Ravi Chugh | Research Statement

Reimagining the User Interfaces for Programming

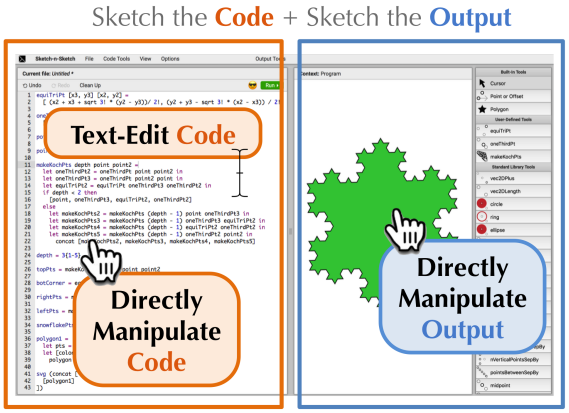
The programming process has remained stubbornly similar for decades: the user types source code into a text box, the system compiles and executes the code, and the user views the output and repeats. The output is not connected in meaningful ways to the code that generated it, making it difficult to understand and change the computation. This workflow limits the creativity and pace of expert programmers and shuts out less experienced users.

Can programming languages be equipped with interactive user interfaces?

Can the expressive power of programming be integrated into graphical user interfaces (GUIs) for creating documents, spreadsheets, charts, graphic designs, digital music, and more?

PL + UI Techniques. Motivated by these questions, my research pursues **programming language techniques** for synthesizing and transforming code, as well as **user interface techniques** to access the new language capabilities.

During the past several years, my students and I have developed Sketch-n-Sketch, a programming environment—thus far configured to produce HTML pages and Scalable Vector Graphics (SVG)—in which the user "sketches" a program using the code editor in the left pane and "sketches" desired program behavior through examples using the direct manipulation output pane on the right.



To enable this kind of program authoring workflow, we have explored four main directions of work discussed below. Many of these techniques have been investigated in separate versions of Sketch-n-Sketch, and some in different systems entirely. Nevertheless, we describe the ideas collectively, as part of a single system, because they share a unified vision and approach—to enrich general-purpose programming languages with interactive mechanisms for building programs, with domain-specific techniques and user interfaces where necessary for specific applications.

Programming Language Techniques

(1) Bidirectional Evaluation [PLDI 2016, OOPSLA 2018]. The traditional edit-compile-run loop of programming is particularly galling when successive program changes—and the resulting output changes—are narrow in scope.

In Sketch-n-Sketch, the user may "directly manipulate" the program output—textually or graphically, where possible—to make small changes; the system then uses a technique called *bidirectional evaluation* [1, 10] to run the previous program "in reverse" and synthesize necessary program repairs. In addition to a standard evaluation relation $e \Rightarrow v$ that evaluates expression e to value v, we define a *backward evaluation* relation $e \Leftarrow v' \rightsquigarrow e'$ that, given a modified value v', transforms the original expression e to e'. Backward evaluation proceeds by comparing the original output value v with the goal v', and synthesizing repairs to e such that, ideally, the new program e' evaluates to v'. To provide more complex or domain-specific repairs, expert users can customize the backward evaluator using the rich theory of *lenses* previously developed in the literature. Compared to prior work on bidirectional programming, our techniques allow arbitrary programs in a general-purpose functional language to be run in reverse, making it more likely for such techniques to be practical.

To further investigate bidirectional evaluation, we are extending our techniques to synthesize repairs corresponding to non-local edits (e.g., "cut-copy-paste"), to support additional language features (e.g., imperative assignment, objects, and exceptions), and to automatically derive backward evaluators from ordinary (forward) evaluators.

(2) Program Sketching with Examples [POPL 2019, ICFP 2020]. When programs do not parse or type-check—a routine, sometimes-protracted occurrence during development—programming environments provide little guidance as users work to fill in missing program fragments.

First, to provide continuous, "live" feedback about the behavior of a program under development, we propose a partial evaluator for *sketches*—incomplete programs, with *holes*—by continuing to evaluate parts of the program

that do not depend on the missing pieces [13]. The results of partial evaluation can provide useful feedback as users decide how to continue writing the program.

Second, Sketch-n-Sketch analyzes the results of partial evaluation—including any `assert` statements in the sketch—to derive input-output example constraints that are used to synthesize expressions to fill the holes [9]. Our techniques address several usability limitations of prior programming-by-example techniques: users need not provide trace-complete (i.e., inductive) examples in order to synthesize recursive functions; users can simultaneously constrain multiple, interdependent synthesis tasks via `assert` statements; and users can provide partial implementations as part of the specification (static, solver-based techniques support sketching, but prior example-based techniques have not). Based on our improvements, Sketch-n-Sketch synthesizes several data-structure manipulating benchmark tasks with smaller specifications (the combined size of examples and sketch, if any) compared to previous state-of-the-art example-based synthesizers.

Future Directions. To further improve the usability of our program repair and program synthesis techniques, we plan to investigate ways to communicate the search results—including partial solutions—to the user, as well additional ranking metrics—beyond the usual smaller-program heuristic—to help synthesize code fragments that "fit" within the surrounding code and within the previous history of program edits. We will also investigate how additional forms of specification—such as interaction with evaluation traces (in addition to just examples of the eventual values they should produce)—may further help the system suggest desirable program changes.

Most systems that provide program synthesis, including Sketch-n-Sketch, require the user to pick a single solution. The desired behaviors for code under development, however, are subject to change and may not crystallize until subsequent phases of development. It would thus be preferable to delay the choice, allowing program "variations" to persist during the authoring session and to inform subsequent edits by the user and the synthesis engine. Doing so effectively will require techniques to evaluate and analyze program variants efficiently, and to summarize and visualize their results in intuitive ways.

User Interface Techniques

(3) Output-Directed Programming [UIST 2016, UIST 2019]. For many tasks, programmers face a choice: Use a GUI and sacrifice flexibility, or write code and sacrifice ergonomics? To obtain both flexibility and ease of use, we are exploring a workflow called *output-directed programming* [5, 7] in which traditional GUI interactions—such as those found in graphics editors and other direct manipulation interfaces—are interpreted as transformations of ordinary code in a general-purpose programming language.

To create vector graphics in Sketch-n-Sketch, the user draws new elements directly in the canvas, and the system generates high-level, readable code that, when executed, produces those elements. Through GUI actions, the user declares new relationships among output values—e.g., to equate colors, to relate positions, or to group elements—and Sketch-n-Sketch transforms the program to satisfy the declared relationships. These designs can be abstracted into reusable functions, which extend the existing set of drawing tools (i.e., library functions). Unlike many prior programming-by-demonstration techniques, we embrace a full-featured, general-purpose functional language—with an emphasis to build composable functions—because of the potential to provide a spectrum of expressiveness between a "low floor" for novices and a "high ceiling" for experts. To demonstrate the expressiveness of output-directed programming for SVG, we have implemented a variety of parametric designs in Sketch-n-Sketch—the generated code is readable, reusable, and can be inspected and modified with text-edits at any time.

Looking beyond domains with inherently visual representations, we will explore similar workflows for direct manipulation domains which also involve textual and numerical values—such as documents and spreadsheets.

(4) Hybrid Text-and-GUI Code Editors [ICSE 2018, VL/HCC 2020]. For many code transformations, there is a tradeoff between text-editing—which provides flexibility and concision—and structure editing or automated refactoring—which avoid syntax and certain semantic errors.

To combine these benefits, Sketch-n-Sketch augments textual code with graphical widgets that allow the user to structurally select subexpressions and other relevant features in the program; Sketch-n-Sketch then displays a menu of potentially relevant program transformations based on the current selections [8]. In addition to direct manipulation of textual code, we have also developed a technique for structurally manipulating textual representations of output values: by tracing the execution of a `toString` function used to "visualize" a value, our technique automatically derives a *tiny structure editor* on the output string—UI widgets for selecting, adding, removing, and modifying elements of the original value are displayed atop appropriate substrings [6].

Future Directions. The aforementioned techniques offer new GUI interactions for manipulating code and output. For larger programs, evaluation involves many complex intermediate data structures that are not directly represented in the eventual output. A challenge for future work is to devise visualization and interaction mechanisms for all steps of an evaluation trace, in between the static source code and its final output.

The user interface features above have been co-designed with domain- and implementation-specific program transformations. To make hybrid editing environments more extensible, we will investigate domain-specific languages that allow custom program transformations to be defined in easier and more composable ways.

We will furthermore investigate ways to support the smooth transition between multiple views (or "projections") of code and its evaluation. For example, we are defining a mechanism called *live literals* that allows users (or tool builders) to define custom, type-specific GUIs for generating code [12]. We also imagine a notion of "code style sheets" that allows different programming choices—from relatively simple choices such as code formatting, to more complex choices between syntactically-distinct but semantically-equivalent expressions—to be applied according to different user preferences, code comprehension goals, screen sizes, etc.

Each of these pursuits aims to make code editors more interactive, with the raw, unrestricted power of text-editing serving as just one tool among many for writing and reading code.

A "Computational Canvas" for Every Application Domain

The tension between programming languages and direct manipulation GUIs crops up in every direction—web development (for example, JavaScript vs. Dreamweaver), data analysis (R vs. Excel), data visualization (D3 vs. Excel), word processing (LaTeX vs. Word), presentations (Slideshow vs. PowerPoint), 2D graphics (Processing vs. Illustrator or Photoshop), 3D graphics (OpenGL vs. SketchUp), and countless more. In each setting, the story is the same—programming and GUI interfaces provide distinct benefits for various users and usage scenarios; without the ability to move between modalities, users are stuck with each interface's shortcomings.

My central assumption is that each of these software application domains can and should be equipped with a **"computational canvas"—a live, bidirectional, and direct manipulation interface for programming**—that eliminates the dichotomy above. The work described above—contributing to program synthesis, programming by demonstration, bidirectional programming, live programming, structure editing, and automated refactoring—constitute small steps towards this long-term vision. In our continued efforts, we will instantiate and develop any novel techniques for a variety of application domains, discussed below.

Everyday Work. Office applications are ubiquitous: word processors, spreadsheets, presentation editors, calendars, email clients, and so on are used by vast numbers of users for increasingly many professional and personal tasks. Spreadsheets, in particular, are often referred to as the most widely-used programming language: a broad spectrum of spreadsheet users are able to use formulas to programmatically manipulate their datasets. Following the lead of spreadsheets, I plan to investigate how formulas—a lightweight but powerful way to integrate programming within a GUI editor—can be integrated into other office applications. In addition to the "clean-slate" design and implementation approach taken in Sketch-n-Sketch so far, I will consider how to integrate our techniques into existing popular office applications—such as Google Docs or Microsoft Office—through plug-ins that allow (bidirectional) formulas to be mixed into the existing GUIs.

Creative Work. Design tools—such as Adobe's Illustrator, Photoshop, and InDesign—include scores of powerful features that support the construction of complex and nuanced artifacts. However, these tools generally lack proper tools for helping designers to abstract and reuse their work. Furthermore, designers and programmers regularly collaborate, often iterating between prototyping and implementation tasks—the literature documents *designer-developer breakdowns* that are due in no small part to the lack of shared and integrated tooling between these two usually-distinct groups of users. Sketch-n-Sketch only begins to scratch the surface of how "design" and "development" work may be streamlined; this will continue to be a goal for my research.

Data Visualization. There are several common user interfaces for creating visualizations: chart choosers such as Microsoft Excel and Google Sheets, shelf builders such as Tableau, and textual specification or programming languages such as Vega and D3. We are developing an integrated visualization editor called Ivy [11] that aims to streamline these interface modalities. Our approach is to endow declarative visualization grammars (e.g., Vega and Vega-Lite) with language abstraction mechanisms that, first, allow visualizations to be packaged into reusable units and, second, allow these visualizations to be instantiated and explored using type- and semantics-aware reasoning techniques. To demonstrate how Ivy smoothly combines several existing visualization interface modalities, we used

Ivy to implement reusable Vega and Vega-Lite visualizations that emulate a variety of chart types found in existing chart choosers and shelf builders.

Besides data exploration and presentation, data transformation is another fundamental aspect of a data analytics pipeline. In practice, users interleave and iterate transformation, exploration, and presentation tasks, yet existing systems are generally tailored for individual phases. We will seek to integrate these phases and the different user interface modalities for each—building on the computational spreadsheet canvases described above, as well as the exploration and presentation methodologies in Ivy—to realize a computational canvas that combines, and extends, the benefits of traditionally distinct systems like spreadsheets and computational notebooks.

Web Development. Web application development involves many of the characteristics described above: designers and developers collaborate, and the resulting artifacts involve visual and textual elements as well as databases. Beyond these concerns, computational canvases for web applications will also need to support the creation of dynamic behaviors.

Regarding the source programming language itself, our work thus far supports a core functional language, with user-facing syntax that resembles Elm or Haskell. As we extend our program synthesis and program transformation techniques to support widely-used features—such as mutable assignments and objects, as studied in my earlier work on types and static analysis for dynamic languages [2, 3, 4]—we may choose TypeScript as one particular user-facing language. TypeScript and Elm are related to and interoperate with JavaScript, the de facto programming language for the web and thus many of the application domains described above.

Research for Practice. In building these prototype systems, I hope to lay programming language and user interface foundations that enable professional software engineers to develop industrial-strength interactive programming systems. Therefore, in addition to disseminating ideas in academic settings, we will continue to develop and release open-source software, present in venues—such as Strange Loop, Elm Conference, and the Future of Coding podcast—that attract a mix of software engineers and researchers, and distribute tutorials and videos that are less technical than our research papers and seminars. I will also incorporate these systems into a variety of classes that I develop and teach.

Ultimately, the goals of my research—to bridge the gap between programming and GUIs—are to allow experienced programmers to funnel more creativity into tasks that truly require human insight, and to provide novice users a tenable path for learning to harness computational power—boosting productivity in software technology as well as other fields.

Acknowledgements

Many thanks to Brian Hempel, Justin Lubin, Mikaël Mayer, Cyrus Omar, Nick Collins, and Andrew McNutt (in chronological order of collaboration), without whom this work would not have been possible. Thanks also to the U.S. National Science Foundation, the Swiss National Science Foundation, and the University of Chicago for generously supporting this research. Several short passages above are adapted from the references.

References

[1] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. **Programmatic and Direct Manipulation, Together at Last**. In *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[2] Ravi Chugh, David Herman, and Ranjit Jhala. **Dependent Types for JavaScript**. In *Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2012.

[3] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. **Staged Information Flow for JavaScript**. In *Programming Language Design and Implementation (PLDI)*, 2009.

[4] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. **Nested Refinements: A Logic for Duck Typing**. In *Principles of Programming Languages (POPL)*, 2012.

[5] Brian Hempel and Ravi Chugh. **Semi-Automated SVG Programming via Direct Manipulation**. In *Symposium on User Interface Software and Technology (UIST)*, 2016.

[6] Brian Hempel and Ravi Chugh. **Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)**. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020.

[7] Brian Hempel, Justin Lubin, and Ravi Chugh. **Output-Directed Programming for SVG**. In *Symposium on User Interface Software and Technology (UIST)*, 2019.

[8] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. **Deuce: A Lightweight User Interface for Structured Editing**. In *International Conference on Software Engineering (ICSE)*, 2018.

[9] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. **Program Sketching with Live Bidirectional Evaluation**. *Proceedings of the ACM on Programming Languages (PACMPL), Issue ICFP*, 2020.

[10] Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. **Bidirectional Evaluation with Direct Manipulation**. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

[11] Andrew McNutt, Gordon Kindlmann, and Ravi Chugh. **Ivy: An Integrated Visualization Editor via Parameterized Declarative Templates**, April 2020. In submission.

[12] Cyrus Omar, Nick Collins, David Moon, Ian Voysey, and Ravi Chugh. **Filling Typed Holes with Live GUIs**, June 2020. In preparation.

[13] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. **Live Functional Programming with Typed Holes**. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL*, 2019.