.

design. These theories complement traditional mathematical theories of programming languages, which should be used simultaneously. By coupling an analysis of the cognitive load characteristics of a feature and participants' task performance in user studies, it may be possible to develop a predictive model of how designs affect language learnability and programming task performance.

2 INTRODUCTION TO COGNITIVE LOAD THEORY

Cognitive scientists have long modeled human memory as including a *short-term memory* component, which temporarily stores facts that are relevant to one's current task [10]. Unfortunately, the capacity of short-term memory is limited, with researchers finding limits between 4 and 7 items in adults [7]. In addition, items in short-term memory can be discarded as a result of a change in attention or due to the passage of time. Cognitive load theory (CLT) was developed by Sweller [22] in a context of studying learning. CLT models learning challenge in terms of the number of items that one must remember at once in order to accomplish the learning objective. If the learning process requires remembering too many items, then performance suffers in terms of time required, errors made, or both. CLT applies to problem-solving tasks, particularly ones conducted in the context of learning [21].

How can we write software effectively, considering the complexity of software (even a single function or method likely has more than 4 lines of code)? There are three techniques that allow us to work around the limits of short-term memory. First, some relevant facts can be located in *long-term memory*. Although adding items to long-term memory requires rehearsal, long-term memory has an extremely large capacity. Second, by combining disparate facts into one entity ("chunking"), the facts can share one memory "slot." For example, perhaps beacons [24] reflect a way to chunk lines or tokens so that they require less short-term memory. Finally, one can externalize information by recording information in another form, such as in a text file or on paper. Of course, paging this externalized information back in has a time cost, and may be neglected entirely. Failure to restore externalized information could be a source of bugs, as important requirements or facts are omitted from consideration.

When cognitive load is high, CLT predicts lower learning efficiency [21]. High cognitive load is also associated with high error rates [1]. Even in small lab studies with 30-40 lines of code, cognitive load has been found to be high when certain antipatterns are followed in identifier selection [9]. Existing methods (also described by Fakhoury et al. [9]) can be used to measure cognitive load, allowing researchers to investigate the effect of PL/RA designs on cognitive load in relevant tasks.

CLT also predicts an effect called *expertise reversal* [14] in which providing cognitive support directed at novices can actually reduce experts' performance on the same tasks. The idea is that experts may experience unnecessary cognitive load when processing beginner-level advice that they do not need. This might point toward a need for customized error messages or other tools.

3 IMPACT OF DESIGNS ON COGNITIVE LOAD

Programming languages, environments, and proof assistants each embody design choices that impact their users' cognitive load. Key techniques for reducing cognitive load include reducing the amount of information needed for a task (as in features that promote modularity); co-locating relevant information (typically provided by IDE features such as *jump to definition*; and visualizations.

3.1 Programming Languages

What programming language design decisions might influence short-term memory requirements? Any programming subtask that does not directly relate to one of the high-level requirements may consume short-term memory slots. Below is a partial list of programming tasks that may unnecessarily occupy short-term memory.

Determining reference validity When programming in a language in which references can be invalid, reference validity may need to be checked. If neglected, this can result in undefined behavior (in unsafe languages) or runtime errors.

Leveraging imprecise static type information Types constrain the operations that are permitted on an expression. When using an unfamiliar type, a programmer might read declarations or documentation to determine what operations are available, so these operations are in short-term memory, not long-term memory. As a result, doing a sequence of operations with the type might require re-reading the documentation. IDEs can help via autocomplete tools, but these tools are less effective when an expression has an imprecise type, since less information is available regarding what operations are available.

In addition, autocomplete and autosuggest features can be made much more precise in the presence of precise static type information; these features prevent users from needing to remember the full names of uncommonly used methods and functions.

Handling errors Handling errors requires that the programmer consider additional questions that do not pertain to the programmer's primary goal, such as: what might have caused the error? Is the error recoverable? What should be done about it? C programmers are expected to check return values for exit codes, but many of the errors are not recoverable (for example, there is typically no solution if `malloc` returns `NULL`). Java programmers need to handle exceptions, but frequently do not bother to write appropriate handlers [15]. Future research should investigate whether writing error handling code is problematic from a cognitive load perspective and consider whether separating handling errors out as a separate development task might reduce cognitive load.

Implicit operations Should a language allow implicit casts? If so, the semantics of operations may be unclear; if not, the need to insert explicit casts may increase cognitive load (since the programmer must think about the choices of which types to cast to and from). Some languages, such as Scala and Agda, allow implicit arguments. This may reduce cognitive load if the arguments are not relevant to the task, but it may increase it if the programmer needs to figure out how the implicit arguments are inferred.

Reasoning about code re-use Changing one function or module may break dependent modules. Therefore, when making a change, the programmer should be aware of any requirements imposed by clients of the changed code. This requires that the programmer add the dependencies to short-term memory; keeping all of these in mind simultaneously could easily exceed the size limit. One might argue for clear module boundaries and complete requirement specifications, but specifications can be redundant with parts of the implementation (must one re-state preconditions that are expressed in assertions?), and specifications can themselves be inconsistent with the code.

PL designers typically provide features for *modularity* to make it easier to re-use code in different contexts and to protect clients of code from changes in other parts of the codebase. Although this separation of concerns can reduce cognitive load, it can come at a cost: more abstract code can result in more abstract or less-localized error messages, which can be harder to understand. In addition, leveraging modularity features results in more architectural complexity, which can itself increase cognitive load for clients. That is, there can be a tradeoff between load for clients and load for API authors. One approach could be to use program optimization or partial evaluation [2, 11] to reduce information needs for authors of clients. [1].

---

[1]Thanks to reviewer 1 for this observation.

Inheritance is one language feature that results in hidden dependencies; the author of a superclass can be unaware of subclasses (the fragile base class problem [16]). Type systems are a key tool designers can use to promote and inhibit code re-use, in part through subtyping. Future work might investigate how subtyping impacts cognitive load.

Managing effects Designers of pure languages, such as Haskell, often tout the cognitive benefits of avoiding side effects. From a cognitive load perspective, side effects may demand that the programmer keep the possible side effects in short-term memory when calling a function rather than being able to reason only based on the inputs and outputs. If the effects pertain to a module that is not in short-term memory, the programmer may need to load that module into short-term memory, potentially flushing other needed information.

However, pure languages may also impose costs on short-term memory. Programmers may need to track additional state explicitly (e.g., with variables that represent the current environment), requiring additional slots of short-term memory. If the state could have been externalized in a way that does not require programmer attention, a stateful solution may well be better from a cognitive load perspective. By providing a channel of communication among parts of a program that is independent of control flow, state can reduce the need for state transmission to occupy short-term memory slots.

Type systems that capture effects may make the problem even worse by forcing programmers to attend to effects that are irrelevant to their concerns. For example, they may not care whether the function logs to the console, but one cannot do so without specifying an appropriate effect, which may increase cognitive load. On the other hand, effect systems could reduce cognitive load by preventing the programmer from having to consider all possible effects. Future research should evaluate the cognitive load impact of particular effect system designs.

Fulfilling proof obligations When using some verification tools, users may need to use ghost variables and other techniques to fulfill proof obligations. These variables increase cognitive load relative to approaches that do not require them (including ones that do not use verification).

3.2 Programming Environments

Requirements and types. Software components typically need to meet collections of requirements. Functions or methods may have requirements regarding security, performance, invariant maintenance, error checking, error handling, and of course functional correctness. Even this list of six categories may be larger than the short-term memory size! It is not surprising that many functions do not, in fact, meet all of their requirements without extensive testing. Perhaps the process of writing tests helps programmers divide their work in a way that lets them focus on just one or two requirements at a time — a number that *does* fit in short-term memory. Then, they can fix their implementations by using an external requirement store: the set of tests.

Requirements traceability refers to the ability to relate requirements (as typically written in a natural language requirements document) to code or other artifacts that implement the requirements [13]. This is useful in software engineering so that software engineers can answer questions about the purposes of particular sections of code. What if a programming environment provided explicit requirements traceability functionality? The IDE could track pending requirements for each module, and a programmer could annotate components when fulfilling those requirements. Even without any kind of verification process to mechanically ensure the requirements have been met, the mechanism could still serve to help people keep track of which requirements they needed to meet and which they had attended to so far.

Some IDEs provide limited support for tracking unmet requirements by generating warnings that correspond to comments that start with TODO. But this requires that programmers recall unmet requirements while writing code. Instead, suppose programmers first listed requirements, and then worked toward fulfilling them (an approach often recommended by instructors of beginning programming courses!). The process is similar to that proposed by test-driven design or test-first design, but lighter weight because it does not require writing actual test cases first. Instead, prose requirements would suffice, and the IDE could help the programmer track which ones were still to be completed. New maintainers could re-check the revised code against the original list of requirements, and the IDE could help by providing a checklist to examine.

How would the IDE connect code to requirements? Verification approaches, such as dependent types, are one approach, but these carry a significant burden for users. Another approach, when formal verification is not cost-effective, is for the code to carry metadata that maps between natural language requirements and lines of code.

Dependency management. Many languages require that code both import required modules in code (e.g. Java import) as well as specify version of libraries that the program depends on (which the build tool should fetch before building, e.g Maven dependencies). When adding a use of a new library, the programmer must set aside the current task, manage the dependencies, and then return to the current task — at which time the programmer must remember what was in their short-term memory. Could programming environments help queue this kind of work to avoid overloading the programmer's short-term memory? To what extent must these tools be workflow-specific — or can one class of management tool apply in many different kinds of situations?

Information access. When working on programming tasks, programmers frequently need a variety of different kinds of information: documentation, code examples, notes, and of course the code being edited. Jump-to-definition features help programmers easily access relevant information for some tasks. Some systems, such as Code Bubbles [4], help users arrange some of these items on the screen at once. This mitigates short-term memory limitations, which prevent programmers from keeping all the relevant information in their minds. However, IDEs typically do not provide assistance keeping track of relevant facts *automatically* other than with a back/forward button. Can IDEs automatically detect which relevant pieces of information are unlikely to fit in short-term memory and provide appropriate assistance?

Visualizations. Although architectural diagramming tools exist, automatic visualization tools frequently generate diagrams that are too densely-populated for practical purposes. Programmers may need diagrams that include information relevant to their current tasks, but many visualization tools incorporate only limited task information when they generate the visualizations. Visual representations matter in problem solving, however. For example, Zhang [25] showed how the representation of Tic-Tac-Toe can make it seem like a completely different game.

3.3 Proof Assistants

Interactive proof assistants, such as Coq [23], typically track the pending proof obligations that the user is required to fulfill. This obviates the programmer from needing to track these in short-term memory. They also track the set of theorems that have already been proven. The latter presents a challenge, however, since in working toward a particular proof obligation, the user must either repeatedly scan the list of available theorems, or commit them to their (small) short-term memory. Could proof assistants *chunk* the available theorems in order to help the user fit more of them in short-term memory? For example, could they *combine* them where possible to create a smaller number of more-powerful theorems?

Tactics-based theorem provers require users to learn a set of available tactics that can be applied to transform proof terms. Experts have large sets of tactics available in their long-term memories;

novices have learned much smaller sets. But in many cases, the best tactic to apply may not be available in long-term memory and also not have been used recently. Perhaps proof assistants could *suggest* tactics to be applied. Unlike automatic proof search, the idea would be to suggest specific tactics that might be useful, rather than applying a sequence of them automatically. If the suggestions are relevant enough of the time, this approach could help users learn tactics that they could use in the future. Unlike traditional textual autocomplete in IDEs, this form of autosuggest would not be based on lexical prefixes, but instead on proof obligations and available proof terms.

4 RELATED WORK

Crichton et al. [8] studied the role of short-term memory in code tracing tasks, finding a limit of about 7 variable/value pairs and proposing changes that could be made to programming tools to help with code tracing tasks. Reducing cognitive load has been proposed in other HCI contexts, such as multimedia interfaces [19]. Shaffer et al. proposed techniques that might reduce cognitive load in introductory programming courses, such as partitioning information into small segments [20].

Helgesson [12] studied grounded theory analysis of interviews and literature regarding cognitive load in software engineering, finding several categories of contributions to cognitive load: task, environment, information, tool, communication, interruption, structure and temporal. This paper focuses on the *tool* perspective.

Programming with holes [17] and active code completion [18] are both approaches that might help programmers maintain focus on their tasks. Programming with holes allows programmers to defer work for later, leaving type information to direct their follow-up work. Active code completion allows domain-specific user interfaces to guide the user through use of a specialized API, preventing the user from needing to consume short-term memory slots with API details. Likewise, Agda [3] has an emacs mode that allows the user to insert typed holes for completion later.

Code Bubbles [4] and JASPER [6] help programmers collect and reference relevant pieces of information while working in an IDE. Both tools allow programmers to arrange relevant information, such as code, on the screen while working on tasks. These techniques may help reduce programmers' reliance on short-term memory.

5 FUTURE WORK

The list of design decisions that may impact cognitive load is surely incomplete; a more exhaustive list (and a method for enumerating it) would be preferable. Future work should investigate whether the design decisions described above indeed relate to cognitive load, whether the cognitive load leads to a usability cost, and if so, how the cognitive load might be decreased. Furthermore, cognitive load theory is only one of many existing theories from cognitive science; others, such as the concept of *attention*, should be investigated as well. Finally, the theoretical frameworks from cognitive science are unlikely to suffice to justify every PL/RA design decision, so the community should work toward richer theories that provide more explanatory power regarding programmer behavior.

6 CONCLUSION

Existing theories from cognitive science, such as cognitive load theory, may offer guidance on a variety of programming language design questions. Many of those questions have been the subject of debate or empirical evaluation, but having a theory (particularly one with sufficient predictive power) could lead to a more effective design strategy for programming languages and reasoning assistants. This paper proposes exploration of the relationship between the identified language

design decisions, cognitive load, and performance, with the hope of assessing whether cognitive load theory can be leveraged in programming language design.

ACKNOWLEDGMENTS

REFERENCES

[1] Paul L. Ayres. 2001. Systematic Mathematical Errors and Cognitive Load. *Contemporary Educational Psychology* 26, 2 (2001), 227–248. https://doi.org/10.1006/ceps.2000.1051

[2] Sandrine Blazy and Philippe Facon. 1998. Partial Evaluation for Program Comprehension. *Comput. Surveys* 30, 3es (1998), 17. https://doi.org/10.1145/289121.289138

[3] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[4] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 455–464. https://doi.org/10.1145/1806799.1806866

[5] Michael Coblenz, Ariel Davis, Megan Hofmann, Vivian Huang, Siyue Jin, Max Krieger, Kyle Liang, Brian Wei, Mengchen Sam Yong, and Jonathan Aldrich. 2020. User-Centered Programming Language Design: A Course-Based Case Study. In *HATRA*. arXiv:2011.07565 [cs.PL]

[6] Michael J. Coblenz, Amy J. Ko, and Brad A. Myers. 2006. JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange* (Portland, Oregon, USA) *(eclipse '06)*. Association for Computing Machinery, New York, NY, USA, 65–69. https://doi.org/10.1145/1188835.1188849

[7] Nelson Cowan. 2010. The magical mystery four: How is working memory capacity limited, and why? *Current directions in psychological science* 19, 1 (2010), 51–57.

[8] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. *The Role of Working Memory in Program Tracing*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445257

[9] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 286–28610.

[10] Murray Glanzer and Anita R. Cunitz. 1966. Two storage mechanisms in free recall. *Journal of Verbal Learning and Verbal Behavior* 5, 4 (1966), 351–360. https://doi.org/10.1016/S0022-5371(66)80044-0

[11] M. Harman and S. Danicic. 1997. Amorphous program slicing. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*. 70–79. https://doi.org/10.1109/WPC.1997.601266

[12] Daniel Helgesson and Per Runeson. 2021. Towards Grounded Theory Perspectives of Cognitive Load in Software Engineering. https://www.ppig.org/papers/2021-ppig-32nd-helgesson/

[13] IEEE. 2010. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (2010), 1–418. https://doi.org/10.1109/IEEESTD.2010.5733835

[14] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. 2003. The Expertise Reversal Effect. *Educational Psychologist* 38, 1 (2003), 23–31. https://doi.org/10.1207/S15326985EP3801_4

[15] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, 484–487. https://doi.org/10.1145/2901739.2903497

[16] Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *European Conference on Object-Oriented Programming (ECOOP 1998)*. 355–382.

[17] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290327

[18] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. 859–869. https://doi.org/10.1109/ICSE.2012.6227133

[19] Sharon Oviatt. 2006. Human-Centered Design Meets Cognitive Load Theory: Designing Interfaces That Help People Think. In *Proceedings of the 14th ACM International Conference on Multimedia* (Santa Barbara, CA, USA) *(MM '06)*. Association for Computing Machinery, New York, NY, USA, 871–880. https://doi.org/10.1145/1180639.1180831

[20] Dale Shaffer, Wendy Doube, and Juhani Tuovinen. 2003. Applying Cognitive load theory to computer science education. In *PPIG*, Vol. 1. 333–346.

[21] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[22] John Sweller and Paul Chandler. 1994. Why some material is difficult to learn. *Cognition and instruction* 12, 3 (1994), 185–233.

[23] The Coq Development Team. 2021. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.4501022

[24] Susan Wiedenbeck. 1986. Beacons in computer program comprehension. *International Journal of Man-Machine Studies* 25, 6 (1986), 697–709. https://doi.org/10.1016/S0020-7373(86)80083-9

[25] Jiajie Zhang. 1997. The nature of external representations in problem solving. *Cognitive Science* 21, 2 (1997), 179–217. https://doi.org/10.1016/S0364-0213(99)80022-6