10th Workshop on Evaluation and Usability of Programming Languages and Tools

PLATEAU 2019, October 24, 2019, New Orleans, Louisiana, USA

Edited by
Sarah Chasins
Elena L. Glassman
Joshua Sunshine

Editors

Sarah Chasins
University of California, Berkeley, USA
schasins@berkeley.edu

Elena L. Glassman
Harvard University, Cambridge, USA
eglassman@g.harvard.edu

Joshua Sunshine
Carnegie-Mellon University, Pittsburgh, USA
sunshine@cs.cmu.edu

ACM Classification 2012
Human-centered computing → Human computer interaction (HCI); Software and its engineering → General programming languages

ISBN 978-3-95977-135-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-135-1.

Publication date
March, 2020

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.PLATEAU.2019.0

ISBN 978-3-95977-135-1 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Regular Papers

10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019).
Editors: Sarah Chasins, Elena L. Glassman, and Joshua Sunshine
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

Programming languages exist to enable programmers to develop software effectively. But programmer efficiency depends on the usability of the languages and tools with which they develop software. The aim of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) is to discuss methods, metrics, and techniques for evaluating the usability of languages and language tools. The supposed benefits of such languages and tools cover a large space, including making programs easier to read, write, and maintain; allowing programmers to write more flexible and powerful programs; and restricting programs to make them more safe and secure. The 10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019) was held on October 24, 2019 in New Orleans, Louisiana, USA, and collocated with UIST 2019. The workshop gathered the intersection of researchers in the programming languages and human-computer interaction communities to share their research and discuss the future of evaluation and usability of programming languages.

It is our pleasure to present this year's proceedings. Together, these papers demonstrate the remarkable scope and applicability of the workshop, with topics that include software development techniques, software evolution, programming by example, and empirical studies in human-computer interaction. Our thanks go to the authors, reviewers, speakers, and attendees, without whom this workshop would not have been possible.

Joshua Sunshine, Elena Glassman, and Sarah Chasins PLATEAU 2019 Co-Chairs

10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019).
Editors: Sarah Chasins, Elena L. Glassman, and Joshua Sunshine

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Approaching Polyglot Programming: What Can We Learn from Bilingualism Studies?

Rebecca L. Hao
Department of Computer Science and Department of Linguistics, Harvard University, Cambridge, MA, USA
rhao@college.harvard.edu

Elena L. Glassman
Department of Computer Science, Harvard University, Cambridge, MA, USA
glassman@seas.harvard.edu

Abstract

Today's programmers often need to use multiple programming languages together, enough that this practice has been given the name "polyglot programming." However, not much is known about how using multiple programming languages affects programmers, despite its increasing ubiquity. If we want to better design programming languages and improve the productivity of programmers who program in multiple programming languages, we should seek to understand the user in this context: we need to better understand the impact that polyglot programming has on programmers. In this paper, we pose several open research questions to begin to approach this question, drawing inspiration from psycholinguistic studies of bilingualism, because despite the differences between natural languages and programming languages, the questions considered in natural language bilingualism studies are relevant to programming languages, and the existing findings may prove useful in guiding our intuitions, methods, and priorities as we begin to explore this topic. In particular, we pay close attention to the implications that code switching (switching between languages within a conversation) and interferences (ways an unintended language may influence one's use of an intended language) may have on our understanding of how using programming languages may impact a programmer.

2012 ACM Subject Classification Human-centered computing → HCI design and evaluation methods; Software and its engineering → General programming languages

Keywords and phrases Programming languages, polyglot programming, bilingualism

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.1

1 Introduction

Programmers today are often expected to use multiple programming languages at once, engaging in *polyglot programming*. Perhaps they are developing for the web, and using some mixture of HTML, CSS, Javascript, SQL, and others. They could be using a virtual platform like JVM or .NET, using a number of programming languages and libraries together. Maybe they require a more expressive, higher-level scripting language or domain specific language, embedding it into a lower-level one with better run-time performance like C or C++. On a project level, they could build separate programs and let them interact and send data to each other using pipes, cross-compile one language into another, or use a language's ability to interface with other languages. Even within a single file, they may need to mix programming languages [1].

The advantages of polyglot programming include the cost-effective ability to reuse code already written in other languages, and using programming languages that are better suited to one's goals. After all, programming languages are not created equally – each has its characteristics and strengths, and is more useful in certain domains and tasks than others.

Given the commonality of programmers writing code in multiple programming languages, we think that it is important to know more about the impact that using multiple programming

languages has on the programmer, and how the knowledge and use of multiple programming languages affects the code the programmer writes. Intuitively, we may think that knowing multiple programming languages strengthens one's understanding of programming concepts, or that switching between languages is difficult and may incur some cognitive cost, but how well do these intuitions hold and how can we go about studying them?

Though programming languages and natural languages have their differences, there are still a number of compelling parallels. One such parallel that is relevant to polyglot programming is that there are many polyglots of natural language, who use different languages in different domains. Similarly to programming languages, natural language speakers may switch languages at multiple scales, a practice known as code-switching, substituting words and phrases in one base language with those of another, or switching languages altogether depending on who they are speaking to and in what context.

There have been a number of interesting findings in psycholinguistics, the study of the relationship between linguistic behavior and psychological processes, that deal with the learning, use, and switching of multiple languages [7, 8, 9]. In this paper, we examine how psycholinguistic studies of natural language bilingualism may relate to that of programming languages, and argue that a psycholinguistic lens of polyglot programming may be useful towards guiding the questions that we ask, the priorities that we set, and the methodologies that we use. We choose to focus specifically on the case of the *use* of multiple programming languages, which involves code switching (switching between two language systems), and the direct impact that knowledge of multiple programming languages has on programming. However, this psycholinguistic lens may also prove to be useful in other polyglot programming topics, including programming language learning and the effects that the knowledge of multiple programming languages has on programming concept understanding.

In this way, by learning more about how programmers use and are affected by their use of multiple programming languages, we can inform programming language design in the context of its use with other languages, and also inform the practice of using multiple programming languages, working towards increasing programmer productivity and efficiency.

2 Related work

2.1 HCI for programmers

HCI methods have already been used to examine programming languages and environments. This work seeks to better use the knowledge, principles, and methods of human computer interaction (HCI) in programming language design, showing that gathering data on and considering programming language usability leads to better design [15, 17]. Similarly, we seek to understand the impact of knowing and using multiple programming languages on users: the programmers.

2.2 Related work in programming languages

Work in the psychology of programming and the HCI-based approach toward programming language design has made major strides. For example, there are investigations of how a programmer's experience level impacts how they reason about, read, and/or debug programs [6, 13, 21]. However, the question of polyglot programming is less thoroughly explored.

Most of the work that addresses multiple languages focuses on the learning and teaching of programming languages. This work includes justifying and exploring the impact of using specific languages (like Python or Java) in introductory computer science courses [10, 11], and effective methodologies for learning subsequent languages [16, 19].

There is some literature surrounding the design of multi-language systems like .NET that discuss ways to support polyglot programming and its design [14], but it does not focus on the effects that these multi-language systems have on programmers and their productivity.

However, there is evidence that suggests that using multiple programming languages within a project may result in decreases in code quality. Kochhar et al [12] investigated the impact of multiple programming languages on software quality, applying a method of looking into code quality across Github projects proposed by Ray et. al [18]. In this study, Kochhar et al. looked for the effects of using different programming languages on the number of bug fix commits in a dataset of popular projects from Github. They found that in general, projects that used more programming languages were more bug-prone, especially for memory, concurrency, and algorithmic bugs, and that specific languages like C++, Objective-C, and Java were more bug-prone [12].

2.3 Related work in natural languages

Studies regarding natural language polyglots are more numerous. A central observation is that bilinguals operate using different language modes: the monolingual speech mode and the bilingual speech mode [8]. In the monolingual speech mode, the bilingual almost completely deactivates one language, while in the bilingual speech mode, they choose a base language but activate another language occasionally in the form of code-switching and borrowing [8]. This deactivation when a language is not in use involves prefrontal brain activity associated with cognitive control [2]. This cognitive cost may be due to task switching, as it has been shown that in word-reading and picture-naming, a greater time delay occurs after a task switch than a task repetition [20].

However, there is a nuance to *when* this cognitive cost occurs that may be relevant: a recent study suggests that *deactivating* a language requires cognitive control, while *activating* a new language may not [4]. This study used magnetoencephalography (MEG), and found that American Sign Language (ASL)-English bilinguals had increased brain activity in brain areas involved in cognitive control (the anterior cingulate cortex and dorsolateral prefrontal cortex) when deactivating a language, but not when activating one [4]. Additionally, because certain ASL-English bilinguals can sign and speak simultaneously, they also found that using both languages simultaneously did not necessarily incur a greater cognitive cost than producing one language on its own [4].

Psycholinguistic studies have shown that even though it is not conscious or intentional, code switching behavior is often rule-based: there are systematic patterns to how and when speakers code switch. Speakers may code switch by substituting single words from one language into a sentence with the grammar of the base language, or alternate between languages on sentence boundaries [7].

It has also been found that infants demonstrate cognitive load through involuntary pupil dilation and eye-tracking fixations when they listen to language switches. These effects, however, were reduced when going in a non-dominant to dominant language direction, and when switching languages when crossing sentence boundaries [5].

Additionally, even when in a monolingual speech mode and trying to speak one language (L_a), sometimes other languages that the bilingual knows (e.g. L_b) influence their speech in the form of interferences. This is known as cross-linguistic influence, and comes in the form of static interferences or transfers, which are more permanent traces of L_b in L_a (like with accents), and dynamic interferences, which are brief intrusions of L_b in L_a (like accidentally stressing the wrong syllable or momentarily using a word or grammatical structure in the other language). Static interferences are linked to a person's competence in L_b, while dynamic

interferences are more likely to occur when one is stressed, tired, or emotional. A language that is more closely related to the intended language has been found to have more of an influence on the intended language than one that is are more distant [9].

3 Open questions and potential approaches

We pose four guiding questions regarding the use of multiple programming languages that are inspired by the hypotheses, methods, and work within the psycholinguistic study of bilingualism.

1. How does knowledge of certain programming languages influence programming in another language?

Is there cross-linguistic influence [9] between programming languages? Do programmers ever accidentally use an unintended programming language syntax or concept while trying to program in a specific programming language? This could occur on a number of levels, taking on forms like: using a function word or symbol incorrectly in the desired language because of that symbol's use in another language, using an incorrect construct (e.g. for loop constructs look and act differently between C and Python), and approaching a function or program using logic that is clearly inspired by another language (e.g. trying a "Pythonic" way in a language where it is less conventional, or not even possible).

This may be challenging to isolate and quantify especially in existing projects and code, so studying programmers while they are programming may be of interest. We can observe and look out for errors that programmers run into and corrections that they make as they program, and their approaches to certain programming problems, paying particular attention to what programming language, if any, these errors come from and approaches are inspired by. This resembles the collection of linguistic data, which often involves utterances from native speakers, which are then analyzed by linguists for patterns and structures. To better understand the influences programming languages may have on each other, it may also be useful to investigate how much programmers seek equivalences between languages, for example, how often questions are asked about equivalent keywords or constructs from one language to another on forums like Stack Overflow.

2. Are certain kinds of languages more prone to influencing other kinds?

From there, it also seems important to determine which languages influence and are influenced by others, and if there are patterns in terms of the characteristics of the languages involved, or in terms of the relationships between the languages. For example, do similar languages influence each other more, as we find with natural language? Does knowledge of languages that use certain paradigms (e.g. static versus dynamic, functional versus object-oriented) have greater influence over the use of other programming languages?

In order to approach this question, we may need a more concrete way of describing a programming language polyglot's knowledge of their programming languages, since proficiency is likely closely related to the ability for a certain language to influence another. Natural language bilingualism studies have generally used two main factors to characterize bilinguals – language proficiency and language use – and have used these two variables as axes in a grid approach to map the language history of a polyglot [9]. In this approach, a language is represented on this grid as a datapoint of proficiency and use, and multiple grids may be used to show proficiency and use in specific domains, or across time. A similar approach

may prove useful for programming languages as well, because in this way we can investigate the time course of programming language learning, and have an improved ability to study how code quality and performance change with time, without requiring a study to be set up around a specific computer science course and structured learning. However, this may require that we determine what variables are relevant in programming language knowledge, and what data we would like to and can collect from our participants.

3. How and when do programmers code switch?

As programmer "code switching" seems to occur at several distinct levels of granularity, it may be useful to look into the occurrences of "code switching" at these different levels. An example delineation of these levels could be: switching within a line, switching for each line, switching at "logical blocks" of the code (e.g. where the author decided to stylistically include a new line for readability), and switching between files. We could learn more about the contexts in which code switching occurs in practice, to better understand the motivations of why programmers code switch in the first place.

From there, we can then look at the impacts of code switching on programmers. It may be useful to try out switching tasks inspired by psycholinguistic studies that, require speakers to name images in different languages in a controlled manner, measuring reaction time differences or neuroimaging studies [2]. The equivalent of having speakers name images could be having programmers name constructs for short code snippets. Alternatively, to study whether reading and understanding multilingual code incurs a cognitive cost, we could present programs in different languages and observe eye tracking or pupil dilation to indicate the presence of cognitive control.

4. What is the degree of cognitive control being exerted when switching between programming languages? Are there cases where we can reduce the cognitive cost of switching and thereby increase productivity?

Additionally, investigating programming in different languages using MEG may provide information of whether switching programming languages has a similar effect on prefrontal areas of cognitive control. This can be investigated when programming in a language-agnostic way (e.g. writing pseudocode), or when activating and deactivating languages (e.g. writing a program or code for a project that requires switching between programming languages).

Does the degree of cognitive control depend on the programmer's proficiency in a language? Is the cognitive cost higher in novices and lower in experts?

From there, inspired by natural language, we can investigate whether there are cases where the effects of code switching are reduced. Like how the effect is reduced in natural language when going from nondominant to dominant language and on sentence boundaries [5], for programming languages, are effects reduced in certain directions, like going from programming languages one is less familiar with to those one is more familiar with? Are effects reduced more on boundaries of a certain granularity, like between "sentences"? Like how cognitive cost is reduced in ASL-English bilinguals when using both languages simultaneously [4], is there a difference between when trying to program exclusively in one language or when actively harnessing knowledge from multiple languages simultaneously (e.g. pseudocode)?

4 Conclusion

We have proposed several open questions and next steps for the study of polyglot programming, inspired by findings and methodologies in natural language bilingualism. Even though programming languages likely do not rely on the same language faculty attributed to natural language and instead on different programming concepts, natural language bilingualism still serves as a useful model to guide foundational decisions to more concretely approach polyglot programming questions, and provide an intuition for possible phenomena related to the use of multiple programming languages.

This approach may also be extended beyond the *use* of programming languages, to areas like programming language learning and whether there are benefits of knowledge and use of multiple programming languages. For example, does knowing more languages improve your understanding of programming concepts? Bilingualism studies demonstrate a bilingual advantage in non-linguistic tasks that require cognitive control [3] – is polyglot programming beneficial to a programmer's general programming skills?

With an improved understanding of the use of multiple programming languages, we may be better equipped to design programming languages and tools given this growing polyglot programming landscape and improve programmer efficiency and polyglot programming workflows.

References

1 Why are multiple programming languages used in the development of one product or piece of software?, April 2018. URL: `https://softwareengineering.stackexchange.com/questions/370135/why-are-multiple-programming-languages-used-in-the-development-of-one-product-or`.

2 Jubin Abutalebi and David W. Green. Control mechanisms in bilingual language production: Neural evidence from language switching studies. *Language and Cognitive Processes*, 23(4):557–582, 2008. `doi:10.1080/01690960801920602`.

3 Ellen Bialystok. Cognitive complexity and attentional control in the bilingual mind. *Child Development*, 70(3):636–644, 1999. `doi:10.1111/1467-8624.00046`.

4 Esti Blanco-Elorrieta, Karen Emmorey, and Liina Pylkkänen. Language switching decomposed through meg and evidence from bimodal bilinguals. *Proceedings of the National Academy of Sciences*, 115:201809779, September 2018. `doi:10.1073/pnas.1809779115`.

5 Krista Byers-Heinlein, Elizabeth Morin-Lessard, and Casey Lew-Williams. Bilingual infants control their languages as they listen. *Proceedings of the National Academy of Sciences*, 114:201703220, August 2017. `doi:10.1073/pnas.1703220114`.

6 Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *PPIG*, 2002.

7 David W. Green and Li Wei. Code-switching and language control. *Bilingualism: Language and Cognition*, 19(5):883–884, November 2016. `doi:10.1017/S1366728916000018`.

8 François Grosjean. The bilingual's language modes. *One mind, two languages: Bilingual language processing*, pages 1–22, 2001.

9 François Grosjean and Ping Li. *The Psycholinguistics of Bilingualism*. Wiley-Blackwell, Chichester, 2013.

10 Said Hadjerrouit. Java as first programming language: a critical evaluation. *SIGCSE Bulletin*, 30:43–47, 1997.

11 Tony Jenkins. The first language - a case for python? *Innovation in Teaching and Learning in Information and Computer Sciences*, 3(2):1–9, 2004. `doi:10.11120/ital.2004.03020004`.

12 Pavneet S. Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573, March 2016. doi:10.1109/SANER.2016.112.

13 Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. Tracking students' cognitive processes during program debugging—an eye-movement approach. *IEEE Transactions on Education*, 59(3):175–186, August 2016. doi:10.1109/TE.2015.2487341.

14 Bertrand Meyer. Multi-language programming: how .NET does it. *Software Development*, 2002.

15 Jonathan Aldrich Michael Coblenz, Brad A. Myers, and Joshua Sunshine. Interdisciplinary programming language design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 133–146, New York, NY, USA, 2018. ACM. doi:10.1145/3276954.3276965.

16 Casey O'Brien, Max Goldman, and Robert C. Miller. Java tutor: Bootstrapping with python to learn java. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, pages 185–186, New York, NY, USA, 2014. ACM. doi:10.1145/2556325.2567873.

17 John F. Pane, Brad A. Myers, and Leah B. Miller. Using hci techniques to design a more usable programming system. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 198–206, September 2002. doi:10.1109/HCC.2002.1046372.

18 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. *Proc. FSE 2014*, pages 155–165, November 2014. doi:10.1145/2635868.2635922.

19 Karen P. Walker and Stephen R. Schach. Obstacles to learning a second programming language: An empirical study. *Computer Science Education*, 7(1):1–20, 1996. doi:10.1080/0899340960070101.

20 Florian Waszak, Bernhard Hommel, and Alan Allport. Task-switching and long-term priming: Role of episodic stimulus–task bindings in task-shift costs. *Cognitive Psychology*, 46:361–413, 2003.

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

A Pilot Study of the Safety and Usability of the Obsidian Blockchain Programming Language

Gauri Kambhatla
Electrical Engineering & Computer Science, University of Michigan, Ann Arbor, MI, US

Michael Coblenz
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Reed Oei
Department of Computer Science, University of Illinois, Urbana, IL, USA

Joshua Sunshine
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Jonathan Aldrich
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Brad A. Myers
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Although blockchains have been proposed for building systems that execute critical transactions, security vulnerabilities have plagued programs that are deployed on blockchain systems. The programming language Obsidian was developed with the purpose of statically preventing some of the more common of these security risks, specifically the loss of resources and improper manipulation of objects. The question then is whether Obsidian's novel features impact the usability of the language. In this paper, we begin to evaluate Obsidian with respect to usability, and develop materials for a quantitative user study through a sequence of pilot studies. Specifically, our goal was to assess a) potential usability problems of Obsidian, b) the effectiveness of a tutorial for participants to learn the language, and c) the design of programming tasks to evaluate performance using the language. Our preliminary results tentatively suggest that the complexity of Obsidian's features do not hinder usability, although these results will be validated in the quantitative study. We also observed the following factors as being important in a given programmer's ability to learn Obsidian: a) integrating very frequent opportunities for practice of the material – e.g., after less than a page of material at a time, and b) previous programming experience and self-efficacy.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Human-centered computing → User studies; Human-centered computing → Usability testing

Keywords and phrases smart contracts, programming language user study, language usability

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.2

1 Introduction

Blockchains have a myriad of applications, including shipping, supply chain, auctions, storing health records, and voting [8]. A blockchain is used when a central authority cannot be trusted; instead of a singular ledger (as is used by an entity like the Federal Reserve), a distributed ledger is used to keep track of transactions that are made, held accountable by the users of the blockchain themselves. Smart contracts are programs that are deployed across a blockchain network. Since blockchain technology is often used in potentially high-stakes contexts, such as financial transactions, if a bug in a contract is exploited, it could involve loss of important resources (like money or personal items). Some of the more common of these security risks are (1) loss of resources and (2) manipulating objects at improper times. Current status quo blockchain languages (such as Solidity) have no mechanism to prevent such bugs.

Obsidian is a programming language for writing smart contracts that is designed to prevent these two issues; it uses both typestate and linear types to prevent such risks statically [2]. In order to detect and prevent these vulnerabilities, Obsidian introduces the concepts of ownership, assets, and states. Ownership is a property of *references*, rather than the objects themselves. There are three types of references; `Owned`, `Unowned`, and `Shared`. An object can have exactly one `Owned` reference, and any number of `Unowned` references. It can have any number of `Shared` references, but only if there is no `Owned` reference. Making an object an *asset* enables the compiler to prevent losing track of `Owned` references. Assets, combined with permissions (the types of ownership references) allow the compiler to detect the loss-of-resource security bug. Objects can also have *states*, and certain transactions can only occur within a particular state. States help prevent improper manipulation of objects; certain contracts might allow certain transactions only when in an appropriate state. A *contract* is similar to a class in other object-oriented languages, with corresponding *transactions* (analogous to methods) and fields (like member variables) [2].

Yet how do we know Obsidian is effective in accomplishing what it is designed to achieve? Do users actually make fewer of these security mistakes in Obsidian than in other common smart contract programming languages, such as Solidity? And how usable is Obsidian in the long term, after programmers have mastered the language? In this paper, we focus on beginning to answer these questions; we want to see whether the bugs that Obsidian detects are ones we observe people inserting frequently in the laboratory. In addition, we want to see whether the language is in fact usable, and how we can teach it to potential users.

We intend to run a user study where participants will learn either Solidity or Obsidian through a tutorial, and then attempt a series of programming tasks that will test their knowledge of the language. We will also assess the safety of the code they have will write. In order to develop these programming tasks and the Obsidian tutorial, we first ran a series of pilot studies. This paper describes the design and results of those pilot studies, addressing in particular, the following research questions:

RQ1 What are the most significant usability problems with Obsidian?

RQ2 How should a tutorial be best authored to help one learn a language sufficiently well to get to the point where usability can be evaluated?

RQ3 How do we design programming tasks that assess the ability to program effectively and also potentially expose the types of bugs that our tool would detect or prevent?

The answers to these research questions, and what we learned from this study could also be of interest to a broader audience; specifically, the initial findings we report in this paper that could feed into future work are as follows:

- A potential design of a method of teaching programmers to effectively learn an entirely new language composed of complex features
- Hypotheses on why some programmers are more effectively able to learn and utilize a new programming language than others
- Evidence that the additional features Obsidian introduces for security can actually be effectively used by programmers

It is important to note the findings we present here through our pilot studies are not fully validated; we plan to substantiate them in our follow-up quantitative user study.

2 Related Work

Designing programming tools and languages has been shown to be better when done with a human-centered approach [4]. It is often inadequate to just use technical methods in assessing properties of a language. For example, even if type soundness has been proven,

the language might be too complex for a programmer to learn and use [3]. Programming languages are interfaces between a human programmer and a computer to enable people to effectively create programs, and as a result, should be tested and evaluated through user studies with actual developers. Obsidian has been designed using such a human-centered approach [1], and is similarly being evaluated through user studies.

There are differences in *approaches* to programming between novices and experts [6]. Novices generally lack detailed mental models of what they are doing, and approach programming one line at a time, rather than as meaningful blocks [6]. On the other hand, experts tend to be faster and more accurate, able to make use of a variety of effective strategies and more vast knowledge base [6]. The participants in the pilot studies fall somewhere between novice and expert, and their approach to programming may play a role in their success in the tasks given to them. While knowledge is the ability to state how something works, strategies are the ability to apply that knowledge. There are many different strategies programmers use to comprehend programs, including systematic (tracing through all the code in detail), as-needed (only looking at code related to the required task), and inquiry episodes (asking a question, hypothesizing an answer, and verifying by looking at code, or compiling the program), among others [7]. The programming strategy participants in our study use is a variable that could influence how they learn Obsidian.

In addition, studies of novice programmers have shown that past experience is strongly predictive of self-efficacy, and that a strong mental model increases self-efficacy [5]. Ramalingam et al. also found that both of these (self-efficacy and strong mental models) affect performance in an introductory class setting. Active learning helps develop these mental models; it requires an individual to be active in constructing their own knowledge and finding new ideas [9]. An active learning environment is beneficial to learning programming because it is one that provides immediate feedback and engaged participation, which can help develop mental constructions of programming concepts [9]. Although Obsidian is not being taught to novice programmers in a course setting, self-efficacy and mental models might still play roles in the ability to learn the language effectively, and teaching through active learning might help participants pick up the language more successfully.

3 Study Design

In this work, we focused on designing the Obsidian condition for the quantitative user study. Participants in our pilot studies were asked to do the Obsidian tutorial, and then some more in-depth programming tasks, during which they were asked to think out loud. They were told they could ask questions about anything they didn't understand throughout the study; the answers to these questions could then be incorporated in the next iteration of the tutorial and tasks. This pilot study was approved by our IRB, and participants were paid $10/hour for their time during the study.

We chose tasks with potential real blockchain use-cases for external validity; by approximating code an actual smart contract programmer might write, we hope to be able to generalize our conclusions to our proposed end applications of Obsidian. Each task had a series of parts that tested different aspects of Obsidian, and are described below, along with the design of the tutorial.

Obsidian Tutorial

The tutorial consisted of a Qualtrics survey with a few multiple choice, write-in answers, short answer questions, and small programming questions. It was split into eight sections: four on ownership, one on assets, and three on states. An early version of the tutorial had only four

sections, but we subdivided it because of participant feedback of having too much information at once with just a few sections. Participants completed programming questions in a separate Visual Studio Code (VSCode) window, in which participants could compile their code to confirm that they had complied with the type system's requirements. Some of the short-answer questions were purposefully open-ended, as in "Describe the relationship between ownership and states in your own words." The objective was to make the participant think about, and thereby internalize, these concepts, rather than to evaluate their understanding.

Auction Task

The **Auction** task simulates an English auction; there are multiple Bidders who each make a Bid for a single Item being sold by a Seller. The highest Bidder receives the Item for the price of the highest Bid. However, unlike in a normal English auction, when a Bidder makes a Bid, they give the Money to the Auction house *immediately*, and the Money is returned to that Bidder if another Bidder makes a higher Bid. This requirement ensures that all bids are legitimate, rather than allowing for a failure mode in which the sale cannot be completed because a bidder does not pay for the item. A consequence is that the contract must return the bid money every time a higher bid is made. If a participant forgot to do this, they would lose a resource (the bid money), which the Obsidian compiler would detect.

The task included five parts. The most significant parts of this task were 3 and 4, which allowed for a potential loss-of-resource bug. **Part 3:** Write code to return money to a Bidder in the case that the bid is not greater than the current maximum bid. This part (whose starter code is shown in Listing 1) was created to prime the participant for the next question, so the participant is aware there is a way to return money to a Bidder. **Part 4:** Update the current maximum bid (i.e., given that the new bid is greater than the previous, replace the `maxBid` with the new one). Since Money is an asset, and a Bid contains Money, the participant will not be able to simply do something like `maxBid = newBid`; the money will have to be returned to the Bidder before overwriting `maxBid`, otherwise they will get a compiler error in regards to overwriting an asset (losing a resource). This part (starter code also shown in Listing 1) was meant to capture this potential pitfall. To implement this correctly, a participant must return the Money to the Bidder and disown the Bid. They could write code to do this or call a given function that does both these things.

Listing 1 Partial Starter Code for Auction Parts 3 & 4.

```
transaction makeBid(Auction @ Open this, Bid @ Owned >> Unowned newBid,
                    Bidder @ Unowned bidder) {
   if (newBid.getAmount() > bid.getAmount()) {
      setCurrentBid(newBid);
      Bidder tempBidder = maxBidder;
      maxBidder = bidder;
   }
   else {
      //Part 3. TODO: return the newBid money to the bidder.
      //You may call any other transactions as needed.
   }
}
transaction setCurrentBid(Auction @ Open this, Bid @ Owned >> Unowned b) {
   //Part 4. TODO: set the current bid to the new bid b.
   //You may call any other transactions as needed.
}
```

Pharmacy Task

The **Pharmacy** task simulates a pharmacy; there are Prescriptions, PrescriptionRecords (which keep track of Prescriptions), and Patients (who have Prescriptions), as well as the Pharmacy contract, which has a list of PrescriptionRecords. The goal of this task is to have the participants utilize states to prevent improper use of the prescription; a Patient can only fill a Prescription when they have additional refills. There were three parts to this task; the third was most significant, which had a participant writing in lines of code to fill a prescription. In this part (starter code shown in Listing 2), the participant must take an element off a list, apply transactions to that object, and add it back to the list. It assesses understanding of the code, and the ability to use the language to implement what the participant wants to occur, as well as correct use of states.

Listing 2 Partial Starter Code for Pharmacy Part 3.

```
transaction fillPrescription(Prescription @ Unowned prescription) {
   MaybeRecord maybeRecord = prescriptionList.removeIfExists(prescription);
   // TODO Part 3: Fill in the rest of this transaction.
   // You will need to call the doFill transaction in this
   // class (Pharmacy) on the appropriate PharmacyPrescriptionRecord.
   // Be sure to record that the prescription has been filled.
}
```

Gambling Tasks

The **Gambling** task simulates betting at a Casino; before every Game, Bettors place a bet on the outcome of the Game. A set of restrictions and assumptions is given to the participant, as well as a sequence diagram showing a potential timeline of possible events, and structural diagram explaining how contracts relate to each other. This task is purposefully more open-ended; the participant can design and implement the Casino contract however they would like using Obsidian, but the program must comply with the given requirements. The goal of this task was to see how the participants used what they learned about the language to design their own program (and be able to implement it). Code for the other contracts (Bettor, Money, etc.) is not shown here due to space constraints.

Listing 3 Partial Gambling Starter Code.

```
main asset contract Casino {
    Money @ Owned money;
    Game @ Owned currentGame; //The game that is currently being played
    BetList @ Shared bets; //The bets for the current game being played

    Casino @ Owned() {
        money = new Money(100000);
        currentGame = new Game();
        bets = new BetList();
    }
    //TODO: Add your code here.
}
```

4 Initial Results

Since these studies were exploratory, serving as preparation for our evaluative user study, we changed the tutorial and tasks after nearly every participant, based on the qualitative results and participant feedback. We recruited six participants (three men, and three women),

all of whom were undergraduates in computer science at different universities. Despite all the participants being almost the same amount through their undergraduate career (they were either rising Juniors or Seniors), there was a wide range of 3 - 9 years of previous programming experience, and they all had a different CS education. All the participants were familiar with Java to varying degrees (ranging from 1.5 to 8 years of experience), which was a requirement for participation, since Obsidian is similar to Java. None of the participants had any previous experience with blockchain programming. Each participant was given an anonymous participant ID: R0, R1, etc.

Participants worked on a 15" MacBook Pro with a second monitor that displayed the documentation pages and instructions for the tasks. We recorded screen and audio of each session. The programming tasks were all done in VSCode, for which we created a plugin for Obsidian syntax highlighting, and through which we ran the Obsidian compiler.

Some participants did not seem to understand and internalize the information given in the tutorial despite reading it. For example, R2 said the idea of ownership made sense theoretically, but not in application. This may be because R2 missed some key concepts, like the fact that ownership is a property of a reference, not an object. In contrast, other participants seemed to fully comprehend the material; R1, R3, and R5 got nearly all questions correct. A summary of the tutorial results are shown in Table 1.

Table 1 Tutorial Results.

Participant	Tutorial Version	Questions Correct	Time
R0	2 Parts, No questions	N/A	N/A
R1	4 Parts, all multiple-choice	19/19	35 min
R2	4 Parts, all multiple-choice	11/19	40 min
R3	8 Parts: multiple-choice, short answer, programming	20/22 (non-code), 8/8 (code)	1.25 hours
R4	8 Parts: multiple-choice, short answer, programming	14/22 (non-code), 4/8 (code)	2.5 hours
R5	8 Parts: multiple-choice, short answer, programming	20/22 (non-code), 8/8 (code)	53 min

All the participants except R4 did the **Auction** task. They all had the most trouble with part 4; neither R0 nor R2 was able to complete it. Every participant who did part 4 started by writing `bid = newBid`, which overwrote an owned reference to an asset; the compiler generated an appropriate error message. Participants R1, R3, and R5 realized that ownership of the original bid must be transferred first. R1 even said out loud after typing this in, "Oh is bid an asset? Yes, it is an asset. Then this should fail." After compiling the code and confirming the failure, R1 made sure the original bid was transferred. R3 did the same, but first returned the Money in the Bid to its Bidder. R5 was the only one that used the given transaction `returnBidMoney()` (which gave the Money back to the Bidder and disowned the current bid).

All the participants except R4 were given the **Pharmacy** task. R0 was able to complete parts 1 and 2, but was unable to do any of the third part; the participant could not figure out what to do, and was stopped by the experimenter due to time constraints. Participant R2 did part 1 incorrectly, and part 2 correctly. R2 was unable to figure out the third part, and stopped after a significant amount of trial and error due to time constraints. R2 was asked by the experimenter to answer in pseudocode; the participant did this mostly correctly

(but forgot to check if the PrescriptionRecord actually existed). Participant R3 did the whole task correctly, but forgot to append the PrescriptionRecord back on the list after consuming a refill. Participants R1 and R5 did the entire task correctly.

Only R3 and R5 were given the **Gambling** task (it was created after R1, and both R2 and R4 could not take it because of time constraints). R3 took 1 hour to complete the task and get the code to compile. Most of this time was spent on understanding the architecture of the objects given, and the structure of the problem itself. Participant R5 took 36 minutes to do this task and get the code to compile. In this version, the requirements were unchanged, but the problem was made more clear, and there were fewer layers of abstraction. For R5, an architectural diagram was added to show how objects are related (e.g., a Bet has a Bettor and a BetPrediction), in addition to the sequence diagram (showing an example of potential actions) given to R3. Both R3 and R5 successfully designed and implemented a program that met all the requirements given in the specification. They made correct use of states and permissions. R3's program was 63 lines long, and R5's was 56 lines long.

5 Discussion

A clear distinction can be drawn between the results of the different participants. Participants R0, R2, and R4 struggled with the study (both the tutorial and the programming tasks), while participants R1, R3, and R5 found the tutorial and tasks easy to understand. There was almost no middle ground; one of the challenges in designing the tutorial and tasks was that after one participant, it seemed the exercises were very difficult and needed to be simplified, but after another, they seemed too simple. While R0, R2, and R4 needed prompting and additional explanation, and still did not finish all the exercises, R1, R3, and R5 easily completed the tutorial (including the programming exercises for R3 and R5). These three participants were able to work through the programming tasks, write code in the language, understand the compiler errors, and make fixes when necessary. They had understood the new *concepts* they were taught; while working through the Pharmacy task, R5 said "... and this takes an owned [reference] and makes it unowned...," showing that the participant grasped the idea of ownership. Both R1 and R3 also said similar things while thinking out loud. On the other hand, R0, R2, and R4 got stuck; it was clear that although they read through the tutorial, they did not fully understand the concepts. For example, despite reading in the documents, answering questions about it in previous parts of the tutorial, and given an explanation by the experimenter, R4 did not seem to understand how state and permission transitions worked as types for a parameter in a transaction declaration.

Below are hypothesized explanations for the differences among participant performance:

- **General programming experience**. The participants who struggled had an average of 3.2 years of programming experience; the others had an average of 7.5 years of experience. One of the key observations we made during the pilot studies was that the participants who struggled had the most trouble because upon getting an error, they would either get stuck or start trying random solutions that neither made sense conceptually nor syntactically. Participants who successfully completed the tutorial and tasks used a variety of techniques to fix compiler errors. They had less trouble understanding the compiler messages (most of the time, they knew what the problem was immediately), and even if they did not, they used strategies to find a solution. These observations suggest that the amount of past programming experience played a role in how effective participants were in using the unfamiliar language.

- **Object-oriented (OO) programming experience**. The less effective participants had an average of 2.2 years of Java experience, and the more effective ones had an average of 5.3 years (familiarity with Java was a prerequisite to participating in this study). Perhaps some concepts that are learned in OO classes, or learned through developing in OO languages affects the way one learns other OO languages.
- **Teaching style**. The tutorial was text- and exercise-based. Perhaps the participants who struggled may have done better if the material had been taught in a video, or lecture format, or in a more interactive way.
- **Type of programmer**. Different programmers use different programming strategies (see §2). Perhaps some of these strategies are more effective in learning or using Obsidian than others. While there were systematic and opportunistic strategies used in the group that was more effective, the ones who had more trouble only used an opportunistic approach.
- **Self-efficacy and interest**. Self-efficacy appeared to play a role in how effective participants were in using the language while completing the tasks. The effective participants were very confident; they were not afraid to question the experimenter about things they did not understand and things they thought were wrong in the tutorial or tasks. They were also more relaxed; the participants who struggled seemed tense. It is difficult to tell whether confidence entailed being effective in completing the tasks, or whether doing well made participants more confident. In addition, lower self-efficacy might have played a role in whether participants asked questions when they were confused, and thus how well they performed. The effective participants also had a genuine *interest* in what they were learning. They made noises of surprise or interest while reading the tutorial, saying for example "Oh, that makes sense", or "huh, that's interesting." In contrast, other participants went through the study like taking an exam, or doing an assignment; they were just doing exercises they were given. Perhaps having an interest in learning a new language made participants more successful at completing the given exercises, or at least created a more open mindset that might have allowed for faster debugging.

RQ1 asked about identifying potential usability problems with Obsidian. Multiple participants mentioned things related to the environment; a few expected more precise autocomplete (VSCode has a default autocomplete that lists any words used previously in the window, which confused some participants), and one asked about in-place error diagnostics (showing errors as one types). In addition, a few participants were confused by some of the compiler errors. Two participants tried to add a permission or state to an object they were creating, like `g = new Game@FinishedPlaying()`. The error message was `Error: '(' expected, but @ found`, which they eventually figured out. R2 did not understand the compiler message `variable is an owning reference to an asset, so it cannot be overwritten` at first, and when it was explained by the experimenter, was unable to figure out how to resolve the problem. R5 was confused by the error message `Can't reassign to variables that are formal parameters or which are used in a dynamic state check.` R5 commented that even distinguishing between the two (whether a formal parameter or used in a dynamic state check) might be more helpful. None of the participants had much feedback about the language itself, although there was a general consensus that states were an easier concept to grasp than ownership because they are more familiar.

RQ2, asked about how to design a language tutorial. Teaching any new programming language is hard; teaching a new programming language that has more complex features may be even harder. From our results, we learned a few things: (1) material that should be read needs to be split into manageable sections, (2) simply giving people material to read is

not enough for them to understand it – they need to be given questions along the way to force them to thoroughly comprehend it, and (3) these questions must have them actually *do* problems themselves, rather than merely pick an answer choice. When given large amounts of reading material at a time, participants had trouble remembering all the information; an early participant commented that they wanted to see an example when doing the programming tasks, and while they knew there was probably one in the documentation, they had no idea where to look. After splitting the reading material into eight sections (each less than a page), we observed that later participants found it easier to understand the material initially, but that this also enabled going back to particular documents for reference when doing the programming exercises and tasks. Questions (in particular, questions that have someone *write* or *do* or *try* something) seemed more effective than only having participants read text. Participants who used the version of the tutorial with programming questions said they were helpful, and one participant who did not specifically stated that being able to try out the code would have helped learn and internalize the material more.

RQ3 asked about designing programming tasks that:

1. assess the ability to program effectively;
2. expose the types of bugs likely to be made by participants in the control condition

The Auction task was mostly straightforward except for part 4, in which everyone who attempted it made the same mistake of overwriting an asset. This would have caused the loss of a resource (money) if it had not been caught by the Obsidian compiler. This does seem to be a relatively common security bug, since every participant made the mistake at first. It therefore seems likely that Solidity participants in the user study would insert the same bug as well, but would not have the compiler to remind them of the error. This suggests that this task fulfills goal (2). The Pharmacy task required an understanding of how contracts interact with each other, as well as using states to prevent improper manipulation of objects. The task was intended to require higher-level insight from participants, so the instructions for the third part in this task do not explicitly lay out everything the participant needs to do. The task requires the participant to be able to understand Obsidian code (including the novel concepts of ownership and states). In the Gambling task (again only given to R3 and R5), the participants wrote 50+ lines of code. Though a small program, this is not a trivial amount of code, especially since they were able to design a program to follow the specifications accurately, implement their design correctly using concepts they had only just learned (ownership and states) and have their code compile using syntax that was new to them for this study. As a result, the Pharmacy and Gambling tasks seem to fulfill (1).

In this pilot study, we started to gather evidence that the new language features of Obsidian that allow for writing safer smart contracts can be used effectively by real users. The Auction task had participants use ownership to prevent the loss of a resource, and the Pharmacy task had participants use states to prevent incorrect manipulation of objects. All the participants who did these tasks were able to use ownership and states, and all of the more experienced ones used ownership and states to complete them correctly. Although the features of the language that allow for greater safety are more complicated, and one might assume make the language less usable, we did not observe this in these initial results. This will need to be more fully validated with additional participants in our user study.

6 Threats to Validity

The participants were a convenience sample of undergraduates studying computer science; this may not be representative of the general population of programmers. In the quantitative study, we hope to have a more diverse group of participants who more closely align with the

intended users of Obsidian. We only had six participants, and not all of them attempted all parts of the study. In addition, some of our participants had limited object-oriented programming experience. As a pilot study, our goal was to refine our study design and hypotheses, and for those purposes, we believe our approach sufficed.

7 Conclusion and Future Work

We designed and conducted pilot studies to find potential usability problems with Obsidian, its tutorial, and the tasks used to evaluate it. We created a tutorial that helps at least some participants learn Obsidian well enough to use it effectively. We also created programming tasks that test a participant's ability to use Obsidian competently and have the potential for participants to make the types of security bugs that are caught by Obsidian's compiler. We identified hypotheses for why some of our participants found it easier to learn and use Obsidian than others, and more generally, why picking up new languages is easier for some people than it is for others. We found that an incremental approach of teaching the language that included regular practice opportunities was most effective, but that participants with more programming experience appeared to be much more successful at completing the tasks.

In the future, we will conduct a quantitative user study to compare Obsidian to a control language with respect to usability and security, and will continue to make any necessary changes to our tutorial and programming tasks. In the process, we hope to evaluate our hypotheses regarding why some participants are more successful in learning the language and completing the programming tasks, with a focus on self-efficacy and experience.

References

1 Celeste Barnaby, Michael Coblenz, Tyler Etzel, Eliezer Kanal, Joshua Sunshine, Brad Myers, and Jonathan Aldrich. A User Study to Inform the Design of the Obsidian Blockchain DSL, 2017.

2 Michael Coblenz. Obsidian: A Safer Blockchain Programming Language. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 97–99, Piscataway, NJ, USA, 2017. IEEE Press. event-place: Buenos Aires, Argentina. doi:10.1109/ICSE-C.2017.150.

3 Stefan Hanenberg. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 933–946, New York, NY, USA, 2010. ACM. event-place: Reno/Tahoe, Nevada, USA. doi:10.1145/1869459.1869536.

4 B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer*, 49(7):44–52, July 2016. doi:10.1109/MC.2016.200.

5 Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. In *Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '04*, page 171, Leeds, United Kingdom, 2004. ACM Press. doi:10.1145/1007996.1008042.

6 Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, June 2003. doi:10.1076/csed.13.2.137.14200.

7 M. A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2):183–207, March 2000. doi:10.1016/S0167-6423(99)00036-2.

8 Dmitrii Suvorov and Vladimir Ulyantsev. Smart Contract Design Meets State Machine Synthesis: Case Studies. *arXiv:1906.02906 [cs]*, June 2019. arXiv: 1906.02906. URL: http://arxiv.org/abs/1906.02906.

9 Edward Zimudzi. Active learning for problem solving in programming in a computer studies method course. *Educational Sciences*, 3(2):9, 2012. URL: https://ubrisa.ub.bw/handle/10311/1170.

Type-Directed Program Transformations for the Working Functional Programmer

Justin Lubin
University of Chicago, Chicago, IL, USA
justinlubin@uchicago.edu

Ravi Chugh
University of Chicago, Chicago, IL, USA
rchugh@uchicago.edu

Abstract

We present preliminary research on DEUCE$^+$, a set of tools integrating plain text editing with structural manipulation that brings the power of expressive and extensible type-directed program transformations to everyday, working programmers without a background in computer science or mathematical theory. DEUCE$^+$ comprises three components: (i) a novel set of *type-directed program transformations*, (ii) support for *syntax constraints* for specifying "code style sheets" as a means of flexibly ensuring the consistency of both the concrete and abstract syntax of the output of program transformations, and (iii) a domain-specific language for specifying program transformations that can operate at a high level on the abstract (and/or concrete) syntax tree of a program and interface with syntax constraints to expose end-user options and alleviate tedious and potentially mutually inconsistent style choices. Currently, DEUCE$^+$ is in the design phase of development, and discovering the right usability choices for the system is of the highest priority.

2012 ACM Subject Classification Human-centered computing → Human computer interaction (HCI); Software and its engineering → Domain specific languages; Software and its engineering → Integrated and visual development environments

Keywords and phrases program transformations, structured editing, refactoring, code formatting

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.3

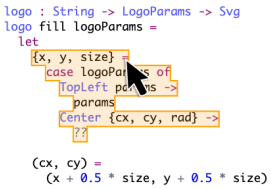
1 Introduction

As a medium for storing, transmitting, and interpreting information, text is as versatile as it is ubiquitous. Countless programs and interfaces operate and rely on text files, from the UNIX command line to nearly every programming language compiler. One particularly strong asset of text is its power to succinctly represent structured data in a way that is understandable to both humans and computers alike, as in CSV (comma-separated values) files, HTML (hypertext markup language) documents, and – the focus of this paper – programming language source files.

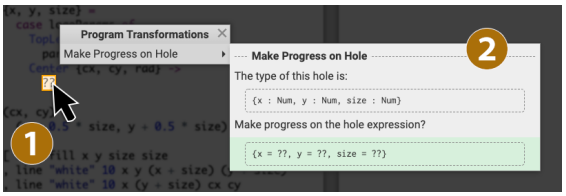
Unfortunately, this flexibility comes with a price: manual editing of structured text can be tedious and error-prone. On a basic level, one problem with manipulating structured text is that to do so requires knowledge of and adherence to rigid, static systems such as parsing and type checking. A single missed comma in a CSV file or improperly annotated variable in program source code can cause a complete failure on the part of the computer to interpret the text as the author intended. In the case of performing a nontrivial manual transformation on a program, the problem is exacerbated even further: programmers must also worry about the *runtime behavior* of their code and reason about changes in semantics (or lack thereof) that might be a result of their transformations.

10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019).
Editors: Sarah Chasins, Elena Glassman, and Joshua Sunshine; Article No. 3; pp. 3:1–3:12

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Structurally selecting a let-binding equation in DEUCE.



(b) Activating a program transformation in DEUCE by (1) structurally selecting an expression – in this case, a hole – and (2) interacting with the popup menu.

Figure 1 The DEUCE user interface.

Tools known as *structure* (or *projectional*) editors [9, 13, 24, 25] attempt to alleviate these difficulties by offering an interface that allows programmers to directly manipulate a *projection* of the underlying structure that is more faithful to the structure than is standard text. A major drawback of these systems is their reliance on non-standard file formats, and, as a result, their incompatibility with the large set of existing tools that operate on programs.

One attempt to reconcile the flexibility of plain text with the power of projectional editing is DEUCE [2, 8], a structure-aware code editor that operates on standard program source text, but augments the editing experience with direct manipulation capabilities for invoking relevant, automated program transformations. In DEUCE, the underlying structure of the program is revealed to the user via a set of overlaid polygons, as depicted in Figure 1a. After *structurally selecting* various parts of the program by pressing the shift key, hovering, and clicking on the polygons that appear, a "Program Transformations" menu appears that is automatically populated with a set of relevant transformations for the selected polygons, as depicted in Figure 1b. Hovering over the output of a program transformation previews it in the code panel, and clicking on the output updates the program with the transformed code.

While a good first step, DEUCE falls short in several regards. In particular, it has two main limitations:

(Limitation A) DEUCE only offers a relatively small number of ad-hoc program transformations; and

(Limitation B) the implementation of these transformations is tedious and non-compositional, requiring manual munging of abstract syntax trees annotated with low-level syntactic details such as whitespace.

To address these limitations, we propose and present initial work on a vast expansion of the DEUCE system – which we here call DEUCE$^+$ – with the goal of bringing expressive and extensible program transformations to the working programmer.

Paper Outline. In Section 2, we run through a high-level example demonstrating the desired (as-of-yet unimplemented) workflow of the DEUCE$^+$ system. Then, in Section 3, we explain the work that is underway to make that workflow possible and explicitly describe how the improvements we propose will address Limitations A and B. Finally, we conclude in Section 4 with a discussion of the further usability challenges that arise from the DEUCE$^+$ workflow.

In our quest to realize these goals, we assert the necessity of harmony between (i) text and structured editing, and (ii) sound mathematical foundations and usable, accessible, tools for everyday programmers without assuming extensive expertise in advanced functional programming or abstract mathematics.

Table 1 The database schema with some example entries.

Flag	Identity	Homepage
1	alice	example.net
1	bob	
0	carol@example.net	
1	dan	example.org
0	eve@example.edu	

2 The Deuce+ Workflow

Consider a website in which users register and have their information saved to a database. Originally, users could only register on this website with an email address, but this restriction was later relaxed to require only a username and, optionally, a homepage that lists further contact information. To encourage migration to the new username registration format, users may only list a homepage if they register with a username and *not* with an email. The database is structured into three columns, as shown in Table 1: the first tracks whether the user has registered with an email address (0) or a username (1), the second tracks the provided email or username, and the third tracks the (sometimes empty) user homepage.

To access this database, two library functions are provided:

```
identities : Database -> List (Either Username Email)
homepages : Database -> List (Maybe Homepage)
```

Using these two helpers, our task is to implement a function `show : Database -> String`.

We begin by taking a peek at our project's *code style sheet*, a mechanism provided by Deuce+ to allow users to customize preferences about stylistic choices in their programs:

```
.type-alias[type=tuple] { newlines: per-component; }
.tuple { max-size: 3; }
.top-level-definition { eta-reduction: basic; }
```

The first of these three rules tells us that, when aliasing a tuple type with a new name, each component of the tuple should be on its own line. The second rule ensures that only tuples be of (at most) size three are permitted, encouraging larger tuples to be replaced with records. The final rule indicates that "basic" eta reductions should be performed for top level definitions (no term rewriting beyond dropping arguments to functions). None of the program transformations in Deuce+ are aware of the particular code style in any given project, and, consequently, the authors of these transformations do not need to worry about adhering to style guidelines; Deuce+ handles the stylistic details automatically.

We now sketch out the skeleton of our program using standard text editing.

```
showEntries : (List (Either Username Email), List (Maybe Homepage)) -> List String
showEntries (idents, homes) =
  ??

show : Database -> String
show d =
  String.concat (showEntries (identities d, homepages d))
```

In the definition of `showEntries`, we use a *hole expression* [17] as a placeholder until we are able to make further progress implementing the function.

Immediately, we notice that the type annotation on `showEntries` is long and unwieldy. As in Figure 1a (but not shown here or in the rest of the code listings), we *structurally select* the tuple argument to the `showEntries` function and choose the Make Type Alias from Arguments transformation, entering "Entries" as the new type name.

```
type alias Entries =
  ( List (Either Username Email)
  , List (Maybe Homepage)
  )

showEntries : Entries -> List String
showEntries (idents, homes) =
  ??

show : Database -> String
show d =
  String.concat (showEntries (identities d, homepages d))
```

The new type alias automatically adheres to the specification from our style sheet that type aliases for tuples should have each component on a new line.

However, we are still struggling to make progress on the `showEntries` implementation because the types of our program alone do not ensure that the two lists passed into the function are the same length. In reality, we know that the state in which the two lists are of different lengths *should* be impossible. To take advantage of this fact, we can let the type system know it by structurally selecting the `Entries` type alias, choosing the Refine Type transformation, and entering the equality refinement `\xs -> length (first xs) == length (second xs)`.

```
type alias Entries =
  List (Either Username Email, Maybe Homepage)

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...

showEntries : Entries -> List String
showEntries entries =
  ??

show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> ??
```

The `Entries` type alias has been automatically updated to make the specified impossible state unrepresentable, the argument to `showEntries` has been switched from a (now-outdated) tuple pattern to a single variable name, and a new `convert` function has been introduced that is now used in the `show` function. The `convert` function translates our old `Entries` into the new representation, returning `Nothing` if the conversion fails. A hole expression is inserted in the `show` function to indicate that a default value is needed for when the conversion fails.

We now refactor the `Entries` type alias by structurally selecting the type argument to the `List` constructor and choosing the Introduce Type Alias transformation (which, as before, adheres to the code style sheet specification for type-aliased tuples). We also now provide a default value for `show` in the case of a conversion failure using standard text edits.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

type alias Entries =
  List Entry

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...

showEntries : Entries -> List String
showEntries entries =
  ??
```

```
show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> "Conversion failure!"
```

The `Entries` type alias as well as the `convert` and `show` functions are now complete, so we will omit their definitions until the final code listing for brevity.

We can now make progress on the hole in `showEntries`. The notion of "making progress" on filling in a hole is captured by the Make Progress on Hole transformation. We structurally select the hole in `showEntries` and select this transformation. Deuce$^+$ recognizes that we are trying to fill a hole of type `List String` and that we have an unused argument of type `Entries`, or `List Entry`. Accordingly, one of the options that the transformation presents to the user is the Map over List option with the additional parameter `entries` as the list to map over. We select this option, and the resulting code is automatically eta-reduced in accordance with our code style sheet.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      ??
  in
    List.map showEntry
```

We can make progress on the newly-created hole by structurally selecting it and the `entry` argument, choosing the Make Progress on Hole transformation, and choosing the Pattern Match option.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      case entry of
        (Left username, Nothing) -> ??
        (Left username, Just homepage) -> ??
        (Right email, Nothing) -> ??
        (Right email, Just homepage) -> ??
  in
    List.map showEntry
```

Using standard text editing, we fill in the holes on the branches of the case expression, but we get stuck on the last branch.

```
type alias Entry =
  ( Either Username Email
  , Maybe Homepage
  )

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
```

```
      case entry of
        (Left username, Nothing) -> showUsername username
        (Left username, Just homepage) -> showUsername username ++ "; see " ++ showHomepage homepage
        (Right email, Nothing) -> showEmail email
        (Right email, Just homepage) -> ??
  in
    List.map showEntry
```

We are stuck on the last branch because it is actually an impossible state! We structurally select the branch and choose the Make Impossible code transformation. We are presented with two redefinitions of the `Entry` type alias: a "normalized" and a "standardized" option. The normalized option results in a sum-of-products representation of the valid states, and the standardized option results in a definition in terms of built-in library types.

```
-- Normalized
type Entry
  = Ctor0 Username
  | Ctor1 Username Homepage
  | Ctor2 Email
```

```
-- Standardized
type alias Entry =
  Either (Username, Maybe Homepage) Email
```

Using other code transformations, we can swap one of these implementations for the other at any time, so we do not need to dwell too long on the decision right now. For now, we choose the standardized option. The transformation updates the implementation (but not signature) of the `convert` function as well as the case expression inside the `showEntry` helper.

```
type alias Entry =
  Either (Username, Maybe Homepage) Email

type alias Entries =
  List Entry

convert : (List (Either Username Email), List (Maybe Homepage)) -> Maybe Entries
convert = ...

showEntries : Entries -> List String
showEntries =
  let
    showEntry : Entry -> String
    showEntry entry =
      case entry of
        Left (username, Nothing) -> showUsername username
        Left (username, Just homepage) -> showUsername username ++ "; see " ++ showHomepage homepage
        Right email -> showEmail email
  in
    List.map showEntry

show : Database -> String
show d =
  case convert (identities d, homepages d) of
    Just entries -> String.concat (showEntries entries)
    Nothing -> "Conversion failure!"
```

The function is complete. The data and types have been munged by the code transformations into a cleaner, more manageable form that allows us to focus on the core logic of our program: converting entries in the database to strings.

3 Designing and Implementing Deuce+

To fully realize the workflow described in Section 2, Deuce+ must comprise three distinct but interrelated components:

(Section 3.1) a set of *type-directed program transformations* informed by common code authoring patterns of functional programmers;

(Section 3.2) a *syntax constraint language and engine* to ensure these transformations are composable and produce stylistically consistent output; and

(Section 3.3) a *domain-specific language* for specifying these transformations.

The first of these components will address Limitation A of Deuce by providing a sizeable set of transformations justifiably rooted in existing programmer behaviors, and the second and third of these components will address Limitation B of Deuce by providing a system for building and composing the transformations that is accessible to any user of the system. Moreover, with such tools in place, a large library of automatically composable user-defined transformations will be made possible, further combating Limitation A.

3.1 Type-Directed Program Transformations

With the strong guarantees of a rich type system, programmers can leverage expressive program transformations to alleviate some of the pains caused by the flexibility of text (as demonstrated by [11]). As part of the Deuce$^+$ project, we would like to formalize some of the common, implicit techniques that functional programmers use to develop programs by crafting program transformations to perform these authoring patterns automatically.

From personal experience, we have identified a few such transformations, including the Refine Type, Make Progress on Hole, and Make Impossible transformations from Section 2. However, we would like to design and conduct a user study to observe how functional programmers author code. Do they start with a skeleton of a solution and later fill in the holes? Or do they, perhaps, write code from the top down? There are myriad other ways code can be written, too, and there may not even be a consensus among functional programmers on this issue; nevertheless, a formal user study is in order to even begin to answer these questions. In the meantime, we investigate the three aforementioned program transformations and how they relate to some common functional programming authoring patterns, as summarized briefly in Figure 2.

Refine Type. The Refine Type transformation mirrors the code authoring practice of *maintaining code invariants*. After structurally selecting a type, users may activate the Refine Type transformation to narrow down the type's set of possible values. Inspired by (but not reliant upon) refinement types [6, 7, 23], the Refine Type transformation prompts the user for an invariant that they wish to maintain about the program, and attempts to refactor the type to ensure that the invariant is satisfied. For example, consider a functional queue type (left, below) drawn from Chris Okasaki's *Purely Functional Data Structures* [16]. Given the queue type on the left and the invariant that `|front| >= |back|`, Refine Type might suggest to transform the type to that on the right:

```
type Queue a =
  Queue
    { front : List a
    , back : List a
    }
```

```
type Queue a =
  Queue
    { frontBack : List (a, a)
    , remainingFront : List a
    }
```

The first `|back|` components of `front` and `back` are stored together in the list `frontBack`, and the remaining `|front| - |back|` elements are stored in the list `remainingFront`, so it is now impossible to represent a `Queue` in which `|front| < |back|`. (This transformation is a generalization of the approach used in Section 2 to ensure that the lengths of the two lists in the `Entries` type were equal.) To maintain ease of use and increase backward compatibility, the Refine Type

tool might also provide helper functions corresponding to the previous API of the queue:

```
front : Queue a -> List a
front (Queue q) =
  List.map Tuple.first q ++ remainingFront
```

```
back : Queue a -> List a
back (Queue q) =
  List.map Tuple.second q
```

Make Progress on Hole. The Make Progress on Hole transformation mirrors the code authoring practice of *"following the types,"* an oft-heard adage in the functional programming community suggesting that the type system can guide the user to implement the task at hand essentially automatically. While such advice is clearly not universally applicable, the statement does hold some truth to it, as demonstrated by the practice of type-directed programming [1, 26, 27].

After structurally selecting a hole, users may activate the Make Progress on Hole transformation to fill a hole with an expression (that will likely contain further, more specific, holes to fill in the future). There are at least three type-based approaches that this transformation can rely on to fill holes with helpful expressions: expression templates, type-directed refinement, and program synthesis. *Expression templates* are pre-written generic code snippets that can be suggested to the user to fill the hole at hand based solely on the bindings that are in scope and the type of the hole to be filled, as was done to introduce the `List.map` function in the implementation of the `showEntries` function in Section 2. *Type-directed refinement* is the systematic destructuring of a type into its component types via pattern matching, as was done to pattern match on the `entry` variable in the implementation of the aforementioned `showEntries` function. Another example of type-directed refinement would be, for instance, the filling of a hole of type `(a, a)` with the expression `(??, ??)`, an expression whose holes have been refined to simpler types (albeit at the cost of introducing a greater number of holes/subproblems). A final, more general approach to the task of filling holes lies in the practice of type-directed *program synthesis*, or, generating programs to match a specification (which, in this case, is the type of the hole along with any additional information – such as examples – that the synthesis algorithm may need), as in [5, 10, 12, 18, 19, 20, 21].

Make Impossible. The Make Impossible transformation mirrors the code authoring practice of *making illegal states unrepresentable* [4, 15]. At the time of authoring a type, certain code decisions may seem like a good idea that only later reveal themselves to be cumbersome. For example, a programmer might write a record type representing the state of an application window with the field `content : Maybe String` (where `Nothing` represents a closed window). Some time later, the programmer may realize that windows should save their own position, and thus adds a field `position : Maybe (Int, Int)` (when the window is closed, they reason, it has no position, so `position` must be a `Maybe` type). But, in enacting this change, the programmer has made representable two states that should be illegal: when either one of the fields is `Just something` and the other is `Nothing`.

By structurally selecting the record type and providing the patterns that should be unrepresentable (or by merely structurally selecting the branches in a case expression that should be unrepresentable), the programmer can use the Make Impossible tool to make values of the selected type that match the specified patterns impossible to represent. In the windowing example from above, the Make Impossible tool would transform the record of `Maybe` types to `Maybe { contents : String, position : (Int, Int) }`.

This transformation is made possible by the algebraic nature of datatypes, as used in [14]. While work is still underway to make the intuition rigorous, we may view the Make Impossible transformation on a datatype as operating on the polynomial functor of which

Refine Type	$\longleftrightarrow$	*maintaining code invariants*
Make Progress on Hole	$\longleftrightarrow$	*following the types*
Make Impossible	$\longleftrightarrow$	*making illegal states unrepresentable*

Figure 2 Summary of transformation and code authoring pattern correspondences.

the type is a fixed point, represented suggestively as $\tau - P$, where τ is the datatype functor and P is the pattern we wish to eliminate. For example, viewing record types as product types, we can determine the solution to the windowing example from above using algebraic laws related to this transformation (which are still under investigation):

`(Maybe String, Maybe (Int, Int))` $-$ `(Just _, Nothing)` $-$ `(Nothing, Just _)`

$$\rightsquigarrow (1 + s) * (1 + (i * i)) - s * 1 - 1 * (i * i)$$
$$= 1 * 1 + 1 * (i * i) + s * 1 + s * (i * i) - s * 1 - 1 * (i * i)$$
$$= 1 + s * (i * i)$$

$\rightsquigarrow$ `Maybe (String, (Int, Int))`.

3.2 Syntax Constraints

Implementing program transformations can be difficult and time-consuming work; it is clear that even just the three transformation described in Section 3.1 will require great thought and careful execution in their design and implementation. On top of the inherent complexity of the program transformation at hand, transformation authors – in the most basic setting – will also need to worry about routine but difficult-to-get-right considerations such as whitespace handling, declaration ordering, and naming conventions, among many other stylistic considerations. Moreover, sets of transformations authored by different people might all work with mutually incompatible styles, resulting in a poor user experience.

Automatic formatting tools eliminate these problematic choices by applying language- or project-specific concrete formatting rules to abstract syntax trees. Although formatting tools can be extremely valuable, they are typically limited to concrete syntax (even though many structural choices can be viewed as stylistic, too) and based on hard-coded rules that are not easily adapted to multiple configurations.

To address this problem, we propose the notion of code style sheets, or, the notion that stylistic choices for both concrete and abstract syntax be made automatically based on *syntax constraints*. Syntax constraints are an expressive means for encoding a variety of stylistic choices to be optimized to meet different requirements (e.g. line widths, stylistic preferences, and surrounding context). Much like the distinction between HTML, CSS, and JavaScript, syntax constraints specify code style completely independently of the code itself and any transformations that operate on it, reducing the burden of transformation authors while also ensuring a more consistent and flexible experience for users of these transformations.

When needed, Deuce$^+$ will feed the syntax constraints (along with unformatted code) into a solver engine, which will output formatted code. Presently, the exact nature of this engine as well as of the syntax constraints themselves are under investigation.

3.3 A Program Transformation Language

Even assuming that both concrete and abstract formatting are taken care of by the code style sheet engine, writing the transformation code for manipulating abstract syntax trees can be challenging. Speaking from the personal experience of implementing thousands of line of program transformations, it is difficult to maintain a declarative, consistent, and reusable pattern of implementation that scales well for even a few dozen transformations.

To resolve these issues, we propose a domain-specific language for program transformations that can operate on three different levels of abstraction: the concrete syntax tree, the abstract syntax tree, and the generalized syntax tree. The concrete syntax tree and abstract syntax tree are familiar: the former including rigid details such as the exact whitespace of the code to be transformed and the latter including only the underlying structure that is fed to, for example, the evaluator of the language. The *generalized syntax tree* operates at an even higher level than the abstract syntax tree, and allows for an even more declarative approach to specifying program transformations. It relaxes certain relationships between nodes in the abstract syntax tree so that, for example, an ordered list of let-bindings becomes an unordered set of let-bindings and function application operators (such as `|>` in Elm, OCaml, and F# and `$` in Haskell) are unified into a single function application node. Translating from the transformed high-level description of the program back to the concrete syntax that the programmer sees can be done by the use of the code style sheet engine, and the concerns of style and transformation logic are cleanly separated.

The program transformation language should also be able to interface with the code style sheet, exposing transformation-specific properties that allow the programmer to fine-tune how the transformation operates on a granular basis. Moreover, the language should eventually be amenable to *synthesis* of program transformations, reducing the barrier to authorship of transformations even further; indeed, the synthesis of program transformations is an exciting area of related (as in [22]) and future work. As with the syntax constraints and solver, much further investigation is needed to understand, design, and implement the generalized syntax tree specification as well as the domain-specific language as a whole.

4 Further Usability Challenges for Deuce$^+$

If the goal of Deuce$^+$ is to make powerful program transformations backed by elegant mathematical ideas accessible to – and authorable by – the everyday programmer, then good usability is of utmost importance. Every single operation described thus far should be influenced and rooted in not only sound mathematics, but proper usability studies. Although the development of Deuce$^+$ is currently very early in the design process, there are a few preliminary usability considerations that need to be addressed as Deuce$^+$ develops.

- With so many program transformations available to the user (especially including transformations authored by third parties), it is critical that the user be able to find the correct transformation for the task at hand, even if the user does not know its name. We will need to investigate what mental models users have of the transformations available to them and determine heuristics (likely relating to the structurally selected expressions) and display mechanisms (such as code diffs or animation) for showing them relevant program transformations.
- After a transformation has been selected, its output may be hard to decipher, so how does a user know when a program transformation is "correct?" We will need to investigate how to help users be confident (and correct) in their output selections.

- The languages underlying the program transformations themselves must also be streamlined and intuitive for end-user transformation authors. Drawing from interdisciplinary programming language design [3], we will need to design and evaluate the style sheet and transformation specification languages holistically, aiming for providing an experience that supports and encourages expressive and extensible code.
- The structured editing user interface introduced by Deuce will need to be improved to support usability improvements such as drag and drop, fluid and dynamic animations, and novel code overlays beyond simple polygons (such as for type information).

Completion of these tasks will ensure that – at every step of the way – the usability of Deuce$^+$ is of the highest priority across all its components, from its graphical user interface to its program transformation authoring and end-user experience.

References

1. Edwin Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2017. URL: https://books.google.com/books?id=eWzEjwEACAAJ.
2. Ravi Chugh. Structured Editing for Elm* in Elm, 2018. URL: https://www.youtube.com/watch?v=IcgmSRJHu_8.
3. Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, pages 133–146, New York, NY, USA, 2018. ACM. doi:10.1145/3276954.3276965.
4. Richard Feldman. Making Impossible States Impossible, 2016. URL: https://www.youtube.com/watch?v=IcgmSRJHu_8.
5. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837629.
6. Tim Freeman and Frank Pfenning. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. doi:10.1145/113445.113468.
7. Susumu Hayashi. Logic of refinement types. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 108–126, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
8. Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 654–664, New York, NY, USA, 2018. ACM. doi:10.1145/3180155.3180165.
9. JetBrains. JetBrains MPS. URL: https://www.jetbrains.com/mps/ [cited August 14, 2019].
10. Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 253–268, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314602.
11. Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, pages 182–196, San Francisco, California, USA, January 2008. URL: http://www.cs.kent.ac.uk/pubs/2008/2634.
12. Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Live Example-Directed Program Synthesis, 2019.
13. John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010. doi:10.1145/1868358.1868363.

14 Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. URL: http://strictlypositive.org/diff.pdf.

15 Yaron Minksy. Effective ML, 2010. URL: https://blog.janestreet.com/effective-ml-video/.

16 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

17 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. *PACMPL*, 3(POPL), 2019. doi:10.1145/3290327.

18 Peter-Michael Osera. *Program synthesis with types*. PhD thesis, University of Pennsylvania, 2015.

19 Peter-Michael Osera. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2019, pages 64–76, New York, NY, USA, 2019. ACM. doi:10.1145/3331554.3342608.

20 Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2738007.

21 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908093.

22 Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press. doi:10.1109/ICSE.2017.44.

23 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. doi:10.1145/1375581.1375602.

24 Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, September 1981. doi:10.1145/358746.358755.

25 Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 41–61, Cham, 2014. Springer International Publishing.

26 Geoffrey Washburn and Stephanie Weirich. Good Advice for Type-directed Programming Aspect-oriented Programming and Extensible Generic Functions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, WGP '06, pages 33–44, New York, NY, USA, 2006. ACM. doi:10.1145/1159861.1159867.

27 Stephanie Weirich and Liang Huang. A Design for Type-Directed Programming in Java. *Electron. Notes Theor. Comput. Sci.*, 138(2):117–136, November 2005. doi:10.1016/j.entcs.2005.09.014.

Designing Declarative Language Tutorials: A Guided and Individualized Approach

Anael Kuperwajs Cohen
Macalester College, St Paul, MN, USA
akuperwa@macalester.edu

Wode Ni
Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA
http://www.cs.cmu.edu/~woden/
woden@cs.cmu.edu

Joshua Sunshine
Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA
https://www.cs.cmu.edu/~jssunshi/
sunshine@cs.cmu.edu

Abstract

The ability to declare what a program should include rather than how these features should be implemented makes declarative languages very useful in many visual output programs. The wide-ranging uses of these programs, in domains ranging from architecture to web programming to data visualization, encourages us to find an effective method to teach them. Traditional tutorial systems are usually non-interactive and have a gap between the learning and application. This can leave the user frustrated without a way to move forward in the learning process. A general lack of guidance can lead the student down an incorrect path. To prevent these difficulties, we propose a guided tour followed by novel question types that both direct the student's learning and creates a focused environment to practice individual skills. Lastly, we propose a study to test the hypothesis that this tutorial is quicker to complete and results in a greater understanding of the declarative language.

2012 ACM Subject Classification Applied computing → Interactive learning environments

Keywords and phrases Declarative Programming, Programming Language Tutorial, Visualizations

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.4

1 Introduction

Declarative languages have been successful in many domains because of the multiple advantages they possess. The readability [1], succinct composition, and unordered nature of the code can make them easier to use [6]. For creating programs with visual output, declarative languages are especially prevalent. There are many examples that are widely used, including HTML, CSS, D3, and others. The reason for that is the ability to declare what the aspects of the visualization should be rather than how they should be built [11]. Due to the common usage of declarative languages and their prominence with visualizations, we are investigating how to effectively teach a declarative language. We are conducting this investigation within the context of a mathematical diagramming and education system called Penrose, which utilizes a declarative language, Substance, that resembles standard mathematical notation. An uncomplicated, accessible way to learn Substance would support the system overall. As a solution to our primary research question, we propose using a guided tour followed by a series of novel question designs that provide targeted, focused application practice.

A guided tour is a context-sensitive tutorial that uses constraints and checkpoints to guide the user through the different aspects of the program. Many video games use guided tours instead of multi-page manuals because there is less frustration among users [2]. Each time there is a new skill to be learned, the tutorial will show the player how to complete

the action through instruction and practice, followed by a checkpoint. The tasks are smaller pieces of the overall game, so they build upon each other to complete the tutorial. Both video games and declarative languages consist of many small pieces and skills that build up into proficiency, therefore it is likely a guided tour will be an effective strategy for declarative languages. The possible downfalls that this method avoids, however, are seen with traditional programming language tutorials. The mixture of textual instructions followed by exercises leads to a gap between learning and applying. Removing this gap should result in faster learning [7].

To secure a smooth transition into using the system freely, there will be practice questions that follow the guided tour. These questions come in a set of novel designs that are more focused on honing individual skills. Typical models, on the other hand, have exercises that slowly build to an activity that encompasses all of the learned skills. This does not take into account the needs of the student and which areas require more practice. Providing targeted feedback through specific questions, with the appropriate level of difficulty, will ensure the student effectively learns all the necessary skills for using the declarative language [12].

Along with this design, having a visualization as an aid can be extremely useful in the field of education. Due to the undeniable connection between visual outputs and declarative languages, the educational benefits of diagrams are important as well. External representations, such as diagrams, portray the information in an alternative way and assist the learning process. Often, seeing the information visually, as opposed to textually, promotes better reasoning and problem solving [9]. Additionally, inferring information from visuals is easier due to their emergent properties [8]. When visualizations are paired accurately with a student's current capabilities and the task at hand, which is the aim of the practice problems, there are cognitive benefits that enhance learning [3].

In the following sections we introduce our ongoing efforts of designing a tutorial for declarative languages. We start by presenting the design of a guided tour that teaches the declarative language of Penrose, Substance. Following that, we will show the designs of the novel problem types that provide focused but unconstrained practice within the Penrose system. Finally, we propose a user study that evaluates our design.

2 Tutorial Design

A successful tutorial ensures that the student can understand and utilize the content they just learned. Teaching a language, specifically, entails covering all aspects of the language that might be used afterwards. Programming languages often have formal descriptions of the grammar that it implements, such as the BNF. When looking at a declarative, domain-specific language, however, the size of the grammar is smaller, limiting the space of possible programs. Our system can leverage an explicit specification of the language constructs to automatically

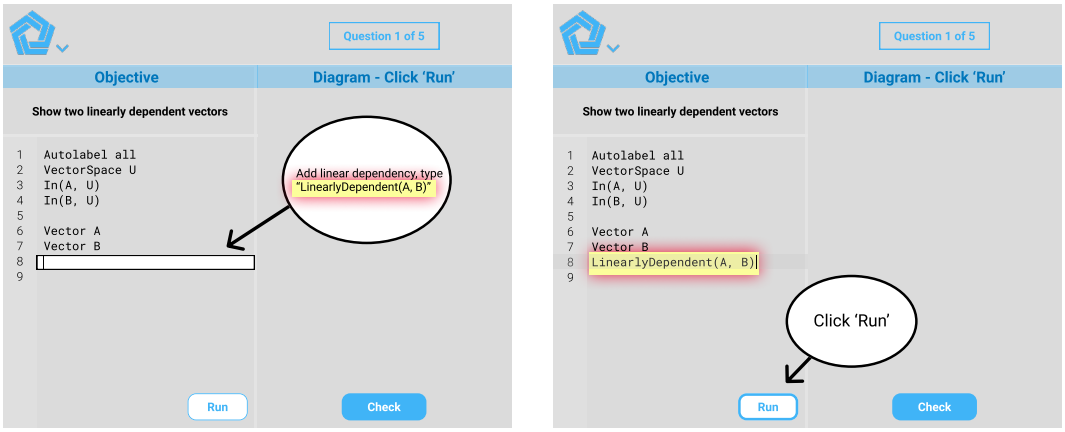
```
-- Type Constructors

type VectorSpace
type Vector

-- Predicates

predicate In: Vector * VectorSpace V
predicate LinearlyDependent: Vector * Vector
```

Figure 1 Domain Program.

generate instructional content such as guided tasks and practice problems. The system derives examples and counterexamples from the high-level grammar to create practice problems that target a specific language construct. In our example, Penrose defines a mathematical domain with a Domain program written in a simple metalanguage that declares all the types and operators available (see Figure 1). By sampling the space of possible Substance programs defined by Domain, the system generates various types of diagrammatic practice problems.

We have not yet generated problems from Domain and there are many open research questions to solve to do so effectively. For example, when generating incorrect answers for multiple choice questions we will want to replicate plausible errors that students might make. Addressing these research questions is future work.

With our system, the guided tour consists of a set of tasks that covers different parts of the domain. Each task is broken up into a series of small steps. In order for the user to learn exactly what is intended, we guide the user through each step and constrain how the user can interact with the interface to ensure completion. We direct the user's attention towards a particular part of the question to show the next step via arrows, changes to the opacity of components in the user interface, and precise directions. Furthermore, the user cannot proceed without completing each step.



(a) Guided Tour Step 3 (b) Guided Tour Step 4

Figure 2 Guided Tour.

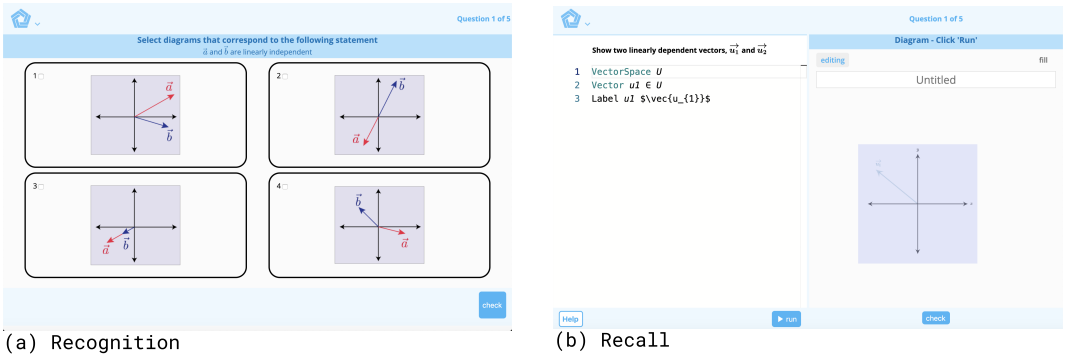
The purpose of using instructions at every step is to avoid frustration. Without personal assistance, advancing in the face of difficulties is a challenging task. In a system such as Penrose, where the purpose is creating mathematical diagrams with ease, we aim to support users without any programming experience. Our goal in the guided tour is that the user should know exactly what they need to do at each step, and thereby increase retention rates [13]. At the same time, combining instruction with the physical task of writing code aids retention [2].

An example task for the guided tour (see Figure 2) has the prompt: "show two linearly dependent vectors", separated into smaller steps:

- Add the first vector, type `Vector A`
- Add a second vector, labeled `B`
- Add linear dependency, type `LinearlyDependent(A, B)`
- Click **Run**
- If the diagram looks linearly dependent, click **Check**

As seen in Figure 2, the opacity of the screen is lower, except for the instructions and the spot where the arrow is pointing. The instructions are contained within a boldly outlined ellipse, and the arrow points to where those instructions should be applied, which is also outlined. All these elements directs the attention of the user.

After progressing through the tasks, users naturally continue practicing with a set of novel question types. These questions utilize the PENROSE system, as described earlier in this section, and visualizations to emphasize the learning that occurs during the guided tour. The questions are focused on practicing specific skills and target the areas where the student is not as strong. The two categories of questions include recognition and recall.



(a) Recognition (b) Recall

Figure 3 Novel Question Types.

Recognition is a category of problems that requires the student to recognize associations between the program outputs and the source programs. The methodology would be analyzing and working with the diagrams from the system. One type of problem that falls into this category is a multiple choice question where users match the code or prompt to the correct diagram (see Figure 3a). The prompt might ask for two linearly independent vectors, where two out of the four options are correct. Another question type is adjusting a diagram to match the code that is provided, meaning the user corrects the error. An extension of this problem is building the diagram from a bank of shapes.

Recall questions strengthen a student's ability to recall language constructs of the DSL. This involves a diagram that is already produced and constructing the answer. Our design supports this in two ways, correcting existing code and writing new code. For the former, there is an error in the code that the user has to find and fix. The latter is starting from the beginning and writing the code that answers the prompt (see Figure 3b). If the prompt requests two linearly dependent vectors, the user must write the code that creates that diagram.

3 Study Design

Implementing a study could provide evidence that supports this tutorial design. The purpose of an evaluation is to show the benefits of the previously described design, mainly that it is a faster and more efficient way of learning a declarative language. We hypothesize that this method will increase the understanding of the user.

Our tentative evaluation plan consists of students with a range of computer science backgrounds split into two groups, control and experimental. The control group learns SUBSTANCE with textual instructions, and have full access to the system. The experimental group uses the guided tour and the novel question types. The dependent measures are the

time it took to complete the tutorial and the understanding, which is assessed through a quiz. The quiz will involve a combination of answering questions about example SUBSTANCE programs and writing SUBSTANCE code. This will be evaluated for accuracy. Every participant completes a pre and post-survey, followed by a debrief of the study.

4 Related Work

There are two main categories of related work that are correlated to our research question: visual learning and language tutorials.

Visual Learning

One focus of visual learning is how to improve visualizations to help students the most. The question's presentation has one of the largest impacts [3]. For example, effective instructions displayed alongside the external representation greatly increases understanding. Furthermore, the student must have enough experience to fully utilize the diagram. Grounded feedback is one way to use a student's prior knowledge [12], where they solve problems using a symbolic representation followed by the answer presented in a feedback representation. This second representation is a more concrete, familiar visualization and encourages students to interpret their answer's accuracy. Grounded feedback is a great tool for teaching, yet making the feedback individual is difficult to implement.

Visualizations can also be used as examples. Along with visual learning, example-based learning has become more common within computer science [5]. Unfortunately, visualizations and examples can be ineffective if they do not engage the student [10].

Language Tutorials

The most common way to learn a new language is through non-interactive methods, such as written text and videos, similar to Khan Academy.[1] Often, practice exercises and do-it-yourself tasks follow the explanations, leaving a gap where information can be forgotten. More interactive tutorials include websites such as Scrimba[2] and A Tour of Go,[3] that mix textual instructions and videos with practice questions. In Scrimba, students can interact with the code as the video is playing, making it easier to test out concepts as they are being taught. A Tour of Go is similar, but with textual instructions instead.

DrScheme [4] (now known as Racket) and the stencils-based tutorial for Alice [7] are examples of teaching within programming environments. DrScheme's purpose is to easily correct the errors many users run into. Experienced users have created work-arounds for these problems, but beginners get stuck. The stencils tutorial for Alice is similar to the guided tour, but focuses on teaching the system rather than a declarative language. This tutorial uses translucent stencils that direct attention to a hole that is regularly colored. Virtual sticky notes display the instructions. After comparing their new tutorial to a traditional one, they found that fewer errors were committed and it was a faster and easier experience overall. The main downside is that users who completed the stencils tutorial were less confident they could work with the program, even though they were more confident that they completed the tutorial correctly.

[1] https://www.khanacademy.org
[2] https://scrimba.com
[3] https://tour.golang.org

5 Conclusion

Since declarative languages are so widely utilized by users of all skill levels, teaching them efficiently is an important problem to investigate. Combining a guided tour and practice problems with new designs that focus on building individual skills suggests a more effective method. A future study could confirm this hypothesis, by looking for how fast the tutorial is completed and overall understanding of the content. This would improve the visual output systems that depend on declarative languages by providing more guidance for students without requiring personal assistance.

References

1 UW Interactive Data Lab | Papers. URL: http://idl.cs.washington.edu/papers/d3/.

2 Erik Andersen, Eleanor O'Rourke, Yun-En Liu, Rich Snider, Jeff Lowdermilk, David Truong, Seth Cooper, and Zoran Popovic. The impact of tutorials on games of varying complexity. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*, page 59, Austin, Texas, USA, 2012. ACM Press. doi:10.1145/2207676.2207687.

3 Julie L. Booth and Kenneth R. Koedinger. Are diagrams always helpful tools? developmental and individual differences in the effect of presentation format on student problem solving. *The British Journal of Educational Psychology*, 82(Pt 3):492–511, September 2012. doi:10.1111/j.2044-8279.2011.02041.x.

4 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12:159–182, 2002. doi:10.1017/S0956796801004208.

5 Sumit Gulwani. Example-based Learning in Computer-aided STEM Education. *Commun. ACM*, 57(8):70–80, August 2014. doi:10.1145/2634273.

6 J Heer and M Bostock. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, November 2010. doi:10.1109/TVCG.2010.144.

7 Caitlin Kelleher and Randy Pausch. Stencils-based tutorials: design and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '05*, page 541, Portland, Oregon, USA, 2005. ACM Press. doi:10.1145/1054972.1055047.

8 Kenneth R. Koedinger. Emergent properties and structural constraints: Advantages of diagrammatic representations for reasoning and learning. In *Proc. AAAI Spring Symposium on Reasoning with Diagrammatic Representations*, pages 154–169, 1992. URL: http://www.aaai.org/Papers/Symposia/Spring/1992/SS-92-02/SS92-02-031.pdf.

9 Jill H. Larkin and Herbert A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1):65–100, January 1987. doi:10.1016/S0364-0213(87)80026-5.

10 Thomas L Naps, Rudolf Fleischer, Myles McNally, and Alma College. Exploring the Role of Visualization and Engagement in Computer Science Education.

11 Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 669–678, New York, NY, USA, 2014. ACM. event-place: Honolulu, Hawaii, USA. doi:10.1145/2642918.2647360.

12 Eliane S. Wiese and Kenneth R. Koedinger. Designing Grounded Feedback: Criteria for Using Linked Representations to Support Learning of Abstract Symbols. *International Journal of Artificial Intelligence in Education*, 27(3):448–474, September 2017. doi:10.1007/s40593-016-0133-9.

13 A. Yan, M. J. Lee, and A. J. Ko. Predicting abandonment in online coding tutorials. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 191–199, October 2017. doi:10.1109/VLHCC.2017.8103467.

Human-Centric Program Synthesis

Will Crichton
Stanford University, Stanford, CA, USA
wcrichto@cs.stanford.edu

Abstract

Program synthesis techniques offer significant new capabilities in searching for programs that satisfy high-level specifications. While synthesis has been thoroughly explored for input/output pair specifications (programming-by-example), this paper asks: what does program synthesis look like beyond examples? What actual issues in day-to-day development would stand to benefit the most from synthesis? How can a human-centric perspective inform the exploration of alternative specification languages for synthesis? I sketch a human-centric vision for program synthesis where programmers explore and learn languages and APIs aided by a synthesis tool.

2012 ACM Subject Classification Software and its engineering → Programming by example

Keywords and phrases Program synthesis, programming by example, PL/HCI

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.5

Acknowledgements I would like to thank my advisor Pat Hanrahan for his everlasting support despite my constantly evolving research direction. And a big thanks to Brian Hempel, Georgia Gabriela Sampaio, and my anonymous reviewers for constructive comments that substantially improved the quality of this paper.

1 A Story of Our Time

Consider the story of Dana the Data Scientist. At Sonmanto, her agritech business, Dana wants to analyze the weekly seed production. Opening a Jupyter notebook, she creates a new code cell, imports a few libraries, and sends off a SQL query to build a Pandas dataframe. Knowing the Pandas API from her data science course and past experience, she computes the week's average seed production using standard dataframe methods.

```
query = sql("SELECT time, production FROM seed_production ORDER BY time")
df = pd.read_sql_query(query)
df.where(df.time >= dt.now() - dt.timedelta(days=7)).production.mean()
```

Concerned that the week's production seems low, Dana wants to see a 7-day rolling average of the last year's production to put this week into context. She has never computed a weekly rolling average before, so she Googles "pandas rolling average". Excellent, Pandas has a `Dataframe.rolling` method, but... it doesn't do quite what she wants. All the examples use windows that contains a fixed number of elements, but she wants windows of a fixed duration potentially containing different numbers of production samples.

Dana continues searching increasingly elaborate queries like "pandas rolling average date dynamic window", and eventually finds some StackOverflow answers that look almost right. However, all of their solutions either use abstract notation like "foobar" or were made for other domains like stock trading. Dana finds it difficult to see the relationship between finance problems and seed production. After twenty minutes of searching, she gives up with a resigned sigh and decides to just implement it in plain Python.

```
for day_start in pd.date_range(df.time.min(), df.time.max()):
    day_end = day_start + datetime.timedelta(days=7)
    window = [row.production
```

```
            for day in pd.date_range(day_start, day_end)
            for _, row in df.iterrows() if row.time.date() == day.date()]
    weekly_prod.append(pd.Series(window).mean())
```

Dana knows the code isn't beautiful, but it gets the job done. Glancing back at the StackOverflow questions, she starts to see the connections after going through her own implementation. But it's close to lunch, and she spent too long on this already. Simplifying the code is a task for another day, and she moves on.

2 A Story of Another Time

. . . Concerned that the week's production seems low, Dana wants to see a 7-day rolling average of the last year's production to put this week into context. She has never computed a weekly rolling average before, so she Googles "pandas rolling average". Excellent, Pandas has a `Dataframe.rolling` method, but. . . it doesn't do quite what she wants.

Rather than continuing to search fruitlessly, she writes down her plain Python solution. Dana highlights the code cell and clicks "Synthesize" in her Jupyter toolbar, opening a dialog box on the side. She writes `Dataframe.rolling` and `Dataframe.mean` into the box, knowing those are likely going to be important parts of a Pandas-specific solution if it exists. Guided by her suggestions, the synthesis engine finishes in under a minute, producing a `rolling` solution contextualized to her dataframe.

```
df.sort_values('time').set_index('time').rolling('7d').mean()
```

Ahh, the rolling function has a special syntax for time windows. But, Dana wonders, what does each part do? Hovering over each part of the program, the synthesis tool uses its counterexamples to shows what would happen if a given method call was omitted or changed. Removing `sort_values` or `set_index` cause the program to raise an error. Changing the window to `rolling(7)` produces an incorrect output.

Plotting the values in Matplotlib, Dana marvels at the simplicity of the solution. She starts to wonder: are there other places in Sonmanto's code base where they could use this pattern? Glancing over at the clock, there's still an hour to lunch, great! Highlighting her old code cell once more, she clicks "Find Similar" to search her notebooks and text files for snippets that look structurally similar to the one she just wrote.

After the search engine returns five plausibly similar programs, Dana runs the synthesis engine in parallel on each one. Noticing that most of the snippets were written by Danny the Data Engineer, she motions Danny over and teaches him about the feature she just learned.

3 The Past and Present of Program Synthesis

The stories above highlight a key fact about modern-day programming: programmers routinely deal with dozens of representations of code and data. In the data science domain, Jupyter notebooks swap between explanation and code. Data flows from SQL databases to Python lists to Pandas dataframes. Operations mix and match bespoke APIs with general-purpose programming constructs. Programmers are continually learning new representations as languages, libraries, and tools emerge and change.

Dana's struggles with Pandas show a prototypical case of acquiring a new representation. Not knowing how to compute a specific kind of rolling average, she uses a combination of documentation, code examples, and prior knowledge to understand whether the Pandas API can solve her problem. Having general programming skills, she can arrive at a standard Python

solution, but not the more concise API-specific solution. As the second story demonstrates, I believe that program synthesis techniques hold promise in helping programmers overcome these kinds of representational transfer problems (or refactoring, migration, etc.). Yet, to date, such a story is still a fantasy.

To understand why, we will briefly examine the history of program synthesis. Synthesis has been predominantly applied in the context of programming-by-example (PBE). In PBE, a user provides examples (input/output pairs) of a pure function, and the synthesizer attempts to find a "good" (e.g. small) function that satisfies those examples. Often, the user is an end-user manipulating spreadsheets or text documents, and the generated program is an invisible macro. Through hard-earned experience with dozens of PBE systems, researchers have both articulated design principles of PBE [8] and ultimately produced the flagship commercial synthesis engine, Excel FlashFill [11]. This effort succeeded in part by a human-centric push to understand both the applications where PBE was most valuable, and the essential usability constraints for real-world usage.

The central question of this paper, then: what does human-centric synthesis look like beyond PBE? Specifically, what applications open up when a user has the programming skills to express specifications at a level beyond examples? Traditionally, these kinds of tasks have been viewed as refactoring or migration, where the existing codebase specifies the desired behavior for the transformed one. Historically, refactoring tools could only perform simple syntactic changes like renaming types or methods. However, recent synthesis tools have shown striking progress in translating between complex high-level representations of code. For example, tools can move between languages like Java → Spark [1], and Fortran → Halide [5]. Tools can refactor APIs, like parallelizing Java streams [7], adding default methods in Java [6], updating SQL queries after schema changes [14].

These approaches have significantly advanced the state-of-the-art in program synthesis techniques. But given the lack of meaningful commercial adoption, it is unclear whether they're trying to solve the right problem. Certainly this fact arises in part from the general difficulty of tech transfer. But this would not be the first time the PL and compilers communities have been led astray by the allure of automating high-level tasks for programmers.

For example, modern compilers do register allocation by solving a complex graph coloring problem with zero user input, and no one takes issue with that. History and experience suggest that deciding which temporary is assigned to which register or stack slot is not a meaningful decision for a programmer. By contrast, decades of research have been invested into automatic parallelization of general-purpose code, like arbitrary C for-loops. However, identifying and expressing parallelism in an application seems more fundamental/important for the programmer to decide than register allocation. Automatic techniques have been ultimately eclipsed by DSLs with understandable, user-programmable models of parallelism like Halide [12] and Spark [16]. After all, autovectorization is not a programming model [10]. So how can program synthesis avoid a similar fate?

4 A Vision for Human-Centric Program Synthesis

The moral of these stories is that understanding the context, challenges, and capabilities of the working programmer is essential for improving the programming experience. Applying such a human-centric lens in designing and evaluating synthesis tools could accelerate the progress of synthesis research and promote the real-world adoption of these techniques. While the goal of this paper is primarily to spark discussion – what do you think human-centric synthesis entails? – I want to set the stage by articulating my principles for improving synthesis tools.

1. **Synthesis tools should use a user's most productive specification language.**
   Input/output pairs have been a popular specification language for synthesis, since "PBD is a natural match for artificial intelligence... by observing the actions taken by the user (training examples), the system can create a program (learned model) that is able to automate the same task in the future." [8] Moreover, for end-users without training in formal languages, I/O pairs are the highest level of abstraction at which they can formally specify behavior. However, programmers can use a diverse array of representations for specifications. These range from testing (e.g. unit testing, randomized test generation [2]) to declarative languages (PlusCal, UML) to programming languages (sketches [13]).
   Synthesis tools should use a specification language based on the difficulty of writing abstract rules versus writing individual examples in a given domain. Dana's window query may be easier for her to specify in Python, while a data structure manipulation like rotating a tensor may be easier to specify by examples. Human-centric evaluations of synthesis should seek to empirically characterize this trade-off.

2. **The synthesized program can not be a black box.**
   Synthesis tools have historically been used like compilers: input the specification, and don't look at the output program, just run it. Again, while this approach works for end-users who may lack the technical knowledge to understand the synthesized program, such an interaction mode is rarely desirable for a programmer. Programs are written, re-read, tweaked, maintained, handed off to other collaborators, and so on. Professional programmers spend only 5% of their time writing code [9, 15].
   Subsequently, Programmers must be able to comprehend and maintain synthesized programs. A synthesis tool should generate readable code and be able to explain its decisions, like Dana's imagined UI. Readability metrics can be informed by existing principles of programming notation design, like the cognitive dimensions framework [4].

These principles have helped me envision application spaces beyond the traditional purview of synthesis, like those characterized in the stories above. For example:

1. **Helping programmers learn new languages and APIs.** Programmers, whether hobbyists or full-time developers, encounter learning opportunities every time they code. Dana's was intentional: she realized she didn't know a feature and searched for it. But many more opportunities are passed by due to lack of awareness of a language or API feature. Anecdotally, I know Rust users that say the linter (Clippy) has helped them learn APIs through simple syntactic patterns. A synthesis tool as an extremely powerful linter could identify when someone likely doesn't know a concept ("there are 10 places in your code base that could be simplified with a for-loop"), highlight relevant code, and even suggest the translation if possible. By explaining its code-generating decisions, a synthesis tool can move beyond code that just works, to code that teaches how it works.

2. **Evaluating the impact of an API/language change.** When maintainers of libraries and languages debate new features, questions arise like: how many people would use this change? Would their code be meaningfully improved with this feature? For example, the Python community recently accepted the contentious PEP 572 "walrus operator." Guido was ultimately convinced by maintainers who combed through their own codebase, demonstrating dozens of places where the proposed feature could be applied [3]. A synthesis tool could help maintainers automate such exploratory tasks and more freely experiment with proposed designs.

Enabling these applications raises a number of exciting research questions in the design, implementation, and evaluation of synthesis tools. If high-level specifications replace I/O pairs, does this reduce the program search space, or is it just a means of generating examples (like QuickCheck)? Can API or language designers make their systems more amenable to synthesis? I hope that perspectives from the PL/HCI community can contribute greatly to these endeavors.

References

1. Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1205–1220. ACM, 2018.
2. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
3. Jake Edge. The pep 572 endgame. `https://lwn.net/Articles/759558/`, 2018.
4. Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
5. Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.
6. Raffi Khatchadourian and Hidehiko Masuhara. Automated refactoring of legacy java software to default methods. In *Proceedings of the 39th International Conference on Software Engineering*, pages 82–93. IEEE Press, 2017.
7. Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. Safe automated refactoring for intelligent parallelization of java 8 streams. In *Proceedings of the 41st International Conference on Software Engineering*, pages 619–630. IEEE Press, 2019.
8. Tessa Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–65, 2009.
9. Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE Press, 2015.
10. Matt Pharr. The story of ispc. `https://pharr.org/matt/blog/2018/04/18/ispc-origins.html`, 2018.
11. Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, pages 107–126. ACM, 2015. `doi:10.1145/2814270.2814310`.
12. Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*, pages 519–530. ACM, 2013. `doi:10.1145/2491956.2462176`.
13. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.
14. Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300. ACM, 2019.
15. Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.
16. Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Is a Dataframe Just a Table?

Yifan Wu
UC Berkeley, Berkeley, CA, USA
yifanwu@berkeley.edu

Abstract

Querying data is core to databases and data science. However, the two communities have seemingly different concepts and use cases. As a result, both designers and users of the query languages disagree on whether the core abstractions – dataframes (data science) and tables (databases) – and the operations are the same. To investigate the difference from a PL-HCI perspective, we identify the basic affordances provided by tables and dataframes and how programming experiences over tables and dataframes differ. We show that the data structures nudge programmers to query and store their data in different ways. We hope the case study could clarify confusions, dispel misinformation, increase cross-pollination between the two communities, and identify open PL-HCI questions.

2012 ACM Subject Classification Information systems → Relational database query languages; Software and its engineering → Software usability; Software and its engineering → API languages

Keywords and phrases Usability of Programming Languages

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.6

Funding *Yifan Wu*: NSF 1564351

Acknowledgements Thanks to my advisor Joe Hellerstein for the inspirations and to Devin Petersohn, Michael Whittaker, Remco Chang, Wenting Zheng, and Eric Liang for their valuable and kind feedback.

1 Introduction

Querying data is ubiquitous – application programmers query data to show relevant information, analysts query data to answer business questions, and scientists query data to find patterns for formulating and testing hypothesis. The uses cases are addressed by different communities.

Application programmers and business analysts are traditionally served by databases. During the 1970s, Codd's seminal paper defined a set of relational algebra over tables. A few years later, IBM developed SQL, a declarative language that can express the algebra. Since then, SQL has become the standard for database management systems. The data scientists are served by more general purpose programming environments like R and Python. Instead of using tables, they use *dataframe*, as seen in *pandas* (Python) [7], R [11], and Spark [13].

Database researchers find dataframes odd. A well-known database researcher, Joe Hellerstein, commented on Twitter in 2016, *Stop. A "data frame" is just a table. Thank you* [19]. In the tweet thread, Leo Meyerovich, a PL researcher, suggested that the systems must have been built for a reason. He found that dataframes helps with a *clean curation of basic DB, HPC, PL, etc. ideas.* Two years later, the difference in opinion continues – when prompted with whether the tweet has "aged well", Joe replied, *A "data frame" is a messy conflation of relations and matrices that wouldn't have evolved in a well-typed language, and which complicates the relevant algebraic operations involved.* [20].

While the database side considers dataframe APIs messy, the dataframe side seems to consider the database APIs inconvenient – from the creator of pandas, Wes McKinney, "[pandas] is what people use for data ingest, data prep, and feature engineering for machine learning models. The existence of other database systems that perform equivalent tasks isn't useful if they are not accessible to Python programmers with a convenient API." [3]. We see from the comment that the question of whether a dataframe is the same as a table is as much about the data structures themselves as it is about the API design, which, as well will show, is often influenced by the data structures both in terms of technical limitations as well as mental models – to understand tables and dataframes, we also need to understand accompanying languages like SQL and pandas.

Not only do system designers have different opinions, users of the query systems also have different views. On a Stack Exchange post, the first search result in Google for "pandas vs. sql" as of August 2019, the second most upvoted post claims that the comparison is "apples to oranges" [10].

Understanding the disagreement is important for language and library designers. Not understanding what functionalities are different or what features are desirable prevents the communities from learning from past lessons and leveraging existing techniques. It may also be confusing to the users to be presented with inconsistent messages and ideas. In this article, we evaluate the differences of the data structures of tables and dataframes, their mental affordances, the operations available, the mediums by which the operations are expressed and the effects on programming experience. We will take a bottom up approach, investigating existing languages – primarily SQL and Python pandas – to draw out relevant concepts and questions. Along the way, we will identify open questions for future work and speculations about the approaches.

2 Data Structures and Operations

The tabular format has existed for a long time as a way of organizing information, dating at least to the *Almagest* almost two thousand years ago [27]. It is no wonder that both the database and data science community have found the format useful. Despite sharing a similar tabular look, tables and dataframes are defined as different data structures and have different operations available. In databases, a table is a *set* of records (rows)[1]. A table is also called a *relation*. The *relational algebra*, proposed by Codd, defines the transformations available to these relations [17]. The design of relational algebra protects users from needing to know how the data is organized in the machine, and makes it possible for users to specify high-level queries, and leads to an inexhaustible number of optimization techniques. In data science, there is more than one definition of a dataframe, listed in Table 1. In the rest of this section, we explore what the definition means for the users of these languages[2].

[1] In practice, tables are often *multisets*, which means that there can be duplicate values. We do not go into a detailed discussion here since the key difference being investigated is that of order.

[2] We will use "language" uniformly to discuss both languages and libraries for simplicity.

Table 1 Definitions of dataframe.

lib/lang	definition
pandas	two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects [7].
R	tightly coupled collections of variables which share many of the properties of matrices and of lists [11]
Spark	equivalent to a table in a relational database [13]

2.1 Set v. Lists

Relational algebra is all about (multi)sets. As a result, the programmer cannot rely on the relative position of the row or column in the table. Dataframes are all about lists[3], where programmers can use the relative positions. As a direct consequence, many operations that are commutative with tables are not for dataframes. For example, concatenating dataframes in pandas takes in a list, whose order, if changed, returns a different result. Take `df12 = pd.concat([df1, df2])` and `df21 = pd.concat([df2, df1])`, `df12.iloc[0]` may not equal `df21.iloc[0]`. In SQL, union-ed tables are the same regardless or the order – `(SELECT * FROM df1) UNION (SELECT * FROM df2)` and `(SELECT * FROM df2) UNION (SELECT * FROM df1)` are the same.

The affordance of order has a big impact on how programmers think. First, being able to rely on *order* makes basic operations like finding a row with values of a certain rank much easier. For instance, if a list is already sorted by, say, `income`, finding the row with the maximum `income` (i.e. rank of 1) is as simple as selecting the first item `df.iloc[0]`. SQL is more verbose: `SELECT * FROM df WHERE income = (SELECT MAX(income) FROM df)`, or less idiomatically, `SELECT * FROM df ORDER BY income DESC LIMIT 1`. The difference may seem small, but consider instead accessing the row of rank, say, 100. The following query is in order: `SELECT * FROM df AS df1 WHERE (SELECT COUNT(*) FROM df AS df2 WHERE df1.income > df2.income) = 99`. There are more steps here than the list-based approach. Now the reader should try writing the query for the median. Our observation is further backed up by the fact that among the pandas APIs called in the top voted Kaggle kernels (as an approximation of pandas API use), `iloc` is more commonly used than `join` and `drop` [8].

Second, looping over a list is sometimes easier to use than mapping over a set. For instance, consider getting the smallest *missing* value, e.g., `{1,2,4,5,10}` misses 3. With loop-based thinking, a programmer can just pause after the first "gap", but in set-oriented thinking, one solution is to create a number of continuous integers, and then select the minimum value that's in the integers table, but not in the set at hand, e.g., `SELECT MIN(n) FROM numbers WHERE value NOT IN (SELECT value FROM t)`. Another example is computing the difference between conceptually consecutive items, like what month-to-month sales change. A SQL query would use a `JOIN` operator, `SELECT t1.month, t1.sales-t2.sales FROM t AS t1 JOIN t as t2 on t2.month = t1.month - 1`.

One may be tempted to draw the conclusion that lists are superior to sets from a programming experience perspective. However, it is unclear if that is due to a bias in education. Besides, the argument for programming without relying on order is increasingly

[3] And arrays, which is also ordered, the key feature discussed.

important for programming with large datasets on the cloud [12], where the cost of maintaining order is incredibly high, both in terms of performance, and engineering and compute resources. Sets are much superior to lists when it comes to performance since operations over sets can be easily parallelized. Since order is so expensive, it makes sense to have programmers ask explicitly for order only when it is needed, as opposed to being assumed to be always present. In addition, operators over sets tend to encourage more declarative thinking, since there is no order to base the procedural thinking on. The key question for the future from this section is understanding how difficult will it be to teach developers to program without a constant assumption about order.

2.2 Matrix vs. Table

In a table, rows and columns are fundamentally different abstractions – one could reason about rows but not columns, which would be second-order logic. Matrices do not differentiate between rows and columns and are traditionally used in linear algebra. Many, like dot-products, are awkward in SQL – the values would be stored in the schema `(rowID, colID, value)`. By contrast, matrices don't provide the logical operations natural to relations, like selection or join. However, dataframes provide a mix of matrix operations and relational operations, because they provide both logical operations and linear algebra like operations.

2.2.1 Mixing Matrix and Tables, the Good Part

This mixture can be handy. Consider Table 2. If a user wishes to compute the total sales for each year, it makes sense to apply an aggregation across the columns – `sales_df.apply(lambda row: sum(row) - sum.year, axis=1)`, rather than writing out all the column names manually – `SELECT fruits + nuts + dairy + meat ... FROM sales`. The limited operations allowed over tables could be casts as *Premature Commitment*, per Green's Cognitive Dimensions of Notations [18]. Perhaps the *messy conflation of relations and matrices* [20] is what makes scripting and "exploratory programming" easy [21].

Table 2 An example table where programmatically iterating through the columns is desirable.

year	fruits	nuts	dairy	meat	...
2017	100	40	300	400	...
2016	200	150	200	400	...
...	...	...	...	...	...

However, there is a better way to achieve the above-mentioned functionality in databases by changing the schema. Instead of having the year *values* as columns, the table can have the values in the rows, with a new schema: `year`, `category`, and `amount` (Table 3). Then the query is `SELECT year, category, SUM(amount) FROM sales GROUP BY year`.

Table 3 An example table that contains the same data as the previous table, but different schema.

year	category	amount
2017	fruits	100
2017	nuts	40
...	...	...

This style of schema design is not accidentally convenient. *tidyverse*, a popular ecosystem of libraries in R, encourages programmers to organize experiment variables as columns, and rows as observations (which makes the data relational). The data layout design increases the ease of manipulation and usage of libraries like `dylplr` and `ggplot2`, as well as better performance [29].

2.2.2 Mixing, the Bad Parts

A quirky outcome of the matrix thinking gone too far is that dataframes allow **duplicate column names**. For example,`pd.DataFrame([[1,2,3],[2,3,4]], columns=['a','a'])` evaluates without error and results in a dataframe with two column `a`'s.

Another issue is the additional notion of **indices** in dataframes. Since a pandas dataframe is a "dict-like container", it has an explicit notion of the index. Programmers can change and use indices – they can even use a "hierarchical" index called `MultiIndex` [6]. For instance, a dataframe of sales information (`store_id`, `date`, `item` etc.) can be indexed with the store type (e.g., Trader Joes) and state (e.g., California). Each state can have multiple store types. As a result, queries like finding all sales at Trader Joes in California: `df.loc[("California", "TraderJoes")]`.

Tables do not have an explicit notion of indices, but rather the concept of keys, which are made up of sets of columns. Following the example, the index can be replaced by two additional columns, `store` and `state`, then the same query can be written with `WHERE state ="California" AND store="TraderJoes"`. Tables do not need indices to achieve the same functionality.

In fact, indices make the downstream API more complex. Let's take a look at `concat` again. While `concat` was compared to `UNION`, the semantics of `concat` are much more complex. In SQL, `UNION` can only be applied to rows of the same width (and type), but in pandas, a programmer need to specify whether the concatenation is along rows or columns and how to match the indices or columns (full `concat` API: `pd.concat(objs,axis,join,ignore_index, keys, levels, names, copy, verify_integrity)`). An index is a redundant construct.

2.2.3 Joins

The join operator is the crown jewel of the relational operators. It accepts a join condition (selection) and a pair of tables as arguments and returns a table [25]. Joins enable programmers to derive all kinds of information from relationships between data.

Joins in dataframes (called `merge`), on the other hand, look very different as a result of the mixing of metaphors. Here's what it looks like: `pd.merge(left_df, right_df, how, on, left_on, right_on, left_index, right_index, sort, suffixes, copy, indicator, validate)`. The merge operator seem rather complex, with concepts like `left_index` and `suffixes` that are not needed for joins on relational tables. Furthermore, despite the complexity, the join condition is limited to column equalities (a concept called *equijoin* in databases). Non-equijoins are very common. For example, to find the weather of events by joining the events table and the weather table. In SQL this can be written as `SELECT * FROM events JOIN weather_log w WHERE events.date > w.start AND events.date < w.end`, and this cannot be expressed using the *merge* function alone in dataframe APIs in pandas or R. The lack of non-equijoin support is not an oversight. One key pandas maintainers, Jeff Reback, explained in a discussion on GitHub that adding non-equijoins is not Pythonic [1]. This is puzzling because, for joins between tables, both equijoins and non-equijoins are predicates over columns and are treated exactly the same algebraically [25].

In this sense, the `merge` is not actually a database join, as was claimed by both R and pandas' documentations [4, 5]. What further complicates the picture is that the `concat` operation (discussed briefly earlier) also have mixed in concepts of join. In fact, in dataframes, `concat` is much closer to `merge` than `UNION` is to `JOIN`; `concat` even takes a inner or outer join specification [4]! This odd overlap of concepts could cause additional confusion. For instance, the `concat` function requires that the names or indices of the columns match, but with `merge`, programmers could specify what columns to merge on, and the names do not always have to be the same. This makes rows and columns not actually symmetric, as the name "matrix" may initially suggest, further complicating the mental model a programmer must hold of dataframes.

Perhaps a better design is to have tables and matrices as two separate data structures each with their own operations.

3 Programming Workflows

Writing SQL and dataframe queries are very different experiences. The first is a declarative language, and the latter is invoking library functions in a host language. SQL was initially intended to be used by non-programmers like accountants and architects through a terminal interface [14, 15]. On the other hand, dataframe libraries are serving data scientists who can program. While SQL encourages programmers to think only about tables and reason about high-level functionalities, which are easier to optimize. Dataframes are used more procedurally, where programmers apply a sequence of operations on the dataset [13]. In this section, we analyze the different implications of the procedural versus declarative approach.

3.1 Accessing Functions

SQL was designed to be used just by itself, typed into a terminal. However, increasingly programmers need to access the query results in a general-purpose programming environment, which they can achieve using a wrapper, often constructing the query as raw strings with no IDE support – no syntax highlighting, no linting, no refactoring, and no type checking. It also makes accessing functions more difficult. Functions are useful for defining custom predicates and aggregations. In SQL, programmers can access custom user-defined functions (UDFs) by registering the function to the database via a wrapper, e.g., `sqlite3.create_function (<custom_agg>)`.

3.2 Code Reuse and Composition

Functions are also useful for sharing code that expresses the same logic, which is good for code reuse. Codesharing is easy with dataframes since they are manipulated with library calls in a general-purpose host language. In a similar vein, dataframe queries can also be composed using functions directly. For instance, the logic to pick different tables or columns can be embedded in normal functions. Composition reduces the complexity of analytic tasks with more succinct and readable code.

On the other hand, in SQL, the primary way to reuse code is via `VIEW`s, which are similar to tables, except that they are not computed and persisted to storage (*materialized*). Since SQL can only express first-order logic, the ability to reuse code using `VIEW`s is limited. For example, programmatically changing or iterating over different tables or columns is second-order logic and cannot be expressed in SQL. Programmers would have to generate code strings, which is brittle or manipulate abstract syntax trees, which is often an over-kill.

Consider a scenario where the programmer first creates an aggregation, then wishes to add a filter before the aggregation. Take a table of flight delay data (`delay`, `origin`, `time`, etc.). To aggregate based on `origin` in SQL, the programmer can write, `CREATE VIEW originAgg SELECT origin, COUNT(*) FROM flights GROUP BY origin`. After seeing the result, they plan to see the same aggregation but filter by flights thats have delays greater than an hour. However, there is no choice but to copy the query and add a where clause – `SELECT origin, COUNT(*) FROM flights WHERE delay > 60 GROUP BY origin`. There is no way to reuse the view `originAgg` because the `delay` column is no longer accessible. Although the copy-paste then edit seems fine, the problem becomes more pronounced with longer queries. With dataframes, the programmer can extract the operations `originAgg=lambda(df):df.groupby(['origin']).count()`, and pass the filtered dataframe to the `originAgg` function.

3.3 Debugging

The function chaining style of dataframes forces the programmer to flatten out the operations, and the results from a function can be inspected. However, this is not the case for SQL. For instance, a programmer may write a query using both `HAVING` (with `GROUP BY`) and `WHERE` clauses. When the result is unexpected, and the programmer wishes to inspect the intermediate result, they would have to write new queries that capture the logic individually. With dataframes, the programmer may already have the intermediate variables, and if not, they can just break the chain and inspect the intermediate values without writing new code.

3.4 Performance

There are many different sequences of operators – a *physical plan* in databases – that evaluate to the same values. Some of the sequences are slower and take more resources than others. With the exception of Spark, dataframe programmers currently have to figure out what is better themselves. To demonstrate, we borrow an example given by the pandas creator Wes McKinney: to sum the values of column `c2` from rows whose `c1` column is negative [23]. With pandas, there are at least two ways:

1. Find the rows, then sum – `df[df.c1 < 0].c2.sum()`. When these functions are invoked, pandas creates a temporary dataframe `df[df.c1 < 0]`, then sums `c2` column of that temporary object. The temporary dataframe can be wasteful if `df` contains a lot of columns.
2. Trim the dataframe down to just the `c2` column, using the index from applying the predicate on `c1`, then sum – `df.c2 [df.c1 < 0].sum()`. Since the df is first projected, it uses less memory.

In SQL, the solution is `SELECT SUM(c2) FROM df WHERE c1 < 0`, and the database will decide what sequence of operators to execute by using a *query optimizer*, which finds a better execution plan regardless of the specification.

This issue is not just limited to the order of operators; the choice of functions can also lead to performance issues. For example, the `apply` function in pandas prevents vectorized processing, and there are often alternatives, such as using a UDF to process the whole column instead of a row at a time, or sometimes to use the `iterrows` function to loop, leading to extensive discussions among users of dataframes [9].

Is the situation unredeemable? Not quite, when evaluated lazily, dataframe operators can still benefit from a subset of query optimization techniques available to SQL [13]. Furthermore, performance is not the only issue programmers care about. Sometimes it is confusing when a

query is slow, and we know that query optimizers don't always work very well [22]. It might be a better programming experience to be able to build a mental model and have a more predictable performance that the user can improve on. Perhaps a good programming system is not one that executes faster on average but one that respects the programmer's agency.

3.5 Code Comprehension and API Recall

Some programmers dislike the syntax of SQL, where the `ALL CAPS` syntax can seem ugly. This is partially due to historical reasons since syntax highlighting wasn't available when SQL was first created (although even today SQL is often run in strings and terminals, keeping the need for the caps).

Caps aside, we would argue, however, that SQL is actually more *role-expressive* than dataframe APIs. Role-expressiveness captures how easy it is for a programmer to parse code into mental structures [18]. Because SQL blocks are fairly constrained, reading the operations is straightforward, at least for simple queries, but this is not the case for dataframes. Consider the many ways of expressing `SELECT * FROM df WHERE a > 3` in pandas, a non-exhaustive list below, with different evaluation below. The syntax differences are partially caused by the use of syntax shortcuts, the use of defaults in function calls, and pandas exposing lower level executions.

- `df[df.a>3]`
- `df[df["a"]>3]`
- `df.loc[df.a>3]`
- `df.loc[df["a"]>3]`
- `df[lambda foo:foo.a > 3]`
- `df.loc[lambda df: df["a"] > 3]`
- `df.loc[df.apply(lambda r: r["a"] > 3, axis=1)]`
- `pd.DataFrame([r for r in df.itertuples() if r.a > 0])`

Having many different ways to express the same logic makes it hard for developers to understand programs of heterogeneous styles. Besides having varying ways to express the same simple logic, the sheer number of APIs (> 200) that are not only overloaded but also have default parameters that may change version to version, making it hard to remember the APIs. Developers may end up looking up documentation or search StackOverflow and break the flow. The question for the future is whether a library should optimize for short solutions to a large number of problems or optimize for a shorter list of APIs.

4 Conclusion and Future Work

So is a dataframe just a table? Our answer is that in the wild, dataframes are used differently from tables and carry many additional useful functionalities at the cost of clarity and performance. More PL-HCI work is needed to create language affordances that could help combine the best of both worlds for programmers and language creators. In particular, we had not time in this paper to investigate the design choices made by other languages that integrate relational operators into a host language, such as LINQ [24], FlumeJava [16], Eve [2], and Object Relation Managers. The creators of these languages have also made various discoveries and claims about the desired properties of the language that might yield insight into what the "Pareto-efficient" trade-off curve is between factors.

In the process of investigation, we have opened up new questions. Some are human factor questions, like whether it is inherent that programmers prefer lists over sets. Others are philosophical questions, like whether language designers should meet users where they are. We hope these questions can help inspire more discussions and analysis of a more HCI-oriented study of query languages, to help guide language and library creators with more material to discuss, less confusion, and better principles.

4.1 A Cliffhanger

Before we end, we must share another teaser controversy. A well-followed database researcher, Kelly Sommers, tweeted, *Why GraphQL when we could have used SQL?* [28]. Sommers continues to argue that *GraphQL exists because JavaScript developers finally realized HTTP API's were too limiting so they reinvented SQL over JSON because JavaScript developers are obsessed with reinventing everything into JSON API's.* The tweet has received over 2K likes. The creators of GraphQL, experienced engineers at Facebook, responded that SQL did not serve them well and that GraphQL is the result of listening and empathizing with the needs of the product developers [26]. Now GraphQL has almost 15K GitHub stars and a large and active developer community. *What should we make of it?*

References

1 Conditional join (merge) in pandas. URL: `https://github.com/pandas-dev/pandas/issues/7480`.
2 Eve: Programming designed for humans. URL: `http://witheve.com/`.
3 HN thread about McKinny, Things I Hate About Pandas. URL: `https://news.ycombinator.com/item?id=15335462`.
4 Merge, join, and concatenate. URL: `https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#set-logic-on-the-other-axes`.
5 Merge two data frames by common columns or row names, or do other versions of database join operations. URL: `https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/merge`.
6 Multiindex / advanced indexing. URL: `https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html`.
7 pandas.dataframe docuemntation. URL: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`.
8 Usage of pandas api by kaggle usage. URL: `https://github.com/modin-project/study_kaggle_usage/blob/master/results.csv`.
9 When should i ever want to use pandas apply() in my code? URL: `https://stackoverflow.com/questions/54432583/when-should-i-ever-want-to-use-pandas-apply-in-my-code`.
10 Why do people prefer pandas to sql? URL: `https://datascience.stackexchange.com/questions/34357/why-do-people-prefer-pandas-to-sql`.
11 data.frame, r-core documentation, 2018. URL: `https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/data.frame`.
12 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.
13 Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.
14 Raymond F Boyce, Donald D Chamberlin, W Frank King III, and Michael M Hammer. Specifying queries as relational expressions: The square data sublanguage. *Communications of the ACM*, 18(11):621–628, 1975.

15 Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

16 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pages 363–375. ACM, 2010. doi:10.1145/1806596.1806638.

17 Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

18 Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

19 Joe Hellerstein. Stop. a "data frame" is just a table, August 2016. URL: https://twitter.com/joe_hellerstein/status/761364295510691840.

20 Joe Hellerstein. A "data frame" is a messy conflation of relations and matrices, March 2018. URL: https://twitter.com/joe_hellerstein/status/978335500250447878.

21 Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, pages 1265–1276, 2017.

22 Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

23 Wes McKinney. Apache arrow and the "10 things i hate about pandas", 2017. URL: https://wesmckinney.com/blog/apache-arrow-pandas-internals/.

24 Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

25 Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

26 Nick Shrock. Graphql exists not just because, November 2018. URL: https://twitter.com/schrockn/status/1060314584525955072.

27 Nathan Sidoli. Mathematical tables in ptolemy's almagest. *Historia Mathematica*, 41(1):13–37, 2014.

28 Kelly Sommers. Why graphql when we could have used sql?, November 2018. URL: https://twitter.com/kellabyte/status/1059956838744158213.

29 Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc., 2016.

Live Programming Environment for Deep Learning with Instant and Editable Neural Network Visualization

Chunqi Zhao
The University of Tokyo, Japan
shunnki.chou@is.s.u-tokyo.ac.jp

Tsukasa Fukusato
The University of Tokyo, Japan
tsukasafukusato@is.s.u-tokyo.ac.jp

Jun Kato
National Institute of Advanced Industrial Science and Technology, Tsukuba, Japan
jun.kato@aist.go.jp

Takeo Igarashi
The University of Tokyo, Japan
takeo@acm.org

Abstract

Artificial intelligence (AI) such as deep learning has achieved significant success in a variety of application domains. Several visualization techniques have been proposed for understanding the overall behavior of the neural network defined by deep learning code. However, they show visualization only after the code or network definition is written and it remains complicated and unfriendly for newbies to build deep neural network models on a code editor. In this paper, to help user better understand the behavior of networks, we augment a code editor with instant and editable visualization of network model, inspired by live programming which provides continuous feedback to the programmer.

2012 ACM Subject Classification Software and its engineering → Development frameworks and environments; Human-centered computing → Visualization toolkits

Keywords and phrases Neural network visualization, Live programming, Deep learning

Digital Object Identifier 10.4230/OASIcs.PLATEAU.2019.7

Funding This work was supported by JST CREST JPMJCR17A1.

1 Introduction

In recent years, deep neural network (DNN) model has garnered tremendous success in various application domains such as Image Processing (IP) and Natural Language Processing (NLP) research. From well-structured models, effective features can automatically be extracted without selecting manual-designed filters. Thus, deep learning has become a competitive solution for most traditional application domains, as well as some new areas where it shows the possibility such as computer graphics and robotics researches.

Deep learning programming, however, distinguish itself from conventional programming with some unique features. That is, deep learning algorithm is not to provide solutions to any specific applications, but to provide a way to extract features from data then optimize the parameters in a huge matrix with the features and the pre-defined constraints, and finally makes the optimized matrix a solution towards the application. Thus, rather than

determining solution details for specific problem by programmer in conventional programming, deep learning programming is to design a general way to search for solutions in various training data.

Many DNN framework libraries [1, 3, 6, 7] provide full-stack deep learning toolbox from preliminary layer definition, GPU-accelerated training optimizer to evaluation function. And nearly all the DNN frameworks follow the same coding style: (i) users use a separated file or block of code to define the network, then (ii) imports the network as an instance in the training control module. The problem is, the training process is resource-consuming. Before the actual training, the user needs to design the DNN architecture, load and pre-process dataset, give the hyper-parameters and determine how data goes through the system. If the shape of data in computing doesn't match the defined layer's parameters, the training process will fail to start. However, in current practice, if the user don't want to debug by running the training script, the only way to check the computed data is in a desired shape or not, is to validate data shapes on scratch paper line by line. Hence instant visualization during programming will be a more timely feedback for the programmer. Some DNN frameworks provide visualization of network structure as a node-link diagram. But they show such visualization only after the code or network definition is written, or even after the training phase is completed. Such visualization is not very helpful for newbies to build deep networks on a code editor.

Therefore, we present a novel web-based live programming environment specialized for neural network programming by integrating an instant and editable visualization into a standard code editor. The system is aware of which part of codes belongs to the layer definition and which part belongs to the data flow control. For the layer definition codes, when the user types in one line that contains the configuration of a neural network layer, the system understands the context and instantly create the layer's visualization at the visual panel; And for the data flow control part, we provide a bidirectional mapping between codes and visualization: we visualize the defined layers as candidates, and enable the user to build codes by dragging & dropping their visualization, or vice versa. This paper reports our design principles and some implementation details, as well as work in progress.

2 Related Work

Neural Network Visualization. Chainer [1] and Keras [3] has built-in visualization functions. However their display are too difficult to grasp the true important information of a neural net. Wongsuphasawat et al. [14] proposed a plugin in TensorBoard to visualize neural nets with data flow direction, but also particularly problematic that programmers may be confused in choosing the best view to show the important information. While several open source tools [2, 5, 4, 8] to visualize the networks are designed, they are focus on the stage after the user finish editing their source code, or even after the model's training. We therefore visualize networks at coding time, and propose the first tool to enable user programming deep learning code with living visual feedback.

Live Programming Environment. Live programming is a technique to provide the programmers with continuous feedback about the current program for understanding the behavior of the under-development program. One approach is to use explicit representations such as images, sounds or video [10, 11, 12] instead of textual information such as stack traces. The other approach is to directly display the value of program variables. For example, Python Tutor [9] divides the execution process into several steps of the program, and continues

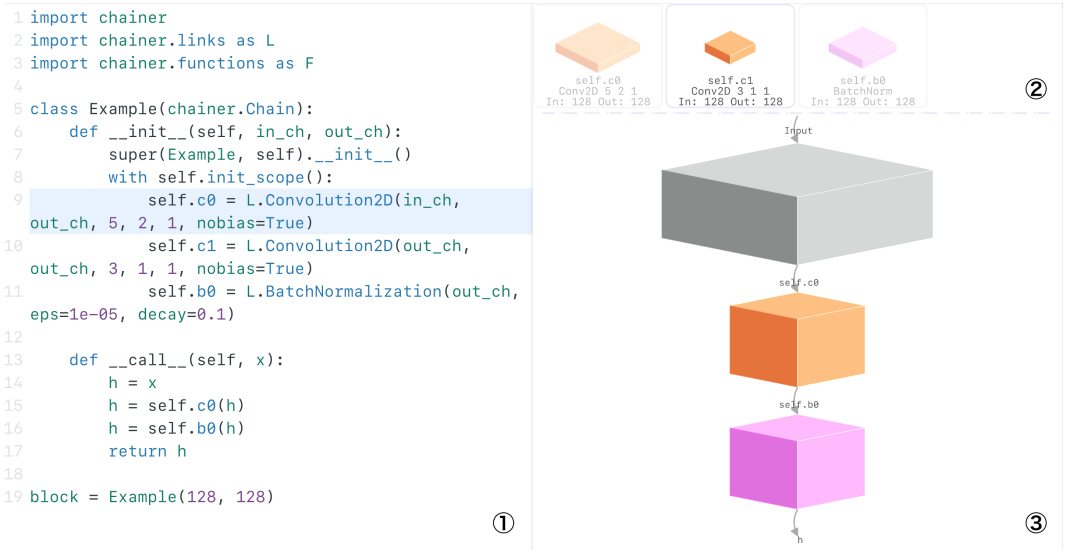


Figure 1 The screenshot of our system's first prototype. The interface is made up of: (1) code editor, (2) grid-like candidate region where layers defined in ___init___ are listed here, (3) data cubes are visualized in the form of a directional graph, whose direction is given in ___call___.

executing the under-development program. Kanon [13] enables to synchronously analyze data structure (e.g., linked list) without actually executing the program and update the data nodes graph. Our work builds on the live programming approach to develop deep learning programs while considering the characteristics of the current program.

3 Design Overview

3.1 Interface

Our interface (See Figure 1) consists of (1) the standard code editor to write DNN programs (the left-hand side) and (2) the visualization panel to display layer/dataflow (the right-hand side).

3.2 Design Principles

Based on the background of deep learning visualization tools, we design the system with the principles below.

First, and also the key idea to support our design is, **the visualization should be presented during coding**. Our motivation is to help user understand the network structure and simplify the debugging of deep learning programming. Naturally, the "live" programming manner that merges "coding" and "test" into a single phase makes the program developing a shorter loop.

Secondly, **besides the code editor, a graphical editor can be more helpful to a newbie**. Our target user is not a deep learning expert - that means, he/she might be a newbie in programming, or a programmer without deep learning programming experience. The utilization of graphical editor instruct the user, especially the newbie user, to build a neural network in a more intuitive way, just like drawing the network structure on scratch paper.

Finally, **to visualize the network in a most common, but interactive way**. Referring to the visualizations of deep neural networks from published papers, we think box-like visualization represents data's shape best. Layers and data are the two main factors in neural networks visualization, but if treated the same, the view will seem in a clutter. We believe that the suitable practice to visualize the network, is to have the two factors separated somehow.

3.3 Assumptions

To better locate our system's feature and the usage scenarios, we make assumptions as below:

- The DNN program is written in Chainer framework. In Chainer, the standard style to define a neural network is the code snippets in Figure 1. The neural network is defined using a Class that extends from chainer.Chain. The class, consists of two important functions: ___init___ gives all the layers' type and shape in the net, while ___call___ determines the order of layers.
- The user will define ___init___ first, then use the defined layers to further give the order of layers that the input data will go through in ___call___.
- The user will create a instance of the DNN he/she defined at the end of the program. This is for the convenience of syntax check.

4 Implementation

The whole system can be decomposed into frontend and backend. The frontend collects user's inputs in code and graphical editors, and the backend acts as a code parser.

4.1 Abstract Syntax Tree Parser in Backend

We set up a Python parser to process the code submitted from browser. The backend program checks the syntax of code, then parses it into abstract syntax tree. Since what we are interested in is the defined layers and data flow information, the program will distil the ast into a much smaller object that only contains the necessary information for user. Finally the object is returned to the browser for further processes.

4.2 Collaborated Code Editor and Visual Panel in Frontend

The proposed system can bidirectionally update the code editor and the visual panel, so this section describes the behaviors in each direction separately.

1) Code ⇒ Visualization. After the browser receives the extracted net information from the backend, it snapshots this frame and visualizes it. In the case where only ___init___ is given, layers will be visualized in (2) of our interface as cube models whose length and width represents the kernel size. Since connections are not defined, all cubes are individually packaged in a cell and listed in a grid-like structure. If ___call___ is also given by the users, layers picked in ___call___ will become transparent and its name will show on the lines between data cubes in (3) of the interface.

2) Visualization ⇒ Code. (In progress) We consider that to define a new layer, coding is a much more efficient means rather than graphical editing, thus we disable the synthesis of layer definition from visualization operation. The more encouraged way to edit codes from

the visual panel in our system is, with the defined layers cubes and the input data node, the user drag the layer, and drop at the data he/she want to process using this layer, in this way, the new line of codes in ___call___ as well as the last data in the dataflow will be synthesized one by one. For deletion, the mapping between code and visualization will still be bidirectional.

5 Limitation and Future Work

The current system is still a prototype, so we only support Chainer and the most common layers at this stage. Support for more frameworks and layers will make our system fit more situations. Besides, we consider adding an argument hinter to simplify layer argument configuration. There also remains space to make the code editor and the visual panel collaborate better, like more simultaneous visual feedback. In addition, evaluation such as user study will be performed to validate the effectiveness of the proposed system.

References

1 Chainer. `https://chainer.org/`. Accessed: 2019-06-05.
2 Hiddenlayer. `https://github.com/waleedka/hiddenlayer/`. Accessed: 2019-06-05.
3 Keras. `http://keras.io/`. Accessed: 2019-06-05.
4 Nn-svg: Lenet- and alexnet-style diagrams. `http://alexlenail.me/NN-SVG/LeNet.html`. Accessed: 2019-06-05.
5 Plotneuralnet. `https://github.com/HarisIqbal88/PlotNeuralNet`. Accessed: 2019-06-05.
6 Pytorch. `https://pytorch.org/`. Accessed: 2019-06-05.
7 Tensorflow. `https://www.tensorflow.org/`. Accessed: 2019-06-05.
8 Tensorspace.js. `https://tensorspace.org/`. Accessed: 2019-06-05.
9 Philip J Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceedings of the 2013 ACM SIGCSE*, pages 579–584. ACM, 2013.
10 Jun Kato. Visionsketch: Gesture-based language for end-user computer vision programming. In *Proceedings of the 2013 ACM SIGPLAN*, 2013.
11 Jun Kato, Sean McDirmid, and Xiang Cao. Dejavu: Integrated support for developing interactive camera-based programs. In *Proceedings of the 2012 ACM UIST*, pages 189–196. ACM, 2012.
12 Dan Maynes-Aminzade, Terry Winograd, and Takeo Igarashi. Eyepatch: prototyping camera-based interaction through examples. In *Proceedings of the 2007 ACM UIST*, pages 33–42. ACM, 2007.
13 Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. Live, synchronized, and mental map preserving visualization for data structure programming. In *Proceedings of the 2018 ACM SIGPLAN*, pages 72–87. ACM, 2018.
14 Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12, 2017.