Bidirectional Programming with Direct Manipulation Ravi Chugh — Research Statement

Can programming languages be equipped with interactive GUIs for building programs?

Programming language user interfaces have evolved remarkably little over the decades: the user types source code into a text box, the system compiles and executes the code, and the resulting output is inert—no longer connected to the code that generated it. This workflow stands in stark contrast to the direct manipulation graphical user interfaces (GUIs) used by millions of people to make documents, spreadsheets, presentations, graphic designs, digital music, and more. My research reimagines the user interfaces for programming by pursuing novel programming language techniques for reasoning about and transforming code, as well as novel user interface techniques to access the new language capabilities.

Program Repair via Bidirectional Evaluation [PLDI 2016, OOPSLA 2018]. To change the output of a program in a traditional language, the user must edit the source code, compile and run it again, and view the new output, often repeating this loop ad nauseam. We have developed an experimental programming environment—called SKETCH-N-SKETCH—in which the user directly manipulates the program output (rendered graphically for HTML and SVG output, or textually otherwise); our *bidirectional evaluator* then runs the program "in reverse," synthesizing small program repairs so that the new program evaluates to the desired output. Compared to prior work on bidirectional programming, our techniques allow arbitrary programs in a general-purpose language to be run in reverse, making it more likely for such techniques to be practical. To further realize this goal, we are extending our techniques to support language features found in widely-used languages (e.g., imperative assignment, objects, and exceptions), and to *automatically* derive backward evaluators from ordinary (forward) evaluators.

Program Synthesis via Live Bidirectional Evaluation [POPL 2019, Draft 2019]. When programs do not parse or type-check—a routine, sometimes-protracted occurrence during development—programming environments provide little guidance to users as they work to fill in missing program fragments in order to achieve their desired outcomes. To provide continuous, "live" feedback about program behavior, we proposed a semantics for evaluating *sketches*—partial programs, with *holes*—by continuing to evaluate parts of the program that do not depend on the missing pieces. Given the partial output of a sketch, SKETCH-N-SKETCH allows the user to provide examples of how the eventual result ought to behave, and our *live bidirectional evaluator* synthesizes expressions to fill the holes. Compared to prior work on programming-by-example, our synthesis techniques address several usability limitations: users can convey more information to the system via sketches, evaluation of the program itself can be used to derive examples, and fewer examples are needed for certain classes of synthesis tasks. To further improve the usability of our synthesis techniques, we will investigate user interfaces for communicating the search results 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.

Program Transformations via Direct Manipulation of SVG Output [UIST 2016, UIST 2019]. To make a visual design in SKETCH-N-SKETCH, the user draws new elements directly in the canvas (as in a traditional GUI editor), and the system synthesizes 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. Unlike many end user programming techniques, we embrace a full-featured, general-purpose language because of the potential to provide a spectrum of expressiveness between a "low floor" for novices and a "high ceiling" for experts. To better handle the inherent ambiguity of interpreting GUI actions as code edits, we will investigate ways to maintain sets of candidate program edits throughout an authoring session. We also aim to extend output-directed program synthesis to domains where objects do not already have a direct visual representation.

Program Transformations via Direct Manipulation of Code [ICSE 2018]. For many code transformations, there is a tradeoff between text-editing—which provides flexibility and concision—and structured editing or automated refactoring—which avoid syntax and certain semantic errors. To combine these benefits, the SKETCH-N-SKETCH code editor augments text with graphical widgets directly atop the program text that allow the user to *structurally select* subexpressions and other relevant features of the program structure, and a context-sensitive menu of program transformations based on the current selections. To make this hybrid approach more extensible, we are designing domain-specific languages for defining custom user interface elements and custom program transformations.

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