

THE UNIVERSITY OF CHICAGO

TANGIBLE VALUES WITH TEXT:
EXPLORATIONS OF BIMODAL PROGRAMMING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
BRIAN HEMPEL

CHICAGO, ILLINOIS

MARCH 2022

Author's version: I've reformatted this copy to improve readability. Page and figure numbers do not match the version of record at <https://knowledge.uchicago.edu> and ProQuest.

Copyright © 2022 by Brian Hempel

This work is licensed under the Creative Commons Attribution 4.0 International License.

<https://creativecommons.org/licenses/by/4.0/>

Figure 4.2 is reproduced with permission, from: Conal M. Elliott. 2007. Tangible Functional Programming. In Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).

Association for Computing Machinery, New York, NY, USA, 59-70. Fig. 14.

DOI: <https://doi.org/10.1145/1291151.1291163>.

In reference to IEEE copyrighted material which is used with permission in this thesis (Chapter 3; excepting 3.4, portions of 3.1, and portions of 3.6), the IEEE does not endorse any of University of Chicago's products or services.

Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

There's a link in the above blurb, but this one will give you more delight:

<https://www.bishopslaw.co.uk/about-us/lawyer-jokes/>.

For my dear family and friends and Zion.

Serve the Lord with fear, and rejoice with trembling.

Psalm 2:11

Table of Contents

Acknowledgments	vii
Abstract	viii
1 Introduction	1
1.1 Bimodal Programming	1
1.2 Related Work	2
1.2.1 Live Programming	2
1.2.2 Direct Manipulation Programming	3
1.2.3 Prior Bimodal Systems	5
1.3 Avoiding the Pitfalls of PBD	6
2 Sketch-n-Sketch: Bimodal Programming for SVG	8
2.1 Introduction	8
2.2 Related Graphical Bimodal Environments	9
2.3 Overview	10
2.3.1 Draw	11
2.3.2 Relate	12
2.3.3 Group	13
2.3.4 Abstract	14
2.3.5 Refactor	14
2.3.6 Koch Snowflake Fractal (feat. recursive drawing)	16
2.3.7 Tree Branch (feat. repetition over list and offsets)	24
2.3.8 Target (feat. repetition by demonstration)	25
2.4 Design and Implementation	27
2.4.1 Solver-based Value Updates	28
2.4.2 Provenance	31
2.4.3 Shape Selection & Feature Widgets	36
2.4.4 Intermediate Value Widgets	38
2.4.5 Value Holes & Location Holes	42
2.4.6 Naming	47
2.4.7 Brands	48
2.4.8 Drawing	49
2.4.9 Focusing	50
2.4.10 Group & Abstract & Merge	51
2.4.11 Repetition	52
2.4.12 Refactoring Tools	54
2.5 Evaluation (Case Studies)	55
2.5.1 Authoring	56
2.5.2 Remaining WWID Tasks	57
2.6 Discussion	58
2.6.1 Can you hide the code?	58
2.6.2 Limitations & Future Work	58
2.7 Summary	60

3	Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)	61
3.1	Introduction	61
3.2	Algebraic Data Types (ADTs)	63
3.3	Algorithm	63
3.3.1	Dependency Tracing	64
3.3.2	Spatial Regions	64
3.3.3	Selections and Actions	65
3.4	Implementation Details	66
3.4.1	A Dependency Provenance Algorithm	67
3.4.2	Code Normalization	70
3.4.3	Concatenation Tree Normalization	71
3.4.4	Positioning Remove and Insert Buttons	71
3.5	Case Studies	72
3.6	Limitations and Future Work	73
3.7	Summary	75
4	Maniposynth: Bimodal Tangible Functional Programming	76
4.1	Introduction	76
4.2	Overview Example	79
4.2.1	List length, without synthesis	79
4.2.2	Undo and delete	84
4.2.3	Value-centric shortcuts, and synthesis	85
4.3	Implementation	87
4.3.1	Architecture Overview	88
4.3.2	Interpreter	89
4.3.3	Fluid Binding Order	92
4.3.4	Synthesizer	93
4.4	Evaluation	98
4.4.1	Study Setups	99
4.4.2	Results	101
4.5	Related Work	106
4.6	Future Work and Conclusion	109
5	Conclusion	111
5.1	Lessons Learned	111
5.1.1	Rely on Dynamic Provenance Tracing	111
5.1.2	Leverage Types	112
5.1.3	Use Labels	113
5.1.4	Embrace Code	113
5.2	Future Work	115
5.3	Summary	118
	References	120
A	Deuce: Bimodal Editing on Expressions	132
A.1	Introduction	132
A.2	Methodology	133
A.3	Results	135
A.4	Related Work	140
A.5	Conclusion	140

Acknowledgments

Personally Thanks to my family for being a constant and reliable support. A special thanks to my parents for loving me and pushing me towards the opportunities in life that led me to this place! Thank you also to my Living Hope Church family for being the closest to a family that I’ve ever experienced in a church, and thanks to my Graduate Christian Fellowship friends for your friendship and fellowship. I think we have all been blessed by each other and I treasure that dearly.

Financially I am indebted to the American taxpayer and hope the research contributions herein warrant the cut of their labors of which I have been the beneficiary. This material is based upon work supported by the U.S. National Science Foundation under grant number 1651794 and the University of Chicago. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the its funders.

Professionally Among my professional associates, I would first like to thank the SKETCH-N-SKETCH team. Justin Lubin and Grace Lu implemented the polygon drawing and multi-selection for Deuce. Justin is also to credit for styling the SKETCH-N-SKETCH UI and reworking the user-facing language from a toy Lisp to a cleaner Elm-like syntax. Mitchell Spradlin and Jacob Albers contributed to the initial version of SKETCH-N-SKETCH [25]. Thanks to Adam Shaw for inspiring the “Rails” example.

For helpful (and/or amusing) discussions throughout the course of this work, I extend my thanks to Kavon Farvardin, Joseph Wingerter, Teo Collin, Andrew McNutt, Kartik Singal, Cyrus Omar, Mikaël Mayer, and Mariana Mărașoiu. Charisee Chiw also deserves credit as a key social instigator for the UChicago PL group.

Philip Guo, Elena Glassman, Aaron Elmore, Peter Scherpelz, and Byron Zhang offered additional feedback on portions of the this work—thank you! Additional thanks to Matt Teichman for incidental assistance.

At the Midwest PL Summit 2019, a random conversation with Brian Howard pointed me to Hanna’s Vital [56, 57]—thanks for the JAR!

Many thanks to my thesis committee, John Reppy, Shan Lu, and Blase Ur, for shepherding this work.

My Advisor Finally, I extend special thanks to my advisor and mentor Ravi Chugh. Direct manipulation of output and the SKETCH-N-SKETCH project were his ideas, and he let me run with them and provided much needed help and support all the way. He was also proactive in encouraging me into professional activities to help me integrate with the PL community. Ravi is the most gracious advisor any grad student could hope for!

Abstract

Direct manipulation is everywhere. While the intuitive *point-click-operate* workflow of direct manipulation is the standard mode of interaction for most computer applications, for over half a century one important application has remained a text-based activity: programming. Can the intuitive workflow of direct manipulation be applied to programming—could programming become as simple as manipulating the program’s output, showing the computer what you want it to do? Alas, 45 years of research on this “programming by demonstration” (PBD) vision has yielded only niche successes.

To confront this impasse, this dissertation reverses a key assumption of PBD systems. Traditional PBD systems eschew textual code, assuming that textual code is difficult for users. But, whatever its faults, textual code is a proven paradigm for understanding and editing programs. Therefore, this work instead embraces textual code: we start with text-based programming in a generic programming language and, rather than replace text, augment it with PBD-style direct manipulation on visualized program outputs. Output manipulations induce changes to the textual code. Such a system is bimodal: at any time, users may program via text edits on code or via mouse manipulations on outputs.

To explore the expressiveness of this bimodal approach, this work presents two programming systems. The first system, called SKETCH-N-SKETCH, mimics a traditional graphics editor, enabling users to use standard drawing interactions to create programs that output vector graphics. The second, called MANIPOSYNTH, brings output-based interaction closer to ordinary programming, offering a graphical interface for constructing OCaml programs that operate on functional data structures. We show the expressive extent of direct manipulation in both systems through examples. Overall, this work expands and illuminates the capabilities of bimodal programming.

Chapter 1

Introduction

As originally defined by Ben Shneiderman, direct manipulation is the workflow characterized by “visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest” [135]. Because of these intuitive affordances, direct manipulation has become the expected way that most people interact with computers. Drawing software, presentation applications, and word processors all function by using the cursor to directly manipulate visual representations of the objects of interest.

But what about programming? Why can’t programs similarly be created by direct manipulation? If direct manipulation could be applied to programming, it might allow novices to learn programming more easily. And even for experts, program text can be opaque and clumsy—direct manipulation might also let them more naturally complete some tasks.

Because direct manipulation promises benefits to both novices and experts, many systems have explored different ways to add direct manipulation to programming. Most of these systems provide direct interactions on the program’s *abstract syntax tree (AST)* but not on the products of the program—its output. And while a few system allow direct manipulation of program output to change the program’s code, these manipulations are limited to specific scenarios. Overall, **it is unknown how expressive direct manipulation of output can be for creating programs**. Therefore, the aim of this dissertation is to vastly expand the answer to the question:

What kinds of code can be created via direct manipulation of program outputs?

1.1 Bimodal Programming

In particular, to leverage the success of text-based programming, we seek to *augment* rather than replace text. The programmer’s direct manipulations will be realized as textual code that is always text editable. Program changes that cannot be accomplished by direct manipulation can still be achieved by ordinary text editing. Furthermore, text edits will not disable the later use of direct manipulation. We call this workflow *bimodal programming*. Bimodal programming is the interaction paradigm in which a programmer freely intermixes text edits with direct manipulation of output in order to craft a program. We answer the question above with the following thesis:

Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-based interactions in a bimodal programming environment.

We justify this thesis by building and demonstrating two different bimodal programming systems, along with one supporting technical mechanism in between. The first system—called Sketch-n-Sketch [59, 63]—is a development environment for creating programs that output SVG vector

graphics (Chapter 2). Looking forward to manipulation of non-visual data structures, we introduce a mechanism—called Tiny Structure Editors (TSE) [60]—for manipulating program values by interacting with their toString representation (Chapter 3). The final system—called Manipsynth [in submission]—is an environment for bimodal programming of ordinary functional data structures (Chapter 4).

Before presenting these systems, we will survey related programming techniques (Section 1.2) and argue that bimodal programming avoids common pitfalls of prior systems (Section 1.3). At the end, we briefly reflect on these systems and conclude (Chapter 5).

For interested readers, Appendix A supplements the above with a study of a different sort of bimodal programming: structural direct manipulation of program *expressions*, rather than values. Such direct manipulations of program expressions are usually offered only in special structure editors rather than in ordinary text buffers, but in the system presented in Appendix A, named DEUCE, adds structural selection and operation to an ordinary text buffer to streamline the invocation of code refactorings. Direct manipulation of code expressions is orthogonal to direct manipulation of output values—a future programming system might incorporate both.

1.2 Related Work

Bimodal programming starts with an ordinary, always-editable text representation of a program and augments the text with both live display of program values *and* the ability to perform program edits by directly manipulating those values. A number of programming systems, surveyed below, share one or more of the above features. These systems may be divided into three categories: (a) *live programming* systems that display program values but require edits to be performed in text, (b) *direct manipulation programming* systems that offer various direct manipulation interactions for program construction, albeit short of the bimodal vision, and finally (c) *bimodal programming* systems, *i.e.*, those systems that offer manipulations of program values while simultaneously representing the program as always-editable text.

1.2.1 Live Programming

Programmers often want to see values their program produces; either values in the program’s output, or, perhaps via a debugger, values at some intermediate state of the program. To see output values after an edit, a programmer must manually trigger and wait for their program to compile and run. Or, to see intermediate values, they must set up a debug session. These repeated operations have the potential to become tedious. The goal of *live programming* is to tighten the programming feedback loop by automatically updating displayed values in response to program changes. As envisioned by Tanimoto [143], the greatest level of liveness is when the visual display of the program state reacts continuously to all program edits as well as to any incoming streaming data.

A large number of live programming systems have been created. Python Tutor [53, 54] is popular teaching tool for visualizing Python program state and includes a live programming mode. Victor’s Inventing on Principle presentation [148] demonstrates several live programming environments and served as inspiration for later work [77, 90]. The LightTable IDE [50] obtained

hundreds of thousand dollars of crowd funding [24]; its key selling points were immediately available documentation and a live updating display of values flowing through code. All these systems live-update the code display in response to program changes. New live programming ideas continue to be explored at the yearly LIVE Programming Workshop [1].

Empirical validation of live programming Is live programming effective? Hancock [55] notes that, intuitively, live programming is like trying to hit a target with a water hose instead of a bow and arrow—the feedback and reaction are continuous. From this intuition we might assume that programmers benefit from seeing the values produced by their code as soon as possible. Alas, this assumption has not yet been empirically validated.

Fabry [41] surveyed four studies and found these existing studies of live programming lacked quantitative evidence for its effectiveness (a conclusion also reached by Rein et al. [128]). For example, for program repair tasks in the spreadsheet programming system Forms/3 with and without live re-evaluation, although participants felt more confident with live feedback enabled, they were no more accurate [154]. And Krämer et al. [82] found that live feedback in Javascript did not help users complete tasks faster (they were, however, faster to fix bugs they introduced during program construction). Even so, despite the lack of quantitative evidence, participants often report qualitative preferences for live programming [41]. Although more recent than Fabry’s survey [41], Huang et al.’s [69] large study of 237 students came to somewhat similar results: live feedback had no effect on learning outcomes, even though students rated live feedback as helpful. For some subtasks, however, students *were* significantly faster with the live feedback. The subtasks with the most pronounced speed differences happened to be code tracing tasks, a kind of task in which live feedback might be expected to help novice programmers who are not as practiced at simulating the computer in their head.

Although live programming has not yet been shown to be broadly helpful, a couple qualitative insights about live programming system design have emerged. Via pilot studies, Kang and Guo [76] observed that two seconds of delay was a good tradeoff between maintaining liveness and reducing the distraction of UI updates. In contrast, Lerner [90] found that immediately live information was not necessarily distracting, although user customization to filter the displayed information is necessary to achieve the best effect.

Despite lack of quantitative evidence, we rely on the qualitative feedback and assume live programming is desirable. Moreover, the bimodal vision seeks to make live programming *more live* by offering direct manipulation of the execution products. Although in this work we do not quantitatively investigate the possible benefit of bimodal interaction on top of live programming, there is a possibility that the addition of bimodal interactions could lead to quantitative improvements in certain scenarios.

1.2.2 Direct Manipulation Programming

Visual dataflow programming. In his 1966 Ph.D. thesis [142], William Sutherland introduced a direct manipulation computer system to perform what is now known as *visual dataflow programming*. In visual dataflow programming, the user graphically creates a computation by laying out nodes and wires on a canvas, much like a flowchart. Nodes represent operations on data, and wires

carry data between operations.

Although the wires can quickly become noisy and resemble literal “spaghetti code” [153], visual dataflow programming has been practically applied to domain-specific tasks. A notable example is the commercially successful LabVIEW [110] environment which primarily targets engineers and technicians working with electronics.

While most nodes-and-wires dataflow programming systems use nodes to represent operations and wires to represent data, the reverse is also possible. PANE [66] is a recent such example: nodes display example values and wires represent transformations between values. Because the example values are foremost in the display and may be clicked to invoke operations on them, PANE’s workflow bears resemblance to the MANIPOSYNTH system presented in Chapter 4. PANE does not, however, maintain an editable text representation of the program.

The visual data processing system Enso [39] (formerly Luna) displays both the operation *and* output on nodes. Additionally, like the systems presented in this dissertation, Enso is a bimodal environment offering an always-editable text representation of the program. Enso does not offer direct manipulation of the displayed output values, however.

Blocks and other structure editors. In block-based programming environments [11], traditional syntactic constructs such as statements, loops, and if-then-else structures are represented as graphical *blocks* that can be directly manipulated. Syntactically valid combinations of blocks snap together like puzzle pieces, allowing the programmer to build up a program with minimal keyboard input. Additionally, a toolbox of available blocks is provided so that new functionality can be discovered and immediately added to the program. Because blocks obviate the need to memorize syntax and can be used without being adept at keyboard input, block-based environments have been used in computer science education, most notably in Scratch [129] and Alice [26].

Block-based editors are a specific instance of *structure editors*, also known as *projectional editors*. In structure editors, instead of editing the program via a raw text buffer, the system offers tree transformations in order to maintain a syntactically valid program throughout its construction. Structure editors do not necessarily operate by direct manipulation. For example, the Cornell Program Synthesizer [144] operated via key commands rather than by a mouse. And, although inspired by the traditional textual rendering of the program, structure editors also differ in how closely they support ordinary raw-text editing. Several structure editors combine both direct manipulation interactions together with a more familiar raw-text editing experience—a combination we also seek. Barista [81] and Greenfoot [18] offer drag-and-drop structural interactions while also mimicking an ordinary text buffer for ordinary editing. Similarly, Deuce [62] (Appendix A) augments an ordinary text buffer with a structural multi-selection mode to quickly invoke refactorings. Unlike the work described here, in structure editors the programmer directly manipulates the code rather than the output.

Constraint-oriented programming (COP). Following in the footsteps of Sketchpad by Ivan Sutherland [141], *constraint-oriented programming (COP)* systems explicitly view building a constrained system as a programming task [15, 64, 42]. In these systems, the programmer declares a series of constraints, either graphically (via direct manipulation) or in text. In the graphical setting, the constraints are common geometric assertions, *e.g.*, “these points should be equidistant from

this other point”; while a non-graphical constraint might be “ x should always be twice the value of y ”. Unlike traditional programming, COP systems run a constraint solver alongside the program, querying the solver *during runtime* to affect the execution of the program. The systems in this dissertation instead follow a standard execution model—any “constraints” are expressed as ordinary math in the program (*e.g.*, $x_2 = x_1 + w/2$) and are executed normally.

Several systems targeted at visual design also follow a COP model but do not seek to expose ordinary text-based code (*e.g.*, [156, 73]). Of these systems, Apparatus [131], Recursive Drawing [130], and Geometer’s SketchPad [72] are notable for supporting recursion.

Related to COP systems, feature-based parametric CAD editors record user actions as a series of steps that together act as a program encoding the creation of the design. Elements may be parameterized based on previously created elements (*e.g.*, a screw head may be defined to be 1.5x wider than the screw cylinder). If an element property is changed, dependent actions in the sequence are re-run to update the design. Thus, the design may be considered a sort of declarative program. Among CAD systems, EBP [122] is notable for allowing the user to perform a step-by-step demonstration to create loops and conditionals.

Programming by demonstration (PBD). To offer end-users some of the benefits of programming, a class of interactions dubbed *programming by demonstration (PBD)* [28] allow users to, instead of typing out code, specify programs by demonstrating the desired actions to the computer. Taking the role of a learner, the computer infers the intent of the demonstrated actions and constructs a program.

Several early PBD approaches used shape drawing as a domain for exploring these non-textual programming techniques. PBD systems usually rely on a visual representation of the program rather than a textual one (*e.g.*, [83, 92]), or show actions step-by-step [98].

Although not as visual as peer PBD systems, Tinker [93] is notable for supporting recursion by demonstration—indeed, any Lisp expression may be created. Unlike the systems presented in this dissertation, manipulations are performed on a symbolic representation of the example not far removed from the underlying Lisp. But, like the systems presented here, the underlying code in Tinker is set in a traditional programming language and that code is featured in the UI.

More recently, PBD techniques have been developed with a practical bent. These systems address a wide variety of domains, such as data visualization [149], mobile applications for collaboration [37], web task automation [94], web scraping [20], and API discovery [157]. Each of these systems focus on solving a particular domain task and, unlike the systems presented here, either do not expose the program as plain text in ordinary code, or do not offer demonstration-based editing after the initial program is generated.

1.2.3 Prior Bimodal Systems

Bimodal programming is defined above as the activity of specifying always text editable code via direct manipulation of program execution products. The systems presented in this dissertation are not the first bimodal systems, although our goal is to push bimodal programming further than prior work. Most prior bimodal systems only support “small” changes to the code via direct manipulation. For example, output manipulation may change numbers [25, 84, 100, 45],

strings [152, 133, 84, 100], or list literals [100] in the code. That is, direct manipulation can make minor tweaks to an existing program but does not help create the program to begin with. A handful of prior systems, however, like those we present, do enable “larger” program changes via output manipulation and can assist the programmer in creating and refactoring the program, not just modifying constant literals. These works will be discussed later in context of the presented systems.

1.3 Avoiding the Pitfalls of PBD

As discussed above, a long line of work has attempted to make programming a more direct and immediate experience than simply typing out opaque code. Programming by demonstration (PBD), in particular, has put forward many ideas over many years for how to make programming more demonstrational, and therefore more approachable, than the arcana of text-based programming—by 2003, there were already two full volumes reviewing the major works to date [28, 91]. Despite all this work, PBD interfaces are rare in practice. Why?

Reflecting on her many years working on PBD systems, Tessa Lau offers five guidelines that PBD systems often neglect [86]. Below, we recount Lau’s principles and argue how bimodal programming might satisfy them.

1. “Detect failure and fail gracefully.” In general, PBD systems accept a series of demonstrations from the user and attempt to *infer* the intended program. Because this inference may involve a complex constraint satisfaction problem, when a wrong program is produced it can be unclear why—in the limit, the constraints may simply be unsatisfiable and the system gives no hint about where the problem might lie. Inference is not fundamental to PBD, however. Each interaction could instead enact a limited change to the code. Instead of constructing a large constraint satisfaction problem over several demonstrations, steps can be small and immediate, with the effect of each step visible and reversible. Where there is ambiguity about the meaning of an interaction, the programmer can choose the desired result before continuing. Because of its clarity and reversibility, we adopt this step-by-step approach for our bimodal systems.

2. “Make it easy to correct the system.” When a demonstration does not produce the desired result, the only fix may be to redo the entire demonstration. But if operations are instead stepwise and small, only a little work is lost if the system misinterprets a user interaction. More importantly, because the code is always editable as text in our bimodal systems, the programmer is not forced to fumble endlessly trying to discover the appropriate interaction: if their interaction fails, they can still enact the change with ordinary text editing.

3. “Encourage trust by presenting a model users can understand.” If the representation of the program is hidden, or is an inscrutable AI model, users will not be able to trust that their program does what they want. In bimodal programming, the program is ordinary, readable code. What the program does is precise and knowable.

4. “Enable partial automation.” Instead of forcing the programmer to always manipulate output, bimodal programming allows the programmer to mix and match direct manipulation with traditional text-based programming.

5. “Consider the bottom-line value of automation.” Lau admonishes the system designer to weigh not just the user effort required to perform a single demonstration, but also the additional costs of learning to use the system and the switching cost of leaving one’s normal workflow to enter the demonstrational interface. On this point bimodal programming has less to offer. Certain bimodal interactions might go unused, despite being notionally superior to their textual counterparts. For example, in SKETCH-N-SKETCH below it is easier to draw a shape on the canvas rather than remember how to type the appropriate function call in the code. Even so, a programmer already typing text may choose to add the function call via the keyboard. A similar phenomena has been shown in traditional integrated development environments (IDEs). IDEs have long had a large suite of labor-saving automated refactoring tools, and yet an open problem in the software engineering community is *how* to get folks to use those tools—one study found that even when automated tools were available, up to 90% of code refactorings were performed manually [109]. Unlike refactoring tools, however, if a bimodal system offers enough bimodal interactions to cover most of a programmer’s workflow, for example by also including tools for program *construction* not covered by standard refactorings, then it may be possible to switch the programming workflow from a primarily text-editing activity to a primarily direct manipulation activity. As evidence of this possibility, text edits in the code editor will not be used in the presentations of the SKETCH-N-SKETCH and MANIPOSYNTH systems below.

As outlined above, bimodal programming addresses many of the problems faced by PBD systems. Why hasn’t bimodal programming been more thoroughly explored? There are at least two challenges. First, maintaining two editable representations is non-trivial—most prior bimodal systems therefore only support code changes in limited scenarios as noted in Section 1.2. Second, raw output from a program may hide *how* that output was generated, but the *how* may be what the programmer wants to manipulate—*e.g.*, a large program that outputs a single number cannot be effectively modified just by indicating a desired change to that number. Because these are not small challenges, the systems presented in the following chapters focus on expanding the expressive power of bimodal programming, *i.e.*, demonstrating new possibilities of what kinds of code can be created with bimodal interactions. Hence the thesis:

Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-based interactions in a bimodal programming environment.

To scope this work, we aim to evaluate expressivity only—what kinds of programs can be created—and do not directly assess usability. We assume users are expert programmers familiar with code and that, for the reasons outlined above, bimodal interactions are desirable. Expressivity will be demonstrated by creating example programs entirely by output-based interactions, without ordinary text edits to the code (though, of course, these edits remain possible).

The next three chapters introduce the three environments by which we explore bimodal programming—SKETCH-N-SKETCH, TINY STRUCTURE EDITORS (TSE), and MANIPOSYNTH—followed by a wrap-up chapter to conclude.

Chapter 2

Sketch-n-Sketch: Bimodal Programming for SVG

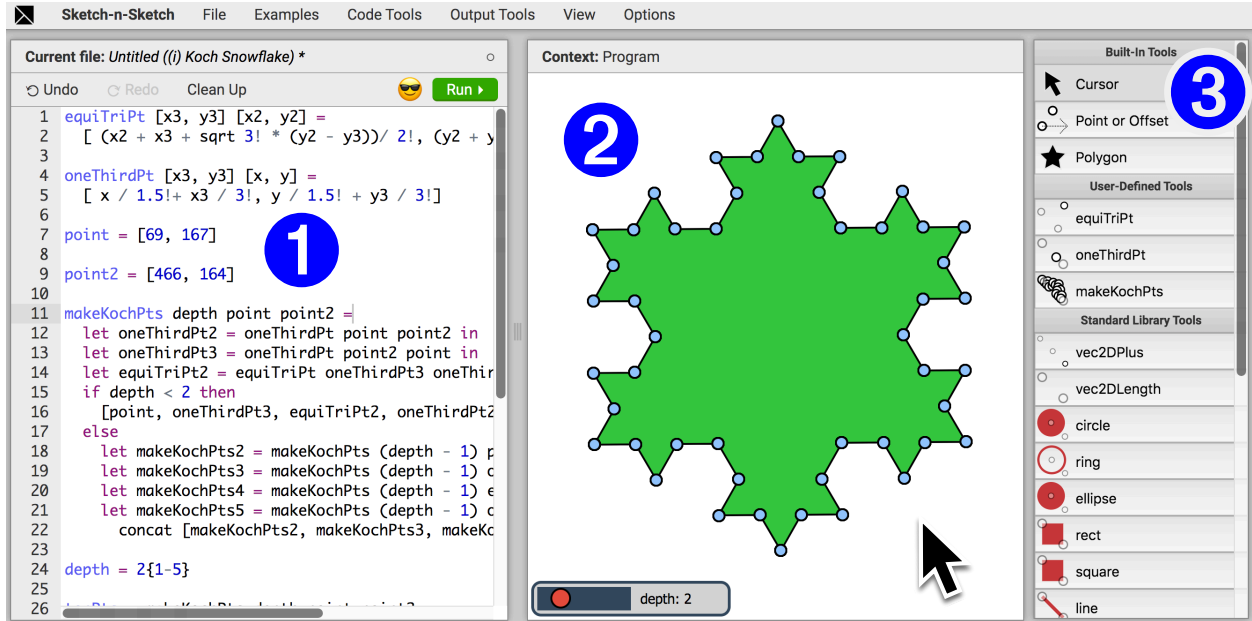


Figure 2.1: SKETCH-N-SKETCH interface.

2.1 Introduction

Visual designers face a choice: they must choose between using graphical editors or programmatic tools. Graphical editors offer convenient direct manipulation, but their capabilities are circumscribed by the application. Programmatic languages and libraries like Processing [12] instead offer near-infinite flexibility, but the affordances of direct manipulation are lost. Could designers have both the flexibility of programming and the power of direct manipulation?

In this chapter we introduce SKETCH-N-SKETCH, a graphical editor for creating programs that output SVG vector graphics. SKETCH-N-SKETCH imitates a traditional graphics editor, with tools

This chapter encompasses work published at UIST 2016 (Hempel and Chugh [59]) and UIST 2019 (Hempel et al. [63]). SKETCH-N-SKETCH was awarded a Best Demo Honorable Mention at UIST 2019. The most concise explanation of SKETCH-N-SKETCH is the UIST 2019 paper [63], from which this chapter borrows. Compared to [63], this chapter adds considerable detail about technical implementation mechanisms.

for directly drawing and manipulating shapes. The source code of the drawing, however, is an ordinary text-based program in an Elm-like [40] functional programming language, allowing for parametric designs involving repetition, duplication via function abstraction, and alignment via variable sharing.

We want to see how far this idea can be taken—what kinds of programs can be created only via direct manipulation on the program’s graphical output? To explore this question in SKETCH-N-SKETCH we:

1. Provide direct manipulation tools for *drawing, relating, grouping, abstracting, and repeating* shapes.
2. Expose *intermediate execution values* for manipulation, in addition to the final output.
3. Offer *focused editing* to enable contextual drawing.
4. Expose *generic code refactoring tools* through output-based interactions.
5. Use runtime tracing to track *value provenance*, to associate output selections with source code locations.

Even though text-based editing remains available at any time, we show how these techniques in SKETCH-N-SKETCH allow a number of designs to be created *entirely* through direct manipulation. We show you can have both direct manipulation and programmatic flexibility within the same system.

To begin, we survey the most closely related bimodal systems for creating graphics-drawing programs in Section 2.2. In Section 2.3, we introduce SKETCH-N-SKETCH’s workflow with a main example, followed by three shorter examples to present additional tools. Then, in Section 2.4, we catalog and describe the myriad technical mechanisms that SKETCH-N-SKETCH uses to offer its bimodal interactions—although SKETCH-N-SKETCH is specialized for programs that output vector graphics, many of these technical mechanisms are relevant for future bimodal programming systems in other domains. To explore SKETCH-N-SKETCH’s expressiveness, we created 16 example programs which are presented in Section 2.5. Possible future work is explored in Section 2.6 and we reiterate SKETCH-N-SKETCH’s merits to conclude in Section 2.7.

2.2 Related Graphical Bimodal Environments

While a handful of systems have demonstrated the ability to drag shapes around to enact minor changes on an existing program [45, 25, 84, 78], the ability to enact larger changes to *construct* a program—drawing new shapes or refactoring code—is less common. Like SKETCH-N-SKETCH, two prior systems offer graphical construction capabilities while realizing the direct manipulations in plain, editable code.

Transmorphic [132] re-implements the Morphic UI framework [97] using static, functional (*i.e.*, stateless) views. Transmorphic retains Morphic’s ability to directly manipulate morphs (*i.e.*, UI elements), but affects the manipulations by changing the view’s text-based code rather than the usual Smalltalk mechanism of changing live object state. In Transmorphic, the programmer may use direct manipulation to add morphs, remove morphs, or change a morph’s primitive properties.

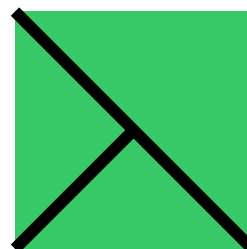
To implement its transformations, Transmorphic associates each visual morph with a syntactic location in the program via a static analysis pass, whereas SKETCH-N-SKETCH instead uses dynamic runtime tracing.

Like the work presented here, APX [101, 103] is a two-pane (code box and output canvas) environment for creating programs that draw pictures. APX additionally supports creation of real-time visual simulations, updated live as the programmer edits their code. On the output canvas, APX supports direct manipulation of, *e.g.*, shape position and size, thus changing numbers in the program. A few larger changes—namely, grouping and insertion of new shapes—are supported as well, although most of APX’s interactions are focused on refactoring code by directly manipulating program terms in the code box.

Compared to Transmorphic and APX, the tools in SKETCH-N-SKETCH address considerably more use cases. In particular, the relation, abstraction, refactoring, and repetition capabilities described below are novel to SKETCH-N-SKETCH. Moreover, neither of the above two systems envisioned that their direct manipulation tools alone could be sufficient for their target domains—both present their workflows as mix of text-based coding and output-based interaction. While SKETCH-N-SKETCH *supports* this mix and match, in order to highlight the expressivity of its tools, the presentation of SKETCH-N-SKETCH here only discusses programs constructed *entirely* by operations on the canvas, without any text-based programming in the code box.

2.3 Overview

SKETCH-N-SKETCH aims to imitate a simple traditional vector graphics editor, but with the design realized as textual code in a programming language. The work presented here builds on a prior initial version of SKETCH-N-SKETCH [25] in which, once a program had been created by text edits, the user could use their mouse to directly move and resize shapes and change colors, all on the program’s output canvas. These manipulations were reified by automatically changing appropriate numbers in the code. Output-based manipulations in the prior work could only modify single numbers in the program. The present work introduces novel program construction and editing facilities, all via mouse-based manipulations on the program’s output canvas.



Below, the interactions in SKETCH-N-SKETCH are introduced through the construction of several example programs. First, we will introduce the main workflow and most commonly used tools in SKETCH-N-SKETCH by demonstrating the construction of the SKETCH-N-SKETCH logo, shown in the inset above. Afterward we will demonstrate other tools by briefly describing key steps in the construction of three further examples (including the recursive fractal in Figure 2.1). Having introduced how the tools are used, Section 2.4 will detail the mechanisms by which all the tools operate.

Figure 2.1 shows the SKETCH-N-SKETCH interface. The code box ❶ on the left is an ordinary source code text editor. The language in SKETCH-N-SKETCH is a simple, functional programming language—an extended lambda calculus with syntax inspired by the Elm language [40]. The canvas ❷ on the right displays the program’s SVG output. The toolbox ❸ offers various drawing tools

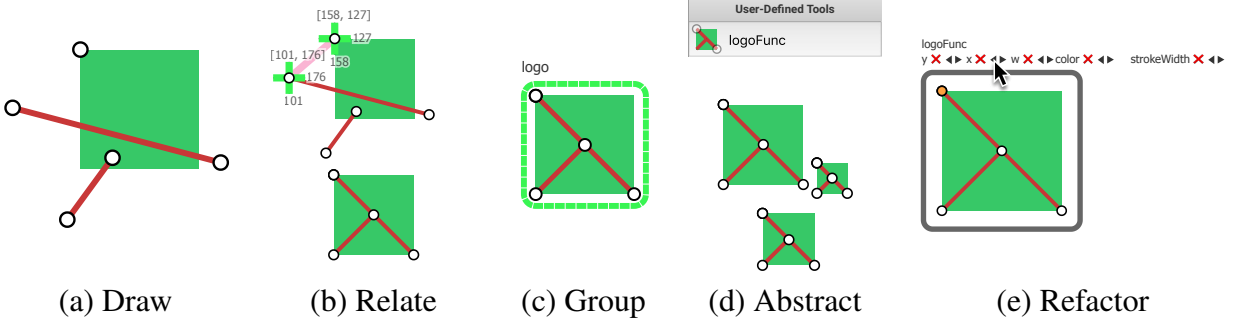


Figure 2.2: SKETCH-N-SKETCH workflow.

for adding items into the program. The particular program shown in Figure 2.1 uses a recursive function to draw a von Koch snowflake fractal [151]. This program was constructed *entirely* with interactions on the output. Indeed, although we may perform regular text-edits to our programs at any point during their construction—and we may do so without losing access to the output-directed tooling for later edits—all examples below will be constructed using *only* output-based edits, thereby highlighting the expressive power of SKETCH-N-SKETCH’s tooling.

The paradigmatic SKETCH-N-SKETCH workflow is summarized in Figure 2.2, which the construction of the logo example will follow. After first *drawing* the needed shapes, the programmer will *relate* properties of the shapes (*e.g.*, constraining the endpoints of the lines to match the corners of the square), and afterwards the shapes can be gathered into a *group*. Because the design is a program, the programmer can *abstract* the shapes into a function, allowing them to reuse their design. Finally, the programmer may *refactor* the program, to, *e.g.*, clean up variable names and choose which shape properties should be arguments to the function. The final code for this example is shown in Figure 2.5. We walk through these steps below.

2.3.1 Draw

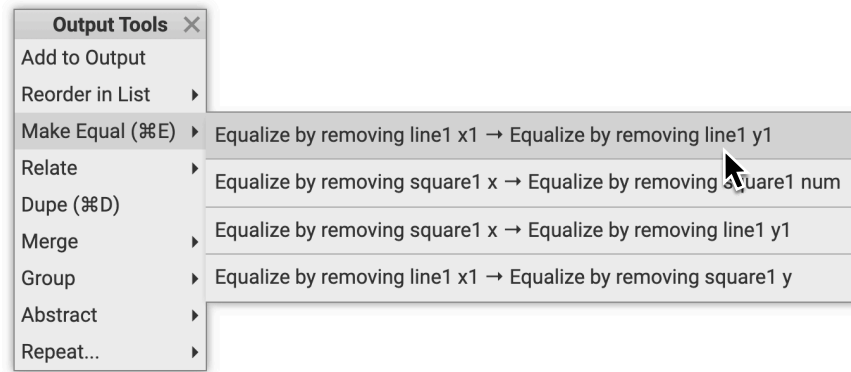
The initial program template provided by SKETCH-N-SKETCH is nearly blank, defining only an empty list of SVG shapes (the last expression of a program defines the return value for the whole program):

```
svg (concat [
])
```

As in a traditional graphics editor, the programmer clicks on the “square” tool from the toolbox ③ and drags their mouse on the canvas. A new `square1` definition is added to the program, sized and positioned in accordance with the mouse movement, and the `square1` variable is added to the shape list so that the square appears in the output.

```
square1 = square 0 [158, 127] 156

svg (concat [
  [square1]
])
```



(a) Equalizing points

```

1
2 square1 = square 110 [230, 229] 333
3
4 line1 = line 0 5 [161, 303] topLeft = [230, 229]
5
6 square1 = square 110 topLeft 333
7
8 line1 = line 0 5 topLeft [561, 475]
9
10 line2 = line 0 5 [109, 668] [303, 468]
11
12 svg (concat [
13   [square1],
14   [line1],
15   [line2]
16 ])

```

(b) Change diff

Figure 2.3: The Output Tools panel appears when an item on the canvas is selected. Each tool may offer multiple results; hovering the mouse over each result previews the change on the code (b) and on the canvas.

The programmer similarly uses the “line” tool to add the needed lines (Figure 2.2a).

2.3.2 Relate

The programmer would like to require that the endpoints of the lines always coincide with the corners of the rectangle. To relate these positions, the programmer first double-clicks the two points they would like to be coincident. The x and y coordinates of these points are selected, shown as green crosses (Figure 2.2b, top; the pink line is an unselected distance feature, introduced in the next example in §2.3.6). Whenever a selection is made upon the canvas, SKETCH-N-SKETCH displays a floating menu of output tools, offering operations on the selected items (Figure 2.3a).

To snap the points together, the programmer chooses the MAKE EQUAL tool. A submenu offers multiple ways to introduce variables into the program so that the points are always in the same position. The results differ in which original position is preserved—should the line move to the square, or vice versa? The programmer may hover their mouse over each result to see its output on the canvas and the diff in the code (Figure 2.3c). The programmer chooses the first result, which

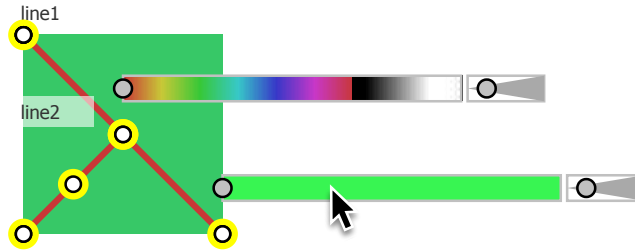


Figure 2.4: When a shape is selected, sliders appear to modify its non-spatial properties, such as color and stroke width. These properties can be selected by clicking the sliders (drawn in green when selected).

moves the line to the square. A `topLeft` variable is introduced and used for both the positions of the square and the line:

```
topLeft = [158, 127]

square1 = square 140 topLeft 100

line1 = line 0 5 topLeft [276, 222]
...
```

If the programmer moves the square with their mouse, the numbers of the `topLeft` point will change and the endpoint of line, because it uses the same variable, will stay pinned to the square's corner.

The remaining endpoints can be similarly related with `MAKE EQUAL`; although the positions of the other corners and center of the square must be calculated—for example, the right edge of the square should be `x + w`. `MAKE EQUAL` automatically inserts the needed math (visible on lines 11-13 in the final code in Figure 2.5).

Non-spatial properties may also be equalized. When the programmer selects the two lines, sliders appear to modify their color and stroke width. The sliders themselves may be clicked to select the corresponding property (Figure 2.4); after selecting both sliders, the programmer applies `MAKE EQUAL` on the colors so both line colors are tied to a single variable in the code.

2.3.3 Group

Analogous to the grouping functionality of traditional graphics editors, `SKETCH-N-SKETCH` offers a `GROUP` tool to gather shapes into a single list. The programmer selects the three shapes and invokes `GROUP`, resulting in a new `squareLineLine` list in the code:

```
...
squareLineLine = [square1, line1, line2]

svg (concat [
  squareLineLine
])
```

Lists are represented on the canvas as a dotted border encompassing the list items (Figure 2.2c, Figure 2.18c). To aid comprehension, variable names are also shown on the canvas next to appropriate items—double-clicking on name allows the variable to be renamed. In Figure 2.2c, the programmer has selected the squareLineLine group list and renamed the variable to logo.

2.3.4 Abstract

The programmer would like to make their logo design reusable—in other words, they would like to create a *function* that, given size and color arguments, generates a logo appropriately. With the logo group list selected, the programmer invokes ABSTRACT from the Output Tools menu. ABSTRACT performs an ordinary Extract Method refactoring to produce a function that returns the selected item, namely [square1, line1, line2]. As a heuristic, ABSTRACT (recursively) pulls in variable bindings used only in the construction of [square1, line1, line2] (provided those definitions have free variables themselves) resulting in a function named logoFunc parameterized over y, x, w, color (of the lines), and strokeWidth:

```
...
logoFunc y x w color strokeWidth =
  let topLeft = [x, y] in
  let square1 = square 140 topLeft w in
  let y2 = y + w in
  let line1 = line color strokeWidth topLeft [x + w, y2] in
  let line2 = line color strokeWidth [x, y2] \
    [(2! * x + w) / 2!, (2! * y + w) / 2!] in
  [square1, line1, line2]

logo = logoFunc y x w color strokeWidth
...
```

The logo design is now reusable. To insert more copies of the design, the programmer could manually write additional calls to this function into the program. Conveniently, however, SKETCH-N-SKETCH's type inference notices that this function accepts at least an x, y coordinate and a width, and this function automatically appears in the toolbox as a custom drawing tool (Figure 2.2d, top; the type inference mechanism is described in §2.4.7). The programmer uses this custom drawing tool to draw two more copies of the function (Figure 2.2d, bottom; Figure 2.5 lines 18 and 20).

2.3.5 Refactor

Although the programmer has produced a reusable design, they may want to change the details of the parameterization. The logoFunc produced by ABSTRACT's heuristics was parameterized on y, x, w, color (of the lines), and strokeWidth, in that order. Copies of the design may differ in those attributes, but all copies must share the same square fill color. The programmer would like the square fill color to differ between copies and also wants x to come before y. SKETCH-N-SKETCH offers interactions to perform these refactorings on the canvas.

```

1 y = 127
2
3 x = 158
4
5 w = 100
6
7 logoFunc x y w fill lineColor strokeWidth =
8   let topLeft = [x, y] in
9   let square1 = square fill topLeft w in
10  let y2 = y + w in
11  let line1 = line lineColor strokeWidth topLeft [ x+ w, y2] in
12  let line2 = line lineColor strokeWidth [x, y2] \
13    [ (2! * x + w)/ 2!, (2! * y + w) / 2!] in
14  [square1, line1, line2]
15
16 logo = logoFunc x y w 140 0 5
17
18 logoFunc1 = logoFunc 275 200 38 140 0 5
19
20 logoFunc2 = logoFunc 207 263 65 140 0 5
21
22 svg (concat [
23   logo,
24   logoFunc1,
25   logoFunc2
26 ])

```

Figure 2.5: Final code for the logo example. Numbers annotated with ! will not change when shapes are moved on the canvas [25].

Items on the canvas that result from a function call are encompassed by a solid border; the border represents the function call. Clicking the border focuses the function call (Figure 2.2e). Other shapes on the canvas disappear and the programmer may edit the function. The arguments to the function also appear, as seen in Figure 2.2e, and may be reordered, removed, or renamed. Arguments may also be added by selecting a property of a shape (*e.g.*, the square color) and invoking ADD ARGUMENT from the Output Tools menu. Although not exercised in this example, focusing a function has a further consequence: using a drawing tool will add the new shape *to the function* instead of to the top level of the program.

In this example, the programmer reorders the *x* and *y* parameters, adds the square color (by selecting the square, selecting its fill color slider, and invoking ADD ARGUMENT), and renames the color parameter to *lineColor*. The final parameterization appears on line 7 of the final code shown in Figure 2.5. The programmer is satisfied. They were able to create the program entirely through output-based manipulations on the canvas by using tools for *drawing*, *relating*, *grouping*, *abstracting*, and *refactoring* their design.

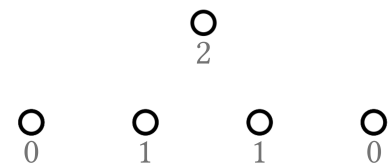
Although this example demonstrated the main workflow for creating designs in SKETCH-N-SKETCH, it did not exercise all the tools available. Three brief examples below introduce the remainder of SKETCH-N-SKETCH's features.

2.3.6 Koch Snowflake Fractal (feat. recursive drawing)

In the logo example above, at each step the programmer manipulated items in the *final* output of the program. In practice, programs may take many steps of computation before producing a final output. In the graphical setting, to allow the user some control over execution steps *before* the final output, SKETCH-N-SKETCH exposes several forms of intermediate execution products on the canvas for manipulation. The prior example introduced two in passing (the dotted and solid borders representing lists and function calls, respectively). To introduce the widgets representing points, along with additional transformation tools, here we discuss the construction of a program that draws a von Koch fractal snowflake [151].

This particular example is notable for two reasons: (a) the program is constructed entirely by operations on intermediates (the program's output is technically empty until the user's final change) and (b) the program involves recursion.¹

Figure 2.6d shows the final design. The design is created by recursively repeating the motif at right. Points labeled 0 are inputs; points labeled 1 will be placed $\frac{1}{3}$ and $\frac{2}{3}$ of the way between the inputs; a final point (labeled 2) will be placed equidistant from the latter, forming an equilateral triangle. Repeating the motif between pairs of points recursively defines the fractal.



To construct this motif and build the overall design, four definitions are required (Figure 2.6):

- (a) A helper function that, given two points, computes a point $\frac{1}{3}$ of the way between them.
- (b) A helper function that, given two points, computes a third point that completes the equilateral triangle with the two input points.
- (c) A function that uses the above two helpers to create the basic motif, *and* recursively repeats the motif within itself. The result is the points of a Koch curve, *i.e.*, one side of the final snowflake.
- (d) A snowflake polygon produced by laying out three instances of Koch curve along the sides of an equilateral triangle.

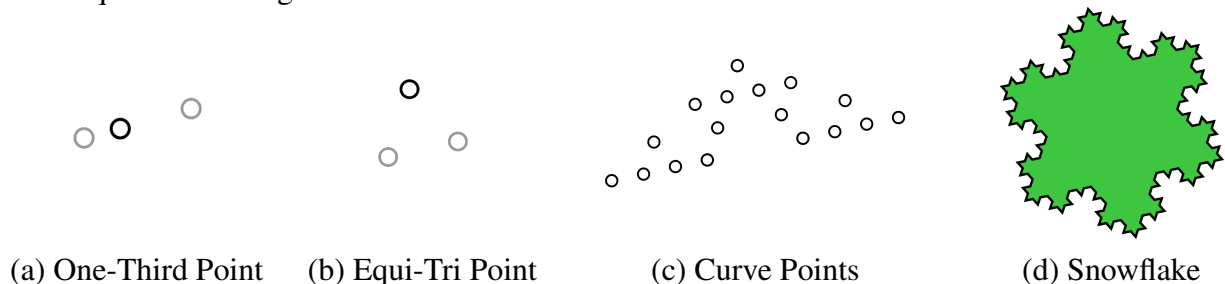


Figure 2.6: Definitions needed to create a Koch snowflake.

¹The supplementary materials at <https://doi.org/10.1145/3332165.3347925> include a narrated video walking through this program's construction.

One-Third Point Function To construct the helper function that takes two points and returns a point $\frac{1}{3}$ of the way between them, We first select the “Point or Offset” tool from the toolbox (Figure 2.1 ③). The “Point or Offset” tool can be used to add new point definitions to the program. Upon clicking the canvas, a new definition is inserted at the top of the program:

```
[x, y] as point = [87, 206]

svg (concat [
])
```

The `[x, y] as point` pattern is destructuring assignment: the `x` variable holds 87, `y` holds 206, and `point` holds the whole two-element list `[87, 206]`. Although these new `x`, `y`, and `point` variables are not yet used—the shape list at the end of the program is still empty—a dot appears on the canvas at (87, 206). SKETCH-N-SKETCH’s evaluator will draw widgets for intermediate execution products on the canvas when certain types of values are encountered during execution. Whenever the evaluator encounters a number-number pair during execution, a *point widget* is drawn as a benign side effect. Point widgets allow points in the program to be selected or manipulated on the canvas, even if that point does not occur in the program’s final output.

Continuing with the “Point or Offset” tool, we add two more point definitions in a similar fashion, and then drag the points with the “Cursor” tool into roughly the right places, with one point $\frac{1}{3}$ of the way between the other two.

```
[x3, y3] as point3 = [264, 131]

[x2, y2] as point2 = [153, 178]

[x, y] as point = [87, 206]

svg (concat [
])
```

We would now like to tell SKETCH-N-SKETCH to *relate* the $\frac{1}{3}$ point (point2 above) in terms of the other two.

After selecting the three points, we hover the mouse over the RELATE tool in the floating Output Tools panel (Figure 2.3a). SKETCH-N-SKETCH attempts to synthesize an arithmetic expression that, were it substituted into the program, would define one of points in terms of the others *and* leave our points in roughly the same place. After about 20 seconds of guess-and-check work, three possible results are shown in the RELATE tool submenu.

▶	point x = x2 * 1.5! - x3 / 2! → point y = y2 * 1.5! - y3 / 2!
	point2 x2 = x / 1.5! + x3 / 3! → point2 y2 = y / 1.5! + y3 / 3!
▶	point3 x3 = x2 * 3! - x * 2! → point3 y3 = y2 * 3! - y * 2!

The RELATE tool synthesized three results²—although depending on where our points were placed, there could be as few as zero or as many as 20 or more results. The second result is mathematically equivalent to what we might deduce by hand (*e.g.*, $x_{out} = x_1 + \frac{1}{3}(x_2 - x_1)$, and similarly for y), so we choose that result. The point2 definition is rewritten, and the point definition—which previously was the last definition—is automatically moved upward so its x and y variables are in scope for point2:

```
[x3, y3] as point3 = [264, 131]

[x, y] as point = [87, 206]

[x2, y2] as point2 = [ x / 1.5!+ x3 / 3!, y / 1.5! + y3 / 3!]
...
```

To turn this concrete expression into a reusable function, we select all the points again and invoke ABSTRACT from the Output Tools panel. In this case there are three possible results: abstracting over only the math for the x coordinate, only the math for the y coordinate, or the entire $\frac{1}{3}$ point. We choose the last result.

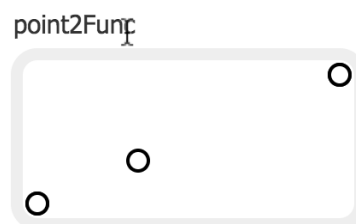
```
[x3, y3] as point3 = [264, 131]

[x, y] as point = [87, 206]

point2Func [x3, y3] [x, y] =
  [ x / 1.5!+ x3 / 3!, y / 1.5! + y3 / 3!]

[x2, y2] as point2 = point2Func point3 point
...
```

A new function—called point2Func—has been placed in our code and the old point2 definition now calls this function to produce its point; as in the logo example, a call widget (gray border at right) and function name appears when we hover over the output point. We click to rename the function to oneThirdPt. Our first helper function is now complete, so we no longer need the example points used to build it. We select all three points and press the Delete key. Perhaps surprisingly, only the definition for point2 disappears, but not the other two points. Only one definition was removed because both of the other points were also involved in calculating point2—DELETE removed something about all three points by discarding the point2 binding. We discuss more details on how SKETCH-N-SKETCH interprets the provenance of values in subsection 2.4.12. Selecting the remaining two points and pressing Delete again removes their two definitions from the program. We are left with our oneThirdPt helper function and an empty shape list.

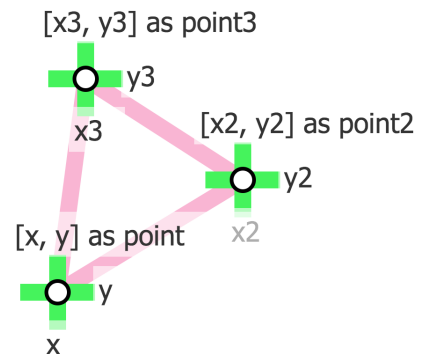


²The *freeze annotations* on 1.5! and 3! indicate that these numbers should not be changed when dragging items on the canvas [25].

Equi-Tri Point Function We would like our second helper function to take in two points and return a point equidistant from both, as if forming an equilateral triangle. As before, we place three points on the canvas in roughly the appropriate positions. Unlike before, the math we need to enforce the points' equidistance is too complicated for the RELATE tool to discover by guessing. Instead, we select the three points and then click the pink lines that appear between the points. These pink lines are *distance features* that represent the distance between the points (shown at right below). Distance features are only shown between selected points to avoid cluttering the canvas.

We invoke the MAKE EQUAL tool with the three distances selected, which queries a solver (REDUCE [58]) to discover how to replace constants in our program with mathematical expressions that enforce the desired equality. In our case, because there are many options for which items might be defined in terms of the others, we are shown a large number of solutions. We choose the second result (only to avoid producing an extraneous offset widget, which are introduced in the tree branch example below).

Now one of the points is mathematically constrained to be equidistant from the other two. As before, we select the points, ABSTRACT the concrete math into a reusable function which we name `equiTriPt`. We delete our example points, leaving the final code for our two helper functions:



```
equiTriPt [x3, y3] [x2, y2] =
  [ (x2 + x3 + sqrt 3! * (y2 - y3))/ 2! \
    , (y2 + y3 - sqrt 3! * (x2 - x3)) / 2! ]

oneThirdPt [x3, y3] [x, y] =
  [ x / 1.5! + x3 / 3!, y / 1.5! + y3 / 3! ]
...
```

Recursive Koch Curve Points Function The principal component of our design is the fractal motif, which we will repeat inside itself to form the points of the fractal. We choose our `oneThirdPt` helper function in the toolbox and draw it on the canvas, resulting in a call to the helper with two new points:

```
oneThirdPt2 = oneThirdPt [65, 199] [235, 141]
```

This call gives us the endpoints of the motif, and one of our $\frac{1}{3}$ points. To get the other, we draw `oneThirdPt` backwards, starting from one endpoint (in green at right, depicted while still drawing) and then ending at other endpoint to *snap-draw* [49] so the existing endpoints are reused instead of adding new points. Snap-drawing is effectively an instantaneous invocation of MAKE EQUAL—the endpoints are pulled out into variables and used for both calls.



```

point = [65, 199]

point2 = [235, 141]

oneThirdPt2 = oneThirdPt point point2

oneThirdPt3 = oneThirdPt point2 point
...

```

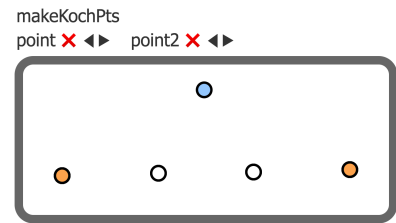
We then switch to our `equiTriPt` helper function and snap-draw it between our $\frac{1}{3}$ and $\frac{2}{3}$ point. The just-created `oneThirdPt2` and `oneThirdPt3` variables are used as arguments to the inserted function call:

```
equiTriPt2 = equiTriPt oneThirdPt3 oneThirdPt2
```

We now have our motif in terms of example points. Selecting our points, we then ABSTRACT. ABSTRACT notices that our $\frac{1}{3}$ and $\frac{2}{3}$ point definitions are only used inside this new function, and so they are pulled into the new function body (as seen in the next code listing below). We name the function `makeKochPts`.

Now, we want to repeat the motif inside itself. If we select `makeKochPts` from the toolbox and draw it on the canvas, SKETCH-N-SKETCH will just insert another call at the top level of the program. Instead, we need the function to call itself *recursively*. We focus `makeKochPts` by clicking on its gray call widget border.

With the function focused, the inputs of our function are colored orange, while the outputs are colored blue. We can draw the function inside itself to create a recursive call. With the `makeKochPts` tool, we snap-draw the function between the first pair of points. SKETCH-N-SKETCH inserts an if-then-else recursive skeleton and the recursive call.



```

...
makeKochPts point point2 =
  let oneThirdPt2 = oneThirdPt point point2 in
  let oneThirdPt3 = oneThirdPt point2 point in
  let equiTriPt2 = equiTriPt oneThirdPt3 oneThirdPt2 in
  if ??terminationCondition then
    equiTriPt2
  else
    let makeKochPts2 = makeKochPts point oneThirdPt3 in
    equiTriPt2
...

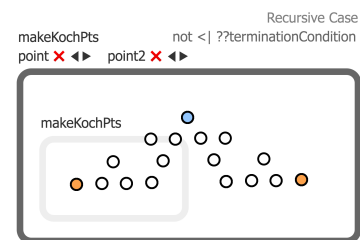
```

To avoid infinite recursion, the if-then-else skeleton branches on a specially named hole, `??terminationCondition`. During evaluation, `??terminationCondition` returns `False` the first time the function is encountered in the call stack, and `True` if the function appears earlier in the call stack, affecting termination at a fixed depth of two. This allows us to run the program and

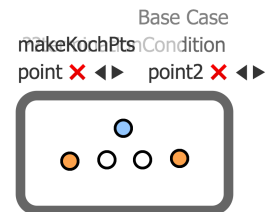
manipulate its output even before we replace the hole expression later. We snap-draw makeKochPts between the remaining three pairs of points; the calls are inserted in the recursive branch.

```
...
if ??terminationCondition then
  equiTriPt2
else
  let makeKochPts2 = makeKochPts point oneThirdPt3 in
  let makeKochPts3 = makeKochPts oneThirdPt3 equiTriPt2 in
  let makeKochPts4 = makeKochPts equiTriPt2 oneThirdPt2 in
  let makeKochPts5 = makeKochPts oneThirdPt2 point2 in
  equiTriPt2
...
```

Our design looks like a fractal now, but the output of the function consists only of equiTriPt2, shown in blue. All the white points are only intermediates. Moreover, most of those intermediates are inside calls to the base case of our function—but we are focused on the recursive case. We must first modify the output of the base case, before finalizing the recursive case. We hover an output point of a recursive call to expose its call widget, whose border we then click to focus the base case.



Focused on the base case (shown at right), we want its output to consist of the leftmost four points of the motif, rather than just the blue equiTriPt2. The fifth point will be provided by the neighboring call to the base case.

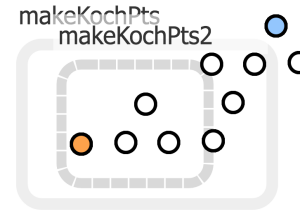


We click-select the three additional points we would like to be in the output, and then invoke ADD TO OUTPUT. In order for the function to output multiple points, the function must now return a *list* of points. The return expressions of both branches of our function are therefore wrapped in lists, and the three selected points are added to the list in the base case.

```
...
if ??terminationCondition then
  [equiTriPt2, oneThirdPt3, oneThirdPt2, point]
else
  ...
  [equiTriPt2]
...
```

Alas, the points are not in the proper order with the list. We must fix the ordering so the polygon we will attach to all our Koch points will be drawn correctly. We select one of our points—which highlights the variable usage in the code so we know where it is in the list—and invoke REORDER IN LIST as necessary to move the point forward, backward, to the beginning, or to the end in the list. When all our points are appropriately reordered in this way, we are done with the base case. We hit Escape to defocus the base case, and then re-focus the recursive case.

The recursive case currently only returns [equiTriPt2]; instead, we need it to combine all the point lists returned from the recursive calls. Recall from the logo example that lists are represented as dotted gray borders which may be selected (shown at right). We select all four returned lists from the calls to the base case and invoke ADD TO OUTPUT, producing the code below:



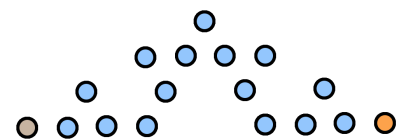
```
...
if ??terminationCondition then
  [point, oneThirdPt3, equiTriPt2, oneThirdPt2]
else
  let makeKochPts2 = makeKochPts point oneThirdPt3 in
  let makeKochPts3 = makeKochPts oneThirdPt3 equiTriPt2 in
  let makeKochPts4 = makeKochPts equiTriPt2 oneThirdPt2 in
  let makeKochPts5 = makeKochPts oneThirdPt2 point2 in
  concat [[equiTriPt2], makeKochPts2, makeKochPts3 \
    , makeKochPts4, makeKochPts5]
...
```

SKETCH-N-SKETCH realizes we are trying to combine lists together and inserts a concat (flatten) call so the function produces a list of points, rather than a list of lists of points. SKETCH-N-SKETCH remembers the order in which we selected the list widgets and inserts the variable uses in the same order, so no reordering is required provided we selected the lists in order. (We can inspect the order by hovering the mouse over each list widget border on the canvas, which highlights the corresponding expression in the code.)

The [equiTriPt] singleton list from the original return value of the function is extraneous; we find and DELETE its list widget, leaving a return expression of:

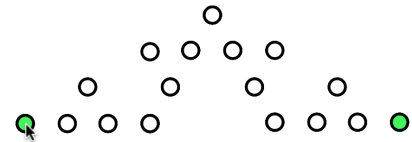
```
concat [makeKochPts2, makeKochPts3, makeKochPts4, makeKochPts5]
```

All the points are now in the output of makeKochPts (and therefore displayed in blue). One last item remains: we must choose a termination condition. The focused call widget for makeKochPts displays the conditional for the recursive case as not <| ??terminationCondition (the not indicates we are in the else case) which we can click to CHOOSE TERMINATION CONDITION. Currently, SKETCH-N-SKETCH offers only one kind of automatically generated termination, fixed depth, which we choose. A depth argument is added to our makeKochPts function which is decremented on the recursive calls (lines 11 and 15 in Figure 2.1). Additionally, the original example call to our function is given a depth of 2 with a {1-5} range annotation so that SKETCH-N-SKETCH draws a slider on the canvas for modifying the depth [25].



```
equiTriPt2 = makeKochPts 2{1-5} point point2
```

Koch Snowflake Polygon Now to make a snowflake! We have a function that produces points for one side of the Koch snowflake. We create an equilateral triangle with `equiTriPt` (shown at right) and then snap-draw `makeKochPoints` along its sides.



To make a single list of *all* the points, we select the three list widgets for the three sides' points and invoke `GROUP`. `GROUP` means *gather into list*; here it offers either to make a list of lists *or* to concat (flatten) all our lists together. We choose this latter option. To finish the design, we choose the “Polygon” tool from the toolbox and click the list widget for our flattened points, which attaches a polygon over the points. To polish the code, we select and equalize the three depth sliders for each of our three calls to `makeKochPts`, obtaining a single depth variable. We're done!

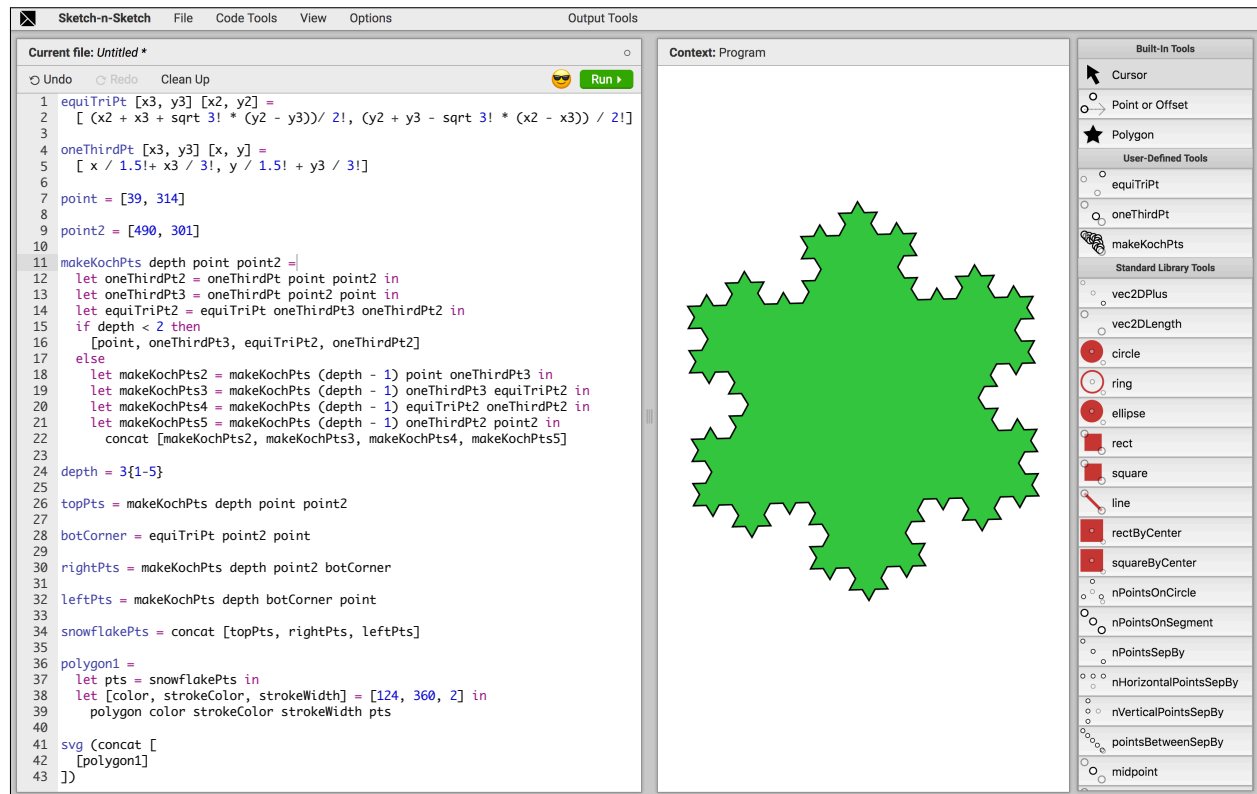


Figure 2.7: The completed Koch snowflake fractal, shown at recursion depth 3 with UI widgets hidden.

2.3.7 Tree Branch (feat. repetition over list and offsets)

Repetition is common in parametric designs. The prior example demonstrated repetition via recursion, but it may also be accomplished by special tools in SKETCH-N-SKETCH. Additionally, SKETCH-N-SKETCH supports the common graphical operation of offsetting one position from another by a fixed amount. To demonstrate these features, we construct the tree branch design shown in Figure 2.9. The construction involves a rhombus abstracted over its center point, which is then repeated over a list of points we draw in the program. (We omit renaming steps in the presentation below.)

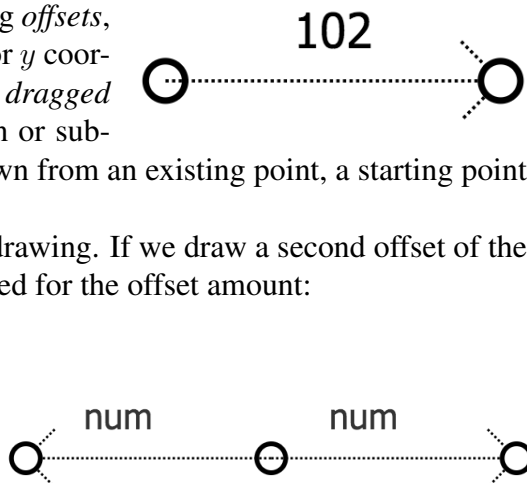
We construct the rhombus around a central point using *offsets*, which are simple additions to or subtractions from an x or y coordinate. The “Point or Offset” tool creates an offset when *dragged* on the canvas (rather than a click), inserting an addition or subtraction operation, e.g., $xOffset = x + 102$. If not drawn from an existing point, a starting point is inserted as well.

Offsets may snap their amounts to each other while drawing. If we draw a second offset of the same length in the opposite direction, a variable is inserted for the offset amount:

```
...
num = 102

xOffset = x + num

xOffset2 = x - num
...
```



We leverage this amount-snapping to quickly create the skeleton of the leaf rhombus (Figure 2.8a). We then draw a polygon, snapping to each offset endpoint, and ABSTRACT the resulting shape into a rhombus function parameterized over $[x,y]$, $halfW$, and $halfH$. We will attach instances of this function over the branch.

The branch is also constructed with offsets, so that it forms an axis-aligned isosceles triangle.

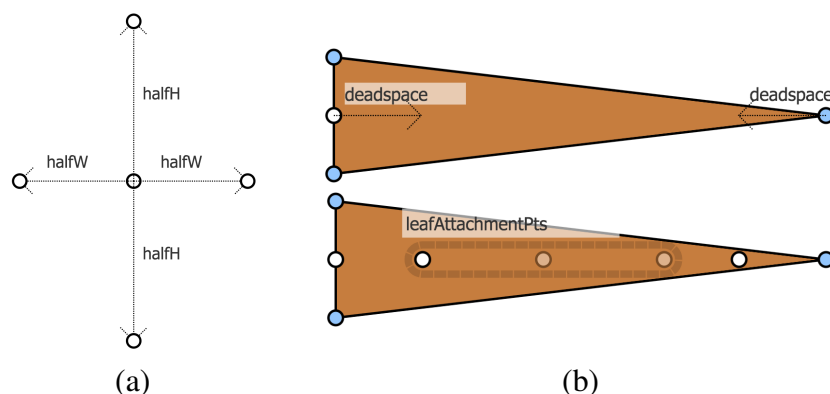


Figure 2.8: Tree branch example construction. (a) A skeleton of offsets. (b) Using deadspace offsets as the endpoints for the row of attachment points.

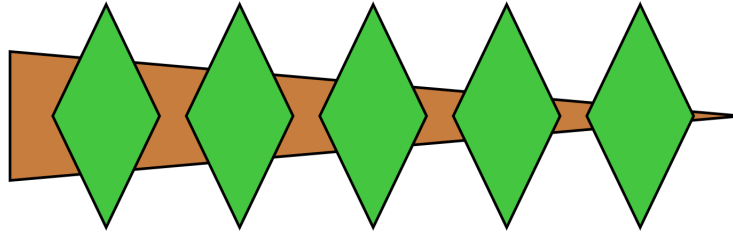


Figure 2.9: The completed tree branch design, utilizing *offset widgets* and *repetition* in its construction. Shown with widgets hidden.

We place two additional “deadspace” offsets inward from the ends of the branch to form the start and end of the attachment points for the leaves (Figure 2.8b).

We then create these attachment points by drawing the `pointsBetweenSepBy` function on the branch, one of several functions in the standard toolbox that returns a list of points. The `pointsBetweenSepBy` function returns points separated from their neighbors by a fixed distance. With this function, making our branch longer will add more leaves rather than spacing them out.

Finally, to repeat our leaf rhombus over the points, we first select the one copy of the rhombus on the canvas. The Output Tools panel then offers multiple tools for repeating the shape. We may **REPEAT WITH FUNCTION**, repeating the shape over a new call to any one of the point-list-producing functions available, or we can **REPEAT OVER LIST**, repeating the shape over an existing point list in our program. We **REPEAT OVER LIST** over the attachment points we just drew. The tool creates a new function abstracted over just a single point (`rhombusFunc2` below) and maps that function over our `leafAttachmentPts`, completing our leafy branch (Figure 2.9).

```
...
rhombusFunc2 ([x, y] as point) =
  let halfW = 40 in
  let halfH = 83 in
  rhombusFunc point halfW halfH
...
repeatedRhombusFunc2 =
  map rhombusFunc2 leafAttachmentPts
...
```

2.3.8 Target (feat. repetition by demonstration)

Repeating over a point list allows copies of shapes to vary only in their spatial positions. To support other repetition scenarios where the varying attributes could be calculated from an index (e.g., 0, 1, 2, ...), SKETCH-N-SKETCH offers a programming by demonstration workflow which we now briefly illustrate through the construction of a target.

We draw three concentric circles snapped to the same center point and change the color of the middle circle. We select the three circles and invoke **REPEAT BY INDEXED MERGE**, from which we select the second of two results—which differs from the first only in that it adds a reverse on

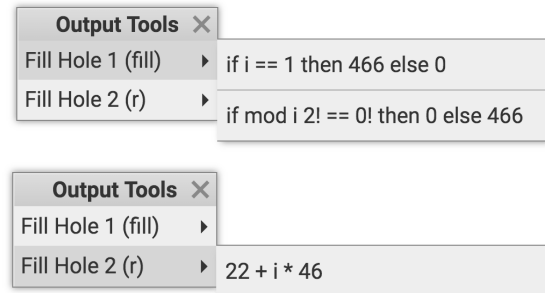
the last line below, so that $i=0$ for the last (topmost) small circle. The tool creates the following code in our program:

```
...
circles =
  map (\i ->
    circle \
      ??(1 => 0, 2 => 466, 3 => 0) \
      point \
      ??(1 => 114, 2 => 68, 3 => 25))
    (reverse (zeroTo 3{0-15}))
  ...
```



This code maps an anonymous function that takes an index ($\backslash i \rightarrow \dots$) over the list $[2, 1, 0]$. Each index is thus transformed into one of our circles. The anonymous function contains an expression formed by merging our original three circle definitions. The syntactic differences between the three original circle expressions—radius and color—have been turned into *programming-by-example (PBE) holes*, represented by $??(\dots)$. The first PBE hole above can be read as “the first time this expression is executed it should return 0, the second time it is executed it should return 466, and the third time 0.”

When a program contains PBE holes, SKETCH-N-SKETCH remembers the execution environments seen by each hole and employs sketch-based synthesis [137] to suggest possible fillings for each hole. Only the variable i ever differs in the execution environments at these holes, so all possible fillings are based upon i , as shown at right.



For the first hole, we choose the $\text{mod } i \ 2 == 0!$ conditional to obtain alternating colors. For the second we choose the only option, a $(base + i * width)$ expression, to calculate the radii.

Finally, note in the code listing on the previous page that the expression that generates the indices, $\text{zeroTo } 3\{0-15\}$, contains a range annotation which exposes a slider on the canvas [25]. The slider allows us to change the number of circles, similar to depth parameter in the Koch snowflake example. We choose to display five circles for the final design (Figure 2.10).



Figure 2.10: The final target design, constructed by filling PBE holes produced with REPEAT BY INDEXED MERGE. Shown with widgets hidden.

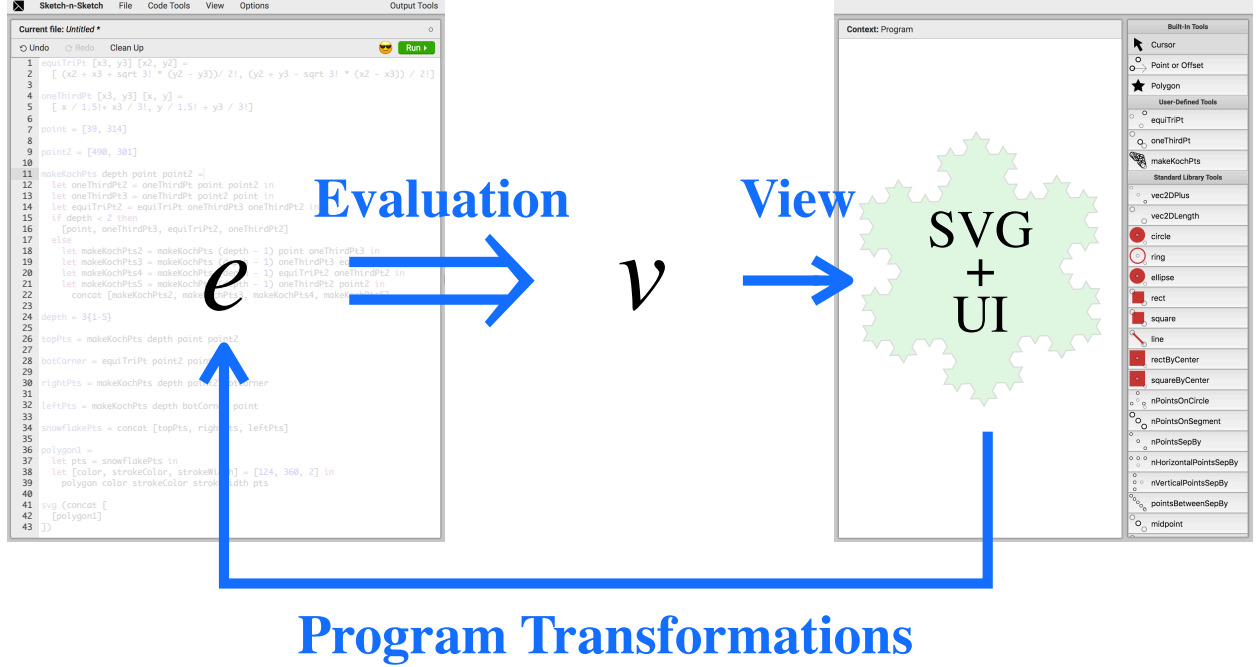


Figure 2.11: SKETCH-N-SKETCH workflow. The program e evaluates to a data structure v which is interpreted as an SVG and decorated with UI elements. User manipulations on the output are realized by transforming the program, resulting in new output to match.

2.4 Design and Implementation

SKETCH-N-SKETCH is a browser-based serverless web application written in Elm [40]. We used a modified standard library (to allow the placement of any type of value in sets) and also added mutation and exception handling to Elm. The SKETCH-N-SKETCH source code is available publicly³ and a live version of SKETCH-N-SKETCH can be tried online at <https://ravichugh.github.io/sketch-n-sketch/releases/uist-2019-acm-archive/>.⁴

At a high level, SKETCH-N-SKETCH operates as follows (Figure 2.11). To facilitate bimodal programming, the canonical representation of the user’s program is its textual code rather than an opaque, internal data structure. When the user presses “Run”, the SKETCH-N-SKETCH executes the code using its evaluator (recording provenance information in the process, discussed in §2.4.2). The final return value of the program is expected to be a data structure representing an SVG image. The SKETCH-N-SKETCH interface converts this output structure into an actual SVG and augments the shapes in the output with event handlers and graphical interface elements for selection and movement. When the user invokes a transformation (from the Output Tools floating menu shown in Figure 2.3 or by moving/resizing a shape), SKETCH-N-SKETCH updates the textual code, reruns

³https://github.com/ravichugh/sketch-n-sketch/tree/snaps_and_generalized_lambda (The version of SKETCH-N-SKETCH described in this thesis is on the branch named `snaps_and_generalized_lambda`).

⁴Chrome tends to work better than Firefox. If the online version of SKETCH-N-SKETCH is dead, the Supplementary Materials of [63] has a copy of the artifact—its README explains how to install and run the solver server.

the program, and displays the new output.

As detailed in the sections below, a large number of technical mechanisms work together to offer all the features in the SKETCH-N-SKETCH environment:

§2.4.1 Solver-based Value Updates explains how moving/resizing items with the mouse changes numeric literals in the code.

§2.4.2 Provenance details the tracing schemes SKETCH-N-SKETCH utilizes to associate selected items in the output with expressions in the program.

§2.4.3 Shape Selection & Feature Widgets recounts how SVG shapes are overlaid with UI elements that facilitate movement and selection of drawn elements.

§2.4.4 Intermediate Value Widgets describes the production and display of widgets to allow modification of certain intermediate values encountered during execution.

§2.4.5 Value Holes & Location Holes elucidates the mechanisms that snap-drawing and the MAKE EQUAL tool use to introduce or reuse variables in the program.

§2.4.6 Naming chronicles how SKETCH-N-SKETCH chooses quality names for expressions extracted into new variable bindings.

§2.4.7 Brands illuminates how types are tagged with extra information so that SKETCH-N-SKETCH can infer how to turn user code into drawing tools.

§2.4.8 Drawing expounds how drawing actions result in new function calls in the program.

§2.4.9 Focusing sets out how users may focus their editing on a single definition.

§2.4.10 Group & Abstract & Merge describes the grouping and abstraction tools.

§2.4.11 Repetition explicates the operation of the repetition tools in SKETCH-N-SKETCH, including the use of PBE holes for repetition by demonstration.

§2.4.12 Refactoring Tools catalogs SKETCH-N-SKETCH’s output-directed refactoring.

2.4.1 Solver-based Value Updates

When the user drags an item on the canvas—a shape, an edge, a corner, or a property’s slider—SKETCH-N-SKETCH updates a numeric literal in the program to affect the change. This interaction, called *live synchronization*, was the main feature of the initial version of SKETCH-N-SKETCH by Chugh et al. [25], upon which this dissertation builds. Below, we recount the operation of live synchronization in the initial SKETCH-N-SKETCH and our improvements to it.⁵

To determine which numeric literals to change during live synchronization, the evaluator in SKETCH-N-SKETCH records control flow-free traces on all numeric values [25]. For example,

⁵Live synchronization, sliders, and freeze annotations are the features this work inherits from Chugh et al. [25]. Outside this section, we do not attempt to distinguish between this work and the initial SKETCH-N-SKETCH.

Static Grammar

Expressions	e	$::=$	$n^\ell \mid op_m e_1 \dots e_m$ $\mid \lambda x.e \mid e_1 e_2$ $\mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$
Unitary Numeric Ops	op_1	$::=$	$\cos \mid \sin \mid \text{acos} \mid \text{asin} \mid \text{abs}$ $\mid \text{floor} \mid \text{ceil} \mid \text{round} \mid \text{sqrt} \mid \ln$
Binary Numeric Ops	op_2	$::=$	$+ \mid - \mid * \mid / \mid \text{pow} \mid \text{mod} \mid \text{atan2}$

Runtime Grammar

Numeric Traces	t	$::=$	$\ell \mid op_m t_1 \dots t_m$
Values	v	$::=$	$n^t \mid \langle E, \lambda x.e \rangle \mid \text{true} \mid \text{false}$
Environments	E	$::=$	$\cdot \mid x \mapsto v, E$

Figure 2.12: Grammar for an illustrative subset of SKETCH-N-SKETCH’s language.

consider the simple program `let x = 5 in x + 10`. Numeric literals are first tagged with unique locations ℓ_i : `let x = 5 $\ell_1$  in x + 10 $\ell_2$` . SKETCH-N-SKETCH evaluates the program to the tagged number $15^{\ell_1 + \ell_2}$ where the expression $\ell_1 + \ell_2$ is a *numeric trace* explaining the origin of the value 15. The trace is a form of “expression provenance” according to the nomenclature of [2]. The dynamic tracing process is straightforward; the details are exposted in Figure 2.13 for the simple language of Figure 2.12. Traces t only include numeric operations which, notably, means that control flow is ignored.

Traces enable mouse manipulations on the output to change a numeric literal in the program. When, on the canvas, the user manipulates that number 15 (via e.g., a slider) SKETCH-N-SKETCH chooses a location to change (using heuristics [25]), creates an equation wherein the location to change is the unknown, solves the equation for the unknown, then replaces the location in the program with the discovered value. For example, manipulating 15 to 30 induces the equation $30 = \ell_1 + \ell_2$. If SKETCH-N-SKETCH chooses ℓ_1 to be the unknown, other locations are replaced with their original values, producing $30 = \ell_1 + 10$. Solving the equation yields $\ell_1 = 20$ which is substituted into the program, resulting in the new code `let x = 20 in x + 10`. In this manner, direct manipulation of numeric properties on the canvas—such as shape locations, sizes, and colors—can be immediately realized by updating numeric literals in the program.

Because traces ignore control flow, while the user is dragging the mouse it is possible that some number will change causing the program to follow a different branch, changing the trace mid-drag. Consider, if the user manipulates the result of `let [x, y] = [0, 0] in if x < 50 then x else y` and, while dragging, the value of x becomes greater than 50, should SKETCH-N-SKETCH suddenly start changing y instead, before the drag is even complete? If so, the user can’t simply drag their mouse back to “undo” the change to x . Instead, SKETCH-N-SKETCH handles the scenario naively: the UI assumes traces do not change during the drag—if x becomes 50 or more, continued dragging will still change x but the live display will show the result as y so long as x is 50 or more. In

$$\begin{array}{c}
\frac{}{E \vdash n^\ell \Downarrow n^\ell} \quad \frac{\frac{}{E \vdash e_i \Downarrow n_i^{t_i}} \quad n = \llbracket op_m \ n_1 \ \dots \ n_m \rrbracket \quad t = op_m \ t_1 \ \dots \ t_m}{E \vdash op_m \ e_1 \ \dots \ e_m \Downarrow n^t} \\
\frac{}{E \vdash \lambda x. e \Downarrow \langle E, \lambda x. e \rangle} \quad \frac{E \vdash e_1 \Downarrow \langle E', \lambda x. e \rangle \quad E \vdash e_2 \Downarrow v_2 \quad x \mapsto v_2, E' \vdash e \Downarrow v}{E \vdash e_1 \ e_2 \Downarrow v} \\
\frac{}{E \vdash \text{true} \Downarrow \text{true}} \quad \frac{E \vdash e_1 \Downarrow \text{true} \quad E \vdash e_2 \Downarrow v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \\
\frac{}{E \vdash \text{false} \Downarrow \text{false}} \quad \frac{E \vdash e_1 \Downarrow \text{false} \quad E \vdash e_3 \Downarrow v}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}
\end{array}$$

Figure 2.13: Recording numeric traces.

practice, however, control flow changes during live synchronization was never a problem for the designs we implemented in SKETCH-N-SKETCH.⁶

We made two improvements to the live synchronization inherited from the initial SKETCH-N-SKETCH of Chugh et al. [25]. We swapped in a better solver and modified the heuristics SKETCH-N-SKETCH uses to choose which numeric literal will change upon a mouse manipulation.

We incorporated a more full-featured solver into SKETCH-N-SKETCH. The original solver in [25] was build for a set of simple examples and could only handle equations where the variable being solved for did not appear multiple times; this was sufficient for most, but not all, equations in practice. To handle these equations, we adopted the REDUCE [58] computer algebra system to serve as SKETCH-N-SKETCH’s solver. REDUCE—a software project over 50 years old!—can easily solve complex mathematical expressions. We also improved SKETCH-N-SKETCH’s heuristics for choosing which numeric literal to change in the program. Consider live synchronization of the simple expression $0 * 1$. SKETCH-N-SKETCH will deterministically choose whether to modify the literal 0 or the literal 1. Because this choice is made before attempting a solution, SKETCH-N-SKETCH would often choose to modify a literal that could never satisfy the needed equation—here, if SKETCH-N-SKETCH chooses to modify the literal 1, there is no way for the expression to evaluate to anything other than 0 and therefore no way for the result to match the user’s mouse movements. These poor choices did occur in practice, so to avoid modifying these futile locations we now first check that modifying the numeric literal can change the result of the expression. In particular, SKETCH-N-SKETCH takes the concrete derivative of the expression with respect to each candidate literal (using forward mode automatic differentiation for efficiency). If the derivative is 0, $\pm\infty$, or NaN, that literal will not be chosen for modification. This improvement significantly reduced occurrences where an item would not move when attempting to drag it with the mouse.

⁶A more pressing quirk to improve is to better handle 2D manipulations when both dimensions could modify the same literal, such as when dragging the point $[a+b, a-b]$. Despite the dependency between the x and y coordinate, the two dimensions are handled independent of each other, producing non-ideal changes.

2.4.2 Provenance

To affect program transformations, the UI maps selections to output values (an SVG-specific process, colored brown below). The provenance of these values is then interpreted as particular program expressions (a general-purpose process, in blue below) to which a transformation is applied.



The numeric traces described in the previous section only record the provenance of number values. We need to be able to track *any* value back to expressions of interest. Therefore, we record the origin of all values using a runtime tracing technique we call “Based On” provenance. To find larger values that contain the value of interest, we supplement “Based On” provenance with “Parents” provenance. Below, we introduce the operation of these techniques and discuss how they are used by SKETCH-N-SKETCH’s tools.

“Based On” Provenance

When the programmer selects a shape or a widget on the canvas and invokes an action such as DUPE or DELETE, SKETCH-N-SKETCH first needs to trace the selected value back to one or more program expressions. After relevant program expressions have been identified, the transformation will be applied to those expressions. Thus, the main question the system must answer is: “For a particular selected value, what expression(s) in the program does the programmer most likely intend to modify?”

As a rough, first-pass filter for discerning the programmer’s intention, we assume the programmer would like to edit the user-visible program itself but not the provided standard library of built-in code.⁷ To further narrow the user’s selection to just an expression or two in the program, we rely on trace information recorded during runtime and tagged onto values. Below we discuss the details of the tagging process and the algorithm for translating the tag on a selected value into expressions in the program.

In the evaluator, at every step of execution the value produced is tagged with two items: (a) the expression being executed, and (b) the values immediately used to evaluate the expression, *i.e.*, the values the result is based on. More formally, compared to a standard big-step semantics, we employ a tagging evaluation relation $E \vdash e \Downarrow v^\tau$, where τ is a tag of the form $\langle e, \{v_1^{\tau_1}, \dots, v_n^{\tau_n}\} \rangle$ which records the expression that produced the value as well as immediate values used in that production. We call this “Based On” provenance.

Figure 2.15 presents “Based On” tagging for a core language (Figure 2.14), with $\widehat{v} = v^\tau$. For brevity, the presentation splits evaluation into two mutually recursive relations: An augmented big-step semantics $E \vdash e \Downarrow v, \{\widehat{v}_1, \dots, \widehat{v}_n\}$ which produces a detagged value v as well as the set of values $\{\widehat{v}_1, \dots, \widehat{v}_n\}$ that v is immediately “Based On,” and the relation $E \vdash e \Downarrow \widehat{v}$ that

⁷Philosophically, the context for modification could be narrowed even further if the programmer has focused a particular definition. Our examples have not required this yet. When the programmer focuses their editing on a particular definition, canvas manipulations almost always affect changes within the focused scope simply because the canvas only offers the products of that scope for manipulation.

Expressions	e	$::=$	$c \mid x \mid e_1 \oplus e_2 \mid \lambda x. e \mid e_1 e_2$ $\mid$ $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ $\mid$ $(e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)$
Contants	c	$::=$	$n \mid b$
Detagged Values	v	$::=$	$c \mid \langle E, \lambda x. e \rangle \mid (\widehat{v}_1, \widehat{v}_2)$
Tagged Values	$\widehat{v}$	$::=$	v^τ
Tags	τ	$::=$	$\langle e, \{\widehat{v}_1, \dots, \widehat{v}_n\} \rangle$
Environments	E	$::=$	$- \mid E, x \mapsto \widehat{v}$

Figure 2.14: Core language for exposition of “Based On” provenance: a lambda calculus extended with numbers, booleans, binary operators $\oplus$, pairs, and if-then-else expressions. During evaluation, all values are tagged with their generating expression and the values the production is “Based On,” as detailed in Figure 2.15.

completes the tagging by gathering the immediate expression and the “Based On” values into a tag $\langle e, \{\widehat{v}_1, \dots, \widehat{v}_n\} \rangle$ that is placed onto the value.

The $E \vdash e \Downarrow v, \{\widehat{v}_1, \dots, \widehat{v}_n\}$ relation in Figure 2.15 highlights which immediate values a resultant value is “Based On.” In particular, control flow is ignored. For example, the result of an if-then-else expression is based on the value produced by the branch taken, but not on the conditional. Similarly, the result of a function application is based only on the return value of the function call—the application is not based on the function called, nor on its arguments. If needed, any arguments used to produce the result can be found by transitively following the “Based On” values of the return value; similarly, if the called function is needed, it can be discovered by inspecting the expressions of those “Based On” values to see what function they appear in.

In a related manner, plucking a value out of a container (via $\text{fst}()$ or $\text{snd}()$) does *not* record that the plucked value is “Based On” the container. Selecting, *e.g.*, the left edge of a rectangle should not be interpreted as selecting the whole rectangle itself. Conversely, selecting the whole rectangle could reasonably be interpreted as referring to all of the rectangle’s constituent values, thus constructing a container *does* record that the container is “Based On” its constituents (cf. the pair construction rule in Figure 2.15). This choice—that contained elements should not be not “Based On” the containers they are pulled out of—is the main difference between this provenance scheme and the dependency provenance scheme of Transparent ML [2] used for TINY STRUCTURE EDITORS in Chapter 3.

“Parents” provenance

Recall from Figure 2.15 that containers are “Based On” their constituent values, but not vice versa. The pair value $(10, 20)$ is based on the value 10 and the value 20, but neither 10 nor 20 is based on the pair. When a programmer selects a container they may be referring to all of its constituent values, but if they select a single constituent value it is unlikely they mean the entire container.

If, instead, the programmer selects *all* the constituent values of a container, then perhaps they

$$\begin{array}{c}
\frac{E \vdash e \Downarrow v, \{\widehat{v}_1, \dots, \widehat{v}_n\}}{E \vdash e \Downarrow v^{(e, \{\widehat{v}_1, \dots, \widehat{v}_n\})}} \\
\\
\frac{}{E \vdash c \Downarrow c, \{\}} \qquad \frac{E(x) = \widehat{v}}{E \vdash x \Downarrow v, \{\widehat{v}\}} \\
\\
\frac{E \vdash e_1 \Downarrow \widehat{v}_1 \quad E \vdash e_2 \Downarrow \widehat{v}_2 \quad v = v_1 \oplus v_2}{E \vdash e_1 \oplus e_2 \Downarrow v, \{\widehat{v}_1, \widehat{v}_2\}} \\
\\
\frac{}{E \vdash \lambda x. e_f \Downarrow \langle E, \lambda x. e_f \rangle, \{\}} \quad \frac{E \vdash e_1 \Downarrow \langle E', \lambda x. e_f \rangle^{\tau_1} \quad E \vdash e_2 \Downarrow \widehat{v}_2 \quad E', x \mapsto \widehat{v}_2 \vdash e_f \Downarrow \widehat{v}_3}{E \vdash e_1 e_2 \Downarrow v_3, \{\widehat{v}_3\}} \\
\\
\frac{E \vdash e_1 \Downarrow \text{true}^{\tau_1} \quad e_2 \Downarrow \widehat{v}_2}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2, \{\widehat{v}_2\}} \quad \frac{E \vdash e_1 \Downarrow \text{false}^{\tau_1} \quad e_3 \Downarrow \widehat{v}_3}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3, \{\widehat{v}_3\}} \\
\\
\frac{E \vdash e_1 \Downarrow \widehat{v}_1 \quad E \vdash e_2 \Downarrow \widehat{v}_2}{E \vdash (e_1, e_2) \Downarrow (\widehat{v}_1, \widehat{v}_2), \{\widehat{v}_1, \widehat{v}_2\}} \quad \frac{E \vdash e \Downarrow (\widehat{v}_1, \widehat{v}_2)^{\tau}}{E \vdash \text{fst}(e) \Downarrow v_1, \{\widehat{v}_1\}} \quad \frac{E \vdash e \Downarrow (\widehat{v}_1, \widehat{v}_2)^{\tau}}{E \vdash \text{snd}(e) \Downarrow v_2, \{\widehat{v}_2\}}
\end{array}$$

Figure 2.15: Evaluation rules showing the recording of “Based On” provenance, whereby each value is tagged with the expression that produced the value, as well as the values immediately used for that production. Each of those values is also tagged, and so on, except for constants or abstractions which are not “Based On” anything prior. When both v and $\widehat{v}$ appear in a rule, v is $\widehat{v}$ with its tag removed (*i.e.*, there is an implicit premise $\widehat{v} = v^{\tau}$).

are trying to refer to the container itself. In the vector graphics setting, this scenario most commonly occurs with points: if both the x and y coordinates of a point are selected, very likely the entire pair (x, y) should be considered selected. “Based On” provenance, unfortunately, does not allow the discovery of containers from constituents—given x and y , we cannot find where they are used as a pair value (x, y) .

To find containers from constituents, we tag all contained values with what we call “Parents” provenance. “Parents” provenance operates as follows: if a step of evaluation results in a value that contains other values, all contained values (and, recursively, their contained values) are mutably tagged as having been carried by this container value. Thus any value can inspect its “Parents” provenance tagging to see what other values it has been contained in.

As suggested above, the “Parents” provenance is used to affect changes to (x, y) pairs when both the x and y coordinates are selected—indeed SKETCH-N-SKETCH does not otherwise allow pair selection because only primitive coordinates are selectable in its UI. “Parents” provenance is also used when resolving snaps: if the value to snap to is not in the execution environment, but

there is a variable holding a container that contains the needed value, then the needed value may be made available by inserting a binding that pattern matches the needed value out of the container variable.

Although “Parents” provenance as described above works in practice, it will likely be replaced in a future implementation of SKETCH-N-SKETCH as it suffers from a theoretical flaw. Given a value, “Parents” provenance is able to answer the question, “What container values carried this value?” But more often, we instead want to answer, “What container values carried this value *to the canvas?*” The user wants to manipulate expressions in the thread of execution that resulted in the displayed canvas, not in irrelevant side branches of execution that just happened to use some of the same values. Unfortunately, the “Parents” provenance mechanism described above cannot distinguish between containers in the primary execution path and containers in other execution paths but holding the same values. In practice, we have not run into trouble with “Parents” provenance—these irrelevant paths seem to be rare in practice, at least in our examples—but future versions of SKETCH-N-SKETCH may record containment using a different mechanism. One possible solution may be to explicitly record pattern matches in the execution traces.

Expression IDs

Within the core language of Figure 2.14 it is not possible to distinguish between structurally identical expressions that occur in different code locations. Consequently, in practice the SKETCH-N-SKETCH parser tags all expressions with unique IDs. These expression IDs, rather than structural equality, are used to determine program locations.

As a further consideration, although it is generally impossible to preserve expression IDs between successive programmer text-edits to a program—the programmer could, *e.g.*, paste in an entirely different program—some attempt is however made to preserve expression IDs during program transformations in order to facilitate composite transformations. For example, the GROUP tool not only adds a new definition to the program and adds a new variable to the shape list, but also successively runs DELETE on each of the individual shapes to remove the prior individual shapes from the output. These sorts of compositions require expression ID references to remain valid across multiple transforms. AST transformations are structural edits rather than text edits, so expression ID preservation is usually automatic. But if new expressions inserted by the transform need to be referenced by later transforms in the composition, the transform must assign new expression IDs to the inserted expressions. In this scenario, for transform authors, SKETCH-N-SKETCH provides a convenient freshening function that walks the AST and reassigns expression IDs only to new or duplicated expressions—all expression IDs that occur exactly once in the AST are preserved.

Interpreting Provenance into Program Expressions

“Based On” provenance records an answer the question, “For a particular value, what *other values* at other execution steps were used to produce it?” Most of SKETCH-N-SKETCH’s tools need the answer to a different question: “For a particular selected value, what *expression(s)* in the program does it most likely refer to?” Below are five ways SKETCH-N-SKETCH’s tools discover relevant program expressions from the “Based On” provenance of a selected value $v^{(e, \{\widehat{v}_1, \dots, \widehat{v}_n\})}$.

Algorithm 1 Interpreting a provenance-tagged value $\widehat{v}$ into a “proximal” set of expressions where the expressions may be filtered by a predicate PRED.

```

function INTERP( $v^{(e, \{\widehat{v}_1, \dots, \widehat{v}_n\})}$ , PRED)
  if PRED( $e$ ) then  $\{e\}$ 
  else  $\bigcup_{i=1}^n$  INTERP( $\widehat{v}_i$ , PRED)
  end if
end function

```

Immediate expression The tool may use the expression e immediately tagged on to the value. In practice, this is often the expression chosen, albeit the expression is chosen by one of the mechanisms below because naively choosing the expression right on the value is not always appropriate—tools often need to limit which expressions in the provenance can be chosen. For example, we only want to select expressions in the user visible program, not the standard library. Or, a tool may wish to exclude variable usages. Consequently, provenance interpretation usually follows one of the strategies below, which allow the possible expressions to be filtered by a predicate.

“Proximal” interpretation matching a predicate To allow filtering of the possible expressions that “explain” where a value came from, Algorithm 1 may be employed. The INTERP function in Algorithm 1 starts with a value of interest $v^{(e, \{\widehat{v}_1, \dots, \widehat{v}_n\})}$ and walks the provenance back as minimally as possible until it finds a set of expressions that both (a) match the given predicate PRED and (b) account for all of the value of interest’s “Based On” values $\widehat{v}_1, \dots, \widehat{v}_n$. In practice, the predicate PRED always at least checks that the expression e is part of user code rather than library code. The walk-back returns as soon as possible because, intuitively, these recent execution steps are most “proximal” to the value of interest and thus are more likely to “explain” the value.

Single expressions Although a selected value may be interpreted as a *set* of expressions in Algorithm 1, many tools can only operate on a *single* expression. Consequently, SKETCH-N-SKETCH also includes a mechanism to return all possible single expression interpretations of a value. The algorithm (not shown) follows the same interpretation principles of Algorithm 1: a single expression is a valid interpretation of a value $v^{(e, \{\widehat{v}_1, \dots, \widehat{v}_n\})}$ if (a) that expression satisfies the given predicate and (b) that single expression accounts for all of the value of interest’s “Based On” values $\widehat{v}_1, \dots, \widehat{v}_n$, excepting those $\widehat{v}_i$ whose transitive provenance is located entirely in standard library code.

Several tools use this interpretation mechanism. For example, the REORDER IN LIST tool searches for an interpretation that is a single expression that occurs inside a list literal.

Unique expressions If the programmer selects a shape and hits DELETE, they are asking to delete the selected shape but, implicitly, all unselected shapes should remain intact. Since a program expression may be involved in the production of multiple canvas shapes, *e.g.*, a function call in a map expression, it is not appropriate to delete such an expression as such a deletion may

inadvertently remove unselected shapes. To avoid this scenario, the DELETE tool searches for interpretations that are *unique* to the shape selected, that is, interpretations whose expressions do not appear in the transitive “Based On” provenance of any of the unselected shapes. In practice, this is affected by invoking Algorithm 1 with a PRED filter that tests that the expression does not appear in the transitive “Based On” provenance of unrelated shapes on the canvas.

All expressions Instead of narrowing the interpretation to a single expression or two, certain tools instead look for *all* expressions reachable by transitively following the “Based On” values of the selected value. For example, the MERGE tool internally utilizes a clone detector which, even if invoked on an entire program, would only find a handful of possible code clones. Consequently, MERGE first finds all expressions reachable by the “Based On” provenance of the selected values and then instructs the clone detector to search for clones that intersect that reachable expression set. Similarly, the ADD ARGUMENT tool—available when some item on the canvas has been selected while a function is focused—searches for all expressions within the focused function that participated in the production of the selected value on the canvas. Each such expression is offered to become an argument to the function.

Providential Equality for Values

There is one final noteworthy use of provenance. When SKETCH-N-SKETCH needs to compare that values are not merely structurally equivalent but also came from the same execution path, SKETCH-N-SKETCH will compare the values’ “Based On” provenance in addition to their structure. Some execution steps, however, are “inert”: they will change a value’s provenance but otherwise pass the value through unchanged (*e.g.*, using a variable). What is most often desired is to compare values exempting such inert execution steps. For this comparison, SKETCH-N-SKETCH will walk the “Based On” provenance of the values backwards as far as possible through inert steps before comparing the values and their provenance at the most recent non-inert execution step.

2.4.3 Shape Selection & Feature Widgets

The ability to select shapes and parts of shapes is key to graphical editors. SKETCH-N-SKETCH is no different. Here we discuss how SKETCH-N-SKETCH handles selections, what items are selectable, and how those selections are interpreted.

Shapes in the output may be selected by single click. Multiple shapes can be selected by clicking with the Shift key depressed, or via lasso selection by dragging the mouse on the canvas when the “Cursor” tool is selected. Selecting shapes enables operation on the shapes. Internally, SKETCH-N-SKETCH represents the selected shapes by their shape number in the output (*e.g.*, internally, a selection of $[2, 5]$ means the second and fifth shape in the output SVG are selected). Intermediate value widgets (§2.4.4) are assigned negative shape numbers. Referencing selections by their shape number is not robust across program transformation when the number of shapes changes or shapes are reordered, but these scenarios did not occur often enough during our testing to warrant a more complicated representation for selections.

To select features (properties) of a shape, when a shape is selected or the user hovers their cursor over a shape, *shape feature widgets* are overlaid (Figure 2.16a). The feature sliders for fill color,

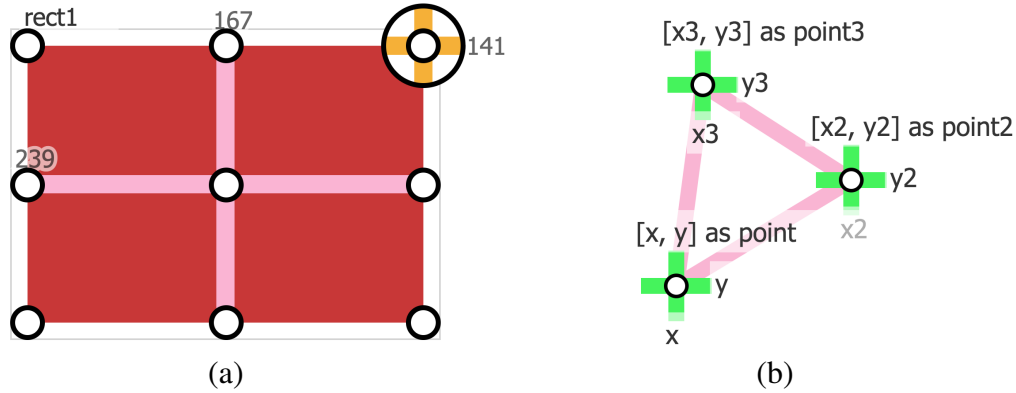


Figure 2.16: (a) Feature widgets for a rectangle: there are nine point features (dots) and two features for width and height (pink lines). Via a single click, the x, y coordinates for the top right corner have been revealed but not yet selected (selected features are green). The rectangle width and height, and the y coordinate of the revealed point are labeled with numbers. The x coordinate of the point is not labeled because it is not a primitive value in the program, but rather a computed value (as a design choice, the math is not shown). If the rectangle were selected, a slider for its fill color would appear as in Figure 2.4. (b) Pink distance features are drawn between all selected points to enable selection of distances.

and (when appropriate) stroke color and width, only appear once a shape is selected. Features may be selected by clicking so they can then be arguments to a transform (*e.g.*, MAKE EQUAL). Point features can also be dragged to move or resize the shape. When a feature corresponds to a variable or number in the program, the name or number is shown—variables may be renamed by clicking.

To reduce clutter, when the “Cursor” tool is selected, feature widgets are only shown for the shapes under the user’s cursor. When a drawing tool is selected, all shape point features are shown, since they may serve as anchors for snap-drawing a new shape—shape distance features (*e.g.*, width or height) are still only shown on mouse hover.

When multiple point features (or point widgets §2.4.4) are selected, *distance features* are drawn between all selected points to allow selection and operation on those distances (Figure 2.16b). Selecting the distance feature deselects the endpoints. For downstream transformations such as MAKE EQUAL, each selected distance is interpreted as $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where x_1 , x_2 , y_1 , and y_2 are the numeric traces of the endpoint coordinates.

The kinds of feature widgets shown differ per shape kind (*e.g.*, for lines, the two endpoints and midpoint are shown). These features are hard-coded in SKETCH-N-SKETCH for common SVG primitives. Selected features are referenced by the shape number plus the kind of feature selected. x and y coordinates may be selected individually; if both are selected SKETCH-N-SKETCH assumes the user has thereby selected the whole point and relevant transforms (*e.g.*, MAKE EQUAL or RELATE) will handle the x and y coordinates separately.

Rather than hard-coding the feature widgets per shape kind, a future possible improvement might to encode the shape features as ordinary functions in the standard library. The features for a shape are identified via type annotations, *e.g.*, `rectTopRight : Rect -> Point` or `rectWidth : Rect -> Width`. Extra annotation will be necessary, however, to notate where a feature like

$$\text{Feature Expressions } \mathcal{F} ::= \widehat{v} \mid op_m \mathcal{F}_1 \dots \mathcal{F}_m$$

Figure 2.17: Feature expressions grammar. op_m are operations from Figure 2.12.

`rectWidth` should be drawn and to notate what values should change when the user drags to resize (e.g., it is currently hard-coded that dragging the left edge of a rectangle should change both the rectangle’s x value and also the rectangle’s width so that the right edge stays fixed).

Many shape features represent computed quantities that do not immediately appear in the program, for example the right edge of a rectangle must be computed from $x + width$. Thus, for downstream usage, any selected features are first converted to *feature expressions*⁸ (Figure 2.17), which share the same grammar as numeric traces (Figure 2.12) except that terminals are provenance-tagged values $\widehat{v}$ instead of numeric locations ℓ . If the downstream transform, such as `MAKE EQUAL`, operates in numeric locations then the feature expression is converted to a numeric trace by replacing each $\widehat{v}$ with its numeric trace. If the downstream transform, such as `DELETE`, instead needs “Based On” provenance information, then the provenance tags in the feature expression’s $\widehat{v}$ s are consulted.

2.4.4 Intermediate Value Widgets

The goal of bimodal programming is to create a program by manipulating the execution products of the program. However, if only the final output of a program is shown, only the final output can be manipulated. Often, what is interesting is *how* that output was computed. To offer some opportunities for manipulating that *how*, `SKETCH-N-SKETCH` displays certain intermediate execution products on the canvas, summarized in Figure 2.18. The overview examples in Section 2.3 demonstrated these various intermediates: points, offsets, list widgets, and function call widgets.

Point widgets Whenever the evaluator encounters a number-number pair during execution of the program, a point widget is emitted. Point widgets are rendered as a movable and selectable dot on the canvas (Figure 2.18a), allowing the programmer to select or move that point. New, bare points can be added to the program with the “Point or Offset” drawing tool; a new binding is added to the top of the program such as `[x, y] as point = [100, 100]`.

Although drawn the same as the corner and edge dots that allow users to resize and select shape parts as discussed above in §2.4.3, point widgets are distinct. Point widgets refer to number-number pairs from execution, unlike shape features which refer to parts of shapes (e.g., the bottom right corner) they may not have occurred during execution. Point widgets are displayed all the time, whereas shape part widgets are only displayed when a shape is selected, when the mouse is over the shape, or when a drawing tool is selected.

In certain programs, multiple point widgets overlap exactly, which can be a source of consternation for the user. Lasso selection will select all the overlapping points, with no clear indication to the user that multiple points are selected; and transforms may operate unexpectedly when applied

⁸Source code note: these are erroneously named `FeatureEquation` in the code.

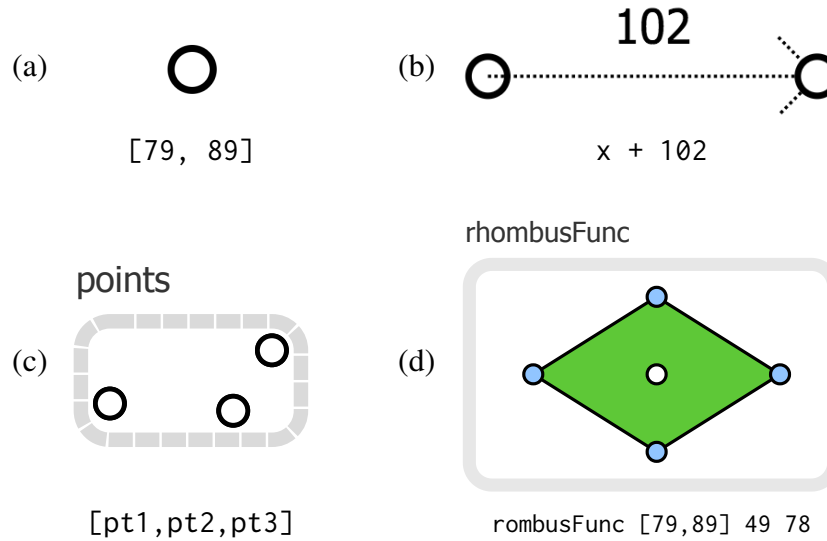


Figure 2.18: (a) Point, (b) offset, (c) list, and (d) function call widgets for intermediate execution products. An example of code that produces the widget is shown below each.

to all the points. A workaround is to avoid lasso selection and use click-selection which will only select the topmost point. In future work, a sidebar explaining selected items could help alleviate this problem.

Offsets In graphics code it is common to define offsets from some base x or y value. Therefore, during evaluation, when a numeric amount is added to or subtracted from an x or y coordinate, an offset arrow is drawn on the canvas (Figure 2.18b). The arrow may be selected or dragged; selecting/dragging references/changes the amount of the offset (102 in Figure 2.18b). The “Point or Offset” tool, when dragged on the canvas, adds a new binding to the program, *e.g.*, $xOffset = x + 102$, that will cause an offset to be drawn. (A new point binding is added as well, if the offset was not drawn from an existing point.)

Although the offset widget, when selected or dragged, refers only to the amount of the offset, the offset widget itself is defined by three numbers: the base x , the base y , as well as the offset amount. These widgets are emitted when the evaluator sees a number added to an x or y coordinate, *e.g.*, $x + 102$, but how does the evaluator know which side of the addition is the base and which side is the offset? And what is the y coordinate for drawing the offset? The expression $x + 102$ does not refer to any y coordinate at all.

To determine the base x and y coordinates for offsets, SKETCH-N-SKETCH uses another form of provenance tracking specifically for offsets. When a point (number-number pair) is introduced, each coordinate is tagged with an indicator of its axis (X or Y) *and* the other coordinate. Now when the evaluator encounters *e.g.*, $x + 102$, the x value holds the associated y coordinate and an offset widget can be produced; that, simultaneously. The evaluator determines that 102 is the amount and x is the base because the x value holds such a tag while 102 does not.

Figure 2.19 formalizes this process. Figure 2.19 builds off Figs 2.14 & 2.15, but note that, in the

Detagged Values	$v ::= \dots \mid n^\iota$
Coordinate Info	$\iota ::= - \mid \boxed{a, \widehat{v}}$
Axis	$a ::= X \mid Y$

$$\frac{E \vdash e_1 \Downarrow n_1^{\tau_1} \quad E \vdash e_2 \Downarrow n_2^{\tau_2} \quad \widehat{v}_1 = n_1^{\boxed{X, \widehat{v}_2}^{\tau_1}} \quad \widehat{v}_2 = n_2^{\boxed{Y, \widehat{v}_1}^{\tau_2}}}{E \vdash (e_1, e_2) \Downarrow (\widehat{v}_1, \widehat{v}_2)^{((e_1, e_2), \{\widehat{v}_1, \widehat{v}_2\})}}$$

$\begin{array}{l} E \vdash e_1 \Downarrow \widehat{v}_1 \quad \widehat{v}_1 = n_1^{\boxed{a, \widehat{v}_o}^{\tau_1}} \\ E \vdash e_2 \Downarrow \widehat{v}_2 \quad \widehat{v}_2 = n_2^{\tau_2} \\ n = n_1 + n_2 \quad \widehat{v} = n_1^{\boxed{a, \widehat{v}_o}^{(e_1 + e_2, \{\widehat{v}_1, \widehat{v}_2\})}} \\ emitOffset(a, +, \widehat{v}_1, \widehat{v}_2, \widehat{v}_o, \widehat{v}) \end{array}$	$\begin{array}{l} E \vdash e_1 \Downarrow \widehat{v}_1 \quad \widehat{v}_1 = n_1^{-\tau_1} \\ E \vdash e_2 \Downarrow \widehat{v}_2 \quad \widehat{v}_2 = n_2^{\boxed{a, \widehat{v}_o}^{\tau_2}} \\ n = n_1 + n_2 \quad \widehat{v} = n_1^{\boxed{a, \widehat{v}_o}^{(e_1 + e_2, \{\widehat{v}_1, \widehat{v}_2\})}} \\ emitOffset(a, +, \widehat{v}_2, \widehat{v}_1, \widehat{v}_o, \widehat{v}) \end{array}$
$E \vdash e_1 + e_2 \Downarrow \widehat{v}$	$E \vdash e_1 + e_2 \Downarrow \widehat{v}$

$$\frac{\begin{array}{l} E \vdash e_1 \Downarrow \widehat{v}_1 \quad \widehat{v}_1 = n_1^{\boxed{a, \widehat{v}_o}^{\tau_1}} \\ E \vdash e_2 \Downarrow \widehat{v}_2 \quad \widehat{v}_2 = n_2^{\tau_2} \\ n = n_1 - n_2 \quad \widehat{v} = n_1^{\boxed{a, \widehat{v}_o}^{(e_1 - e_2, \{\widehat{v}_1, \widehat{v}_2\})}} \\ emitOffset(a, -, \widehat{v}_1, \widehat{v}_2, \widehat{v}_o, \widehat{v}) \end{array}}{E \vdash e_1 - e_2 \Downarrow \widehat{v}}$$

Figure 2.19: Provenance for determining offset widgets, extending Figs 2.14 & 2.15.

actual SKETCH-N-SKETCH user syntax, pairs are written as two-element lists.⁹ When a number-number pair is introduced, each number is tagged with *coordinate info* ι which is either empty or a pair $\boxed{a, \widehat{v}}$ consisting of an axis a (either X or Y) along with the other coordinate's (provenance-tagged) value $\widehat{v}$. When an addition operation is encountered, if exactly one of the two operands carries non-empty coordinate info, an offset widget is produced (*emitOffset*) with the appropriate values for the axis, whether the operation was addition or subtraction, the base x and y values, the offset amount, and the ending amount. For subtraction, an offset is only produced if the first operand carries coordinate info.

As with shape features (§2.4.3), when the user has a drawing tool selected, all offsets are shown. To reduce clutter, however, when the user has the “Cursor” tool selected, offsets whose endpoint has been used (e.g., as the anchor for a shape) are hidden until the mouse hovers over either end of the offset. New, unused offsets are always shown.

⁹Conflating lists and tuples was a regrettable choice.

Because offsets must be based off of 2D positions, using many offsets can result in programs with more point definitions than perhaps necessary. A possible future simplification might be to move all offsets to rulers on the edge of the canvas so that offsets are 1D only. With this change, multiple shapes could more easily reuse the same coordinate, with fewer extraneous points in the program. With offsets representing bare x or y coordinates, they could then function as persistent “magnetic guidelines” [10] that objects may be attached or detached from.

List Widgets List widgets are dotted gray borders that allow the user to select, rename, and operate on lists (Figure 2.18c). The evaluator produces a list widget whenever a step of execution in the user’s program (*i.e.*, not library code) resolves to a list; although lists that do not contain graphical elements are not ultimately drawn on the canvas. The dotting in the border of the widget is intended to evoke the impression of list elements. The border appears when a user’s mouse hovers over a subvalue of the list, but is otherwise hidden to reduce clutter.

Traditional graphics editors allow shapes to be grouped together. In SKETCH-N-SKETCH shape groups are merely ordinary lists—a list of shapes is effectively a group. Selecting the list widget for the shapes allows the user to operate on the group, *e.g.*, to abstract the group into a function or to delete the group. Lists may also be *focused* (§2.4.9) so that drawing operations add to the particular list instead of to the final output. SKETCH-N-SKETCH, however, currently does not allow grouped shapes to be moved simultaneously—live synchronization (§2.4.1) only supports one moving object at a time.

Call Widgets To allow focusing and refactoring of functions, a call widget is produced when the program calls a user-defined function. That is, whenever a function application is evaluated, if the function application was in the user’s program *and* the called function was also defined in the user’s program (not the standard library), then a call widget is emitted. Call widgets are drawn with solid gray borders (Figure 2.18c).

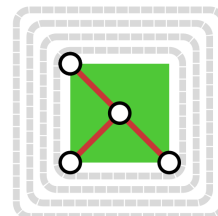
Like list widgets, call widgets allow the function to be renamed or focused for drawing operations. Additionally, when a function call is focused, the function arguments are shown with buttons next to each to remove or reorder the argument. To add an argument, selecting a shape or widget feature reveals an option in the Output Tools panel to add the feature as an argument to the function.

Positioning List & Call Widgets

An item is often a subvalue of many lists at once. In the logo example from the Overview (Section 2.3), four nested list widgets appear when the cursor moves over one of the shapes.

```
...
squareLineLine = [square1, line1, line2]

svg (concat [
  squareLineLine
])
```



When the user hovers over the list widget borders, the appropriate name is shown above the border and the relevant list is highlighted in the code box. From the innermost to outermost list widget, these lists are: (a) the initial group [square1, line1, line2], (b) the variable usage squareLineLine on the second to last line, (c) the unflattened shape list [squareLineLine], and (d) the final flattened shape list concat [squareLineLine].

Naive positioning of these widgets would draw them directly on top of each other, but SKETCH-N-SKETCH positions list and call widgets so that overlap is minimized. The algorithm proceeds as follows:¹⁰

1. List/call widgets are initially positioned to bound their subvalues. In practice, widgets often overlap with this positioning.
2. A directed acyclic graph (DAG) is constructed to determine an appropriate visual nesting order of the list/call widgets. For each pair of widgets A and B, widget A is marked as enclosing B if B is smaller and at least 75% of B's interior area overlaps A's interior area. If A and B have at least 75% overlap but are the same size, the tie is broken deterministically and only one will be chosen to enclose the other.
3. Following this containment DAG from innermost to outermost, list/call widget bounds are expanded to visually bound their enclosed widgets (their DAG descendants).

The above algorithm significantly reduces visual clutter and enables the user to more easily identify and select the desired list or function call.

2.4.5 Value Holes & Location Holes

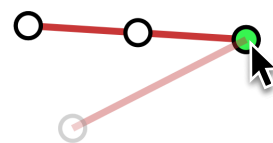
Several of SKETCH-N-SKETCH's tools encode the user's intention into a temporary intermediary program with special AST nodes representing equality constraints. These constraints are automatically resolved in a post-processing step and the special AST nodes—"value holes" and "location holes"—are never shown to the user. These holes are most directly utilized by the snap drawing, MAKE EQUAL, and RELATE interactions, so we describe the implementation of these interactions first before explaining how these special holes are resolved.

Snap drawing While drawing a new shape, the user may snap-draw [49] the start and/or end of the shape to an existing point on the canvas to immediately encode a constraint in the code linking the new shape to the existing point. For example, starting from a single line...

```
line1 = line 0 5 [80, 80] [200, 100]
```

...snap-drawing a second line to the first line's endpoint (as in the inset above) results in a shared point variable to link the lines' endpoints:

```
point = [200, 100]
line1 = line 0 5 [80, 80] point
line2 = line 0 5 [110, 130] point
```



¹⁰ShapeWidgets.computeAndRejiggerWidgetBounds in the source code.

This variable sharing is effected by the following algorithm. An intermediary program is first produced that contains *value holes* representing unresolved snaps. A value hole is a program expression that contains a (provenance-tagged) value from execution. Although the code is never shown to the user, the intermediary program in this case is...

```
line1 = line 0 5 [80, 80] [200, 100]
line2 = line 0 5 [110, 130] [?\{200\}, ?\{100\}]
```

...where $\{200\}$ and $\{100\}$ are value holes that contain copies of the number *values* originally produced on the first line; these values have their provenance attached (not shown). More precisely, the pair on the second line is $[\{\widehat{v}_x\}, \{\widehat{v}_y\}]$, where $\widehat{v}_x$ is the actual value produced previously upon execution of the expression 200 on the first line, as is $\widehat{v}_y$ for the expression 100 on the first line. The provenance tags on $\widehat{v}_x$ and $\widehat{v}_y$ indicate these are copies of those prior values from those prior expressions, and this information will be used later to determine how to resolve the holes and introduce a new variable. The value holes specify that, somehow, the expression at the value hole should evaluate to the same value as and (ideally) share the same provenance as the expression that produced the contained value.

The execution semantics for value holes is simple: a value hole evaluates to the contained value:

$$\overline{E \vdash \{\widehat{v}\} \Downarrow \widehat{v}}$$

This evaluation rule is indeed used: to provide a preview while the user is still drawing the shape, an intermediary program like the above is evaluated with value holes for any snapping.

Values holes can also represent snapping to computed values that are not in the original program, *e.g.*, to the midpoint of the line. To snap to the midpoint of the first line in the example above, SKETCH-N-SKETCH constructs the x and y expressions $(\{80\} + \{200\}) / 2$ and $(\{80\} + \{100\}) / 2$, evaluates them, and places the resulting values into value holes as before.

A final resolution process will resolve any value holes in the intermediary program, introducing new shared variables as appropriate. For values produced by expressions not in the program (*e.g.*, the addition and division to compute midpoints), the numeric trace of the number value (§2.4.1) will be reified into expression. The resolution process also resolves “location holes” produced by the MAKE EQUAL and RELATE tools, which we first discuss before explaining the resolution algorithm.

Make Equal The MAKE EQUAL tool enables users to select shape properties and instruct SKETCH-N-SKETCH to introduce shared variables into the program to ensure that the properties are linked and always the same. 1D properties such as length and width, 2D points, or abstract properties such as color may all be equalized. The mechanism for MAKE EQUAL differs from the snap drawing above. Partly this is an accident of the history of SKETCH-N-SKETCH development, but also because with MAKE EQUAL it is not *a priori* clear which items should be dependent on other (whereas with snap drawing there is a concept of an “old point” and a “new point”). Additionally, MAKE EQUAL allows all the properties being equalized to be computed values (*e.g.*, it is possible to equalize the midpoints of two lines to each other, whereas snaps only allow one of the properties to be computed). The MAKE EQUAL algorithm proceeds as follows.

If all selected features are points, the x coordinates are first equalized to each other, and then all y coordinates to each other; otherwise all selected features are mutually equalized. The equalization algorithm is based on the numeric traces (subsection 2.4.1). Say the user is equalizing three items: the left edges of two rectangles and the right edge of a third rectangle. Each item to be equalized is converted to its numeric trace. Recall from Figure 2.12 that numeric traces contain only mathematical operations $op_m t_1 \dots t_m$ and program locations ℓ (the locations of numeric literals in the code). The three traces in this case would be ℓ_{x_1} , ℓ_{x_2} , and $\ell_{x_3} + \ell_{w_3}$, for the two left edges and the right edge of the rectangles. These traces are gathered into a system of pair-wise equations, in this case the system $\{\ell_{x_1} = \ell_{x_2}, \ell_{x_2} = \ell_{x_3} + \ell_{w_3}\}$. A system of n equations may be solved for n variables, in this case two variables, but which locations should be solved for—which locations should be the *dependent* variables? For our example, there are many possible combinations of dependent locations: $\{\ell_{x_1}, \ell_{x_2}\}$, $\{\ell_{x_1}, \ell_{x_3}\}$, $\{\ell_{x_1}, \ell_{w_3}\}$, $\{\ell_{x_2}, \ell_{x_3}\}$, $\{\ell_{x_2}, \ell_{w_3}\}$, or $\{\ell_{x_3}, \ell_{w_3}\}$. SKETCH-N-SKETCH solves for all possible combinations (via the REDUCE computer algebra system [58]). The solutions here are:

Dependent	Solution	
$\{\ell_{x_1}, \ell_{x_2}\}$	$\ell_{x_1} = \ell_{x_3} + \ell_{w_3}$	$\ell_{x_2} = \ell_{x_3} + \ell_{w_3}$
$\{\ell_{x_1}, \ell_{x_3}\}$	$\ell_{x_1} = \ell_{x_2}$	$\ell_{x_3} = \ell_{x_2} - \ell_{w_3}$
$\{\ell_{x_2}, \ell_{x_3}\}$	$\ell_{x_2} = \ell_{x_1}$	$\ell_{x_3} = \ell_{x_1} - \ell_{w_3}$
$\{\ell_{x_1}, \ell_{w_3}\}$	$\ell_{x_1} = \ell_{x_2}$	$\ell_{w_3} = \ell_{x_2} - \ell_{x_3}$
$\{\ell_{x_2}, \ell_{w_3}\}$	$\ell_{x_2} = \ell_{x_1}$	$\ell_{w_3} = \ell_{x_1} - \ell_{x_3}$
$\{\ell_{x_3}, \ell_{w_3}\}$	<i>no solution</i>	

A candidate program is produced for each solution wherein each dependent location in the original code (a numeric literal) is replaced by its solution expression. The locations ℓ in that solution are temporarily reified as *location holes*—AST nodes that refer to the location of numeric literals elsewhere in the program. Like value holes, location holes are temporary and never shown to the user, but for discussion we will write location holes as $?_{\ell}$. Location holes consist only of that location identifier ℓ and are thus simpler than value holes which contain a value and its entire provenance. Location holes do not have an execution semantics.

For the first solution in the table above, the intermediary candidate program is produced as follows: the numeric literal for the left edge of rectangle 1 will be replaced by the expression $?_{\ell_{x_3}} + ?_{\ell_{w_3}}$ and the literal for the left edge of rectangle 2 will also be replaced by $?_{\ell_{x_3}} + ?_{\ell_{w_3}}$. The literals for the left edge and width of rectangle 3, at locations ℓ_{x_3} and ℓ_{w_3} , are left unchanged. The post-processing step will resolve these holes by lifting the numeric literals at ℓ_{x_3} and ℓ_{w_3} into new variables bindings at a scope visible to all corresponding location holes. The location holes will be replaced by uses of the new variables.

Relate A cousin of MAKE EQUAL, the RELATE transform attempts to guess an equation to relate one of the selected items in terms of the others; more specifically, one numeric literal among the selected items will be replaced with an expression in terms of the other numeric literals. As in MAKE EQUAL, RELATE operates on the numeric trace of each item. A location in the traces is a candidate to be the single dependent variable (the location ultimately replaced by an expression) if the location is unique to a single item’s trace (*i.e.*, that location does not appear in the other selected

item’s traces). For each candidate, an expression is guessed in terms of the locations unique to the other traces (appearing only in a single selected item’s trace).

For only 2 selected items, expressions up to size 3 are guessed; otherwise expressions are guessed up to size 7. The language for guessed expressions consists of addition, subtraction, multiplication, division, locations, or numeric constants. The possible locations are as described above (locations unique to each other selected item). The possible constants are the numeric constants used in SKETCH-N-SKETCH’s standard library (limited to those constants ≤ 10 , if 3 or more items are selected).

A guessed term is accepted if all of the following are met:

1. The dependent location evaluates to within 20% of its original value.
2. The distance between the dependent item and all other selected items does not change more than 20% (note that since only a single numeric location will be replaced, and the user’s selected items may have complex traces, that location may only be a subpart of the selected item’s trace; this condition here is a constraint on the ultimate computed value of that selected item rather than on just the replaced location).
3. The guessed expression uses a location from each other selected item (that is, the expression is indeed in terms of the other selected items).

As with MAKE EQUAL, x coordinates and y coordinates are related separately (the above process happens twice). The 20% constraint is 1D distance along each single coordinate. RELATE imposes an additional constraint between the x and y coordinates that the guessed arithmetic expression must be identical for both, modulo variable names.

For each candidate dependent location and for each valid guessed term, an intermediary program is produced with that location replaced by the guessed expression, with location holes for the locations. The location holes are resolved by post-processing, which we now discuss.

Resolving Value Holes & Location Holes

Value holes and location holes are an opportune representation of equality constraints within the AST. The holes are resolved to concrete expressions in multiple passes. Holes not replaced by earlier passes will be resolved in later passes. Additionally, to capture larger duplications, in the passes below SKETCH-N-SKETCH will first attempt to replace *ancestor* expressions of holes rather than the holes directly. For example, if the AST contains the expression $[?200, ?_{\ell_1} + ?_{\ell_2}]$, SKETCH-N-SKETCH will first look for a resolution that replaces that entire expression, then (if that fails), a replacement for $?_{\ell_1} + ?_{\ell_2}$, and finally will attempt to resolve any remaining holes individually. Thus the above expression may be resolved to a simple variable usage, *e.g.*, `point1`, instead of rebuilding a new point, *e.g.*, `[x, y + h]`. The resolution passes are as follows:

1. **Execution environment.** The execution environment will be examined to see if any variable in the environment can be used in place of the expression. If so, the expression is replaced by a variable usage. For value holes, a value in the environment is a match only if the “Based On” provenance also matches (modulo inert execution steps, as discussed in §2.4.2). Location holes match if the number in the environment came from the same location ℓ .

2. **Moving or introducing variables.** Variables are introduced or moved into scope so that holes may be filled by variable uses. To find the existing expression that should be referenced, for value holes the “Based On” provenance of the value is walked backwards until an expression is discovered that is both in the user’s program (not the standard library) and resolved to the same value. For a location hole $?_\ell$, the desired expression is simply the numeric literal in the program with the appropriate location ℓ .

If the desired expression is statically bound to a variable, that let-binding is moved upwards so it is in scope for the hole, otherwise a new let-binding is introduced containing the expression and the new variable is used in the expression’s original location and for the hole. All additional variables needed by the moved or introduced definition are recursively moved up as well (a cycle check prevents an infinite loop). Variables are renamed as necessary to avoid shadowing (via machinery from Hempel et al. [62]).

Handling larger expressions. As mentioned, SKETCH-N-SKETCH attempts first to replace ancestor expressions of a hole. When attempting to replace a larger expression (e.g., a pair with two value holes like $[?\{200\}, ?\{100\}]$) the “Parents” provenance of the values in the value holes is examined to find a possible program expression that matches the larger expression to replace. If there are no value holes (e.g., a pair with location holes $[?_{\ell_x}, ?_{\ell_y}]$), the whole program is searched for a syntactic match (where the numeric locations ℓ must match). As above, if an appropriate existing expression is found it is moved to a new let-binding or its existing let-bind is moved up so it is in scope, then the expression with holes is replaced by a variable usage.

3. **Destructuring.** Value holes may not all be resolved at this point if, e.g., the value in a hole was introduced inside a library function and does not therefore appear directly in the user’s program. For example, if the program contains `midpt = midpoint pt1 pt2`, a value hole might want the x coordinate of `midpt`, but that x value was introduced in the library code for `midpoint` and does not appear in the program directly. The “Parents” provenance of such values are examined and, if a parent value is available, a destructuring binding is introduced into the program to bind the needed value to a variable: the binding `[x, _] = midpt` in this case. That variable is then used in place of the value hole.
4. **Inlining numeric traces.** If all of the above fail to fill a value hole, and the value in the hole is a number, then the numeric trace (§2.4.1) of the number is converted to an AST expression, with the numeric locations in the trace converted to location holes. The value hole is replaced by this arithmetic expression, and the entire process above is repeated from step 1 to fill the location holes.

The above process has multiple steps because it evolved over time to handle scenarios that occurred as we implemented example programs. In many cases, the hole resolution will successfully use single variables such as `point1` instead of rebuilding `[x, y]` points, and will even avoid duplicating some math expressions already in the code. Even so, the hole resolution is not perfect. Notably, the execution environment is estimated. Additionally, attempting to resolve a value hole outside a function to a value produced inside a function will fail if that value cannot be destructured from the function’s return—the resolution process is not smart enough to, e.g., introduce extra return values from a function.

Final results are sorted by program size, with smallest programs first. Individual tools may provide an additional sort criteria to differentiate between identically-sized programs. For example, for MAKE EQUAL and RELATE, the program with shortest distance (in number of lines) between replaced numeric constants is preferred, and in case of a tie for that metric, replacing constants later in the program is preferred since it is more natural for later items to be based off earlier items. We found this set of heuristics is quite effective at ranking the most natural MAKE EQUAL result first.

2.4.6 Naming

SKETCH-N-SKETCH attempts to provide high quality default names for automatically inserted variables. While in some cases a new name is trivial to generate—drawing a new call to the `rect` standard library function will create a new `rect1` let-binding—choosing a good name when an existing program expression is extracted into a let-binding is not always straightforward. Graphics programs have many numeric expressions and calling them all `num1`, `num2` etc. is not helpful. Therefore, to determine an appropriate name for an existing expression before pulling it into a new let-binding, SKETCH-N-SKETCH examines the program as follows. In the following, “equivalent expression” means an ancestor or descendent expression that will always return the same value (e.g., a type ascription expression $e : T$ will return the same value as its child expression e , thus both are “equivalent” in the below).

1. **Variable usage.** If the expression is a variable usage, that variable name is chosen.
2. **Let-binding RHS.** If the expression (or an equivalent) is on the right-hand side of a let-binding and can be statically matched to a name on the left-hand side, then that name is used for the expression.
3. **Argument at a call site.** If the expression (or equivalent) is an argument of a function call and the called function can be statically determined, then the function definition is examined to find and use the corresponding function parameter name. This rule is key for providing reasonable names as the user introduces shared variables to encode constraints between drawn shapes.
4. **Coordinate in a numeric pair.** Following SKETCH-N-SKETCH’s idiom that all number-number pairs are points, if the two elements in a pair literal $[e_1, e_2]$ are both estimated to be numeric via static analysis¹¹ and no name better than `num` is discovered for the elements, then the names `x` and `y` are used for the elements.
5. **Type ascription.** If e in $e : T$ can only be named trivially by the final rule below, a name is instead chosen from the type T .
6. **Syntactic form.** If none of the above rules apply, a name is chosen based on the syntactic form of the expression, e.g., `num`, `bool`, `numList`, etc.

¹¹SKETCH-N-SKETCH’s type inference was added later in development but could instead be used here.

2.4.7 Brands

Because SKETCH-N-SKETCH designs are ordinary programs, custom functions that produce shapes can be incorporated as new drawing tools in the user interface. Any function that takes two points as input—or one point and a horizontal and/or vertical distance—is exposed as a custom drawing tool. To determine which numbers represent horizontal/vertical distances and to provide reasonable default values for *e.g.*, colors and stroke widths, a modified form of type inference is utilized. In particular, types are tagged with a set of zero or more inferred *brands* [101] that represent the role of particular values.

SKETCH-N-SKETCH uses ordinary Hindley-Milner type inference, adopting algorithm $\mathcal{J}$ [105], but modified as follows. The type at any program location may be tagged with a set of brands. Brands are simply names such as Ratio, Point, Distance, VerticalDistance, HalfWidth, Color, etc. New brands are introduced via type aliases, most of which appear at the top of the SKETCH-N-SKETCH standard library. The standard library functions use these type aliases to note the roles of function arguments:

```
type alias Ratio           = Num
type alias Point           = [Num, Num]
type alias Distance        = Num
type alias HorizontalDistance = Num
type alias VerticalDistance = Num
type alias Width           = HorizontalDistance
type alias Height          = VerticalDistance
...
type alias Rect = SVG

rect : Color -> Point -> Width -> Height -> Rect
rect fill [x, y] w h = ...
```

Using superscripts to represent brand set tags, the rect function above has the expanded type:

$$\text{Num}^{\{\text{Color}\}} \rightarrow [\text{Num}^{\{\}}, \text{Num}^{\{\}}]^{\{\text{Point}\}} \rightarrow \text{Num}^{\{\text{HorizontalDistance}, \text{Width}\}} \rightarrow \text{Num}^{\{\text{VerticalDistance}, \text{Height}\}} \rightarrow \text{SVG}^{\{\text{Rect}\}}$$

Brands are propagated during unification. Upon a successful unification of the ordinary types, the brand tags of the types being unified are unioned. Brands never hinder unification and play no type-checking role. In practice, brand information usually enters the user’s program via use of a standard library function and is propagated by unification. User-defined functions rarely need manual annotation to be correctly identified as drawable.

To provide brand information in more scenarios, SKETCH-N-SKETCH additionally adds brands based on certain syntactic code structures. The structural rules include our definition of point—a number-number pair is tagged with the Point brand, its left number with the X brand, its right with the Y brand—as well as special propagation rules helpful when using offsets—*e.g.*, if an untagged number is added to an X coordinate, the untagged number is assigned the HorizontalDistance brand.¹²

Although similar to dimension types for notating units of measures on numbers [79], brands in

¹²Source code note: see the end of AlgorithmJish.elm for the full list of rules.

SKETCH-N-SKETCH need not apply only to numbers, and, as implemented, brands play no part in checking program correctness, whereas dimension types will reject a program that, *e.g.*, attempts to add centimeters to inches.

The brands concept was originally introduced in the APX [101] bimodal programming environment for the same purposes as in SKETCH-N-SKETCH. Although, because of its object-oriented setting, APX implemented brands via named mixin traits rather than as side information.

2.4.8 Drawing

SKETCH-N-SKETCH includes several mechanism for adding new items into a program. The user may use a toolbox tool and draw with the mouse, or select and duplicate an existing item, or select an existing intermediate and add it to the output. Each of these mechanisms are discussed below.

Drawing shapes Functions in the standard library and program with an appropriate type signature are exposed in the toolbox as drawing tools. With the exception of the Cursor, Point or Offset, and Polygon tools, all tools shown in Figure 2.20 at right are such appropriately typed functions. A function is exposed as a drawing tool if either (a) two of its arguments are points, or (b) one of its arguments is a point and at least one other argument is some distance. Type inference with brands (§2.4.7) is used to determine these roles. The drawn shape position is derived from the user’s mouse movements. For non-positional arguments, the brand on the argument determines its default values (*e.g.*, red for the Color brand, 5 for StrokeWidth, etc.).

When the user draws a new shape, a new definition is inserted into the program and a usage of the new variable is inserted in a location such that the shape appears in the output. The process operates by guess-and-check: SKETCH-N-SKETCH attempts to add the new shape variable to the list literals in the program and succeeds when the size of the output increases by the expected amount. Because concat is often used to flatten shape lists, both shapeVar and [shapeVar] are candidates for insertion into lists. A static dependency analysis is used to avoid drawing into existing group lists (*i.e.*, the system prefers not to draw into lists that other lists depend on, because such a list is probably a shape group rather than the main shape list). We use a guess-and-check process because it is robust to transformations that the user might apply to the shape list. Consider a program that, *e.g.*, maps over the final shape list to add an opacity attribute to every shape. A back-propagation algorithm would need to know about list functions like map and concat in order to correctly rewind the output backwards to find the shape list in the program (as in Mayer et al. [100]). While possible, this approaches requires writing custom back-propagation rules for all list manipulation functions. Additionally, when map is involved, the resulting final shapes in the output are not structurally

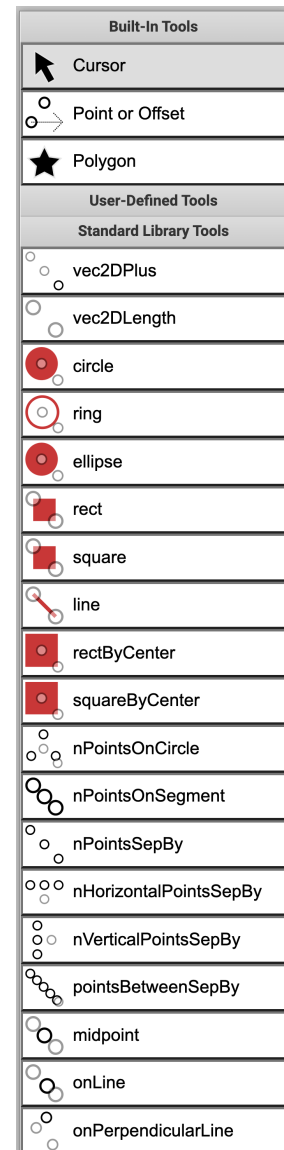


Figure 2.20: Toolbox

equal to the original shapes, meaning we cannot use a structural equality check to see if the desired new shape is now in the output. Thus overall, we adopt a guess-and-check process (to avoid implementing extensive back-propagation machinery) and count the change in the number of shapes in the output (to avoid a structural equality check).

Duplicate The DUPLICATE tool in the floating Output Tools menu (Figure 2.3a) examines the selected value’s provenance to find a single expression in the program, unrelated to unselected shapes, which is then duplicated into a new binding and (potentially) added to the shape list. The duplicated expression is never merely a variable usage.

Add to Output The ADD TO OUTPUT tool attempts to add the selected item(s) to the output of the focused function. ADD TO OUTPUT is a special case of shape drawing and reuses the same guess-and-check logic for determining where the new variable usage should be placed.

2.4.9 Focusing

The user may not always want to edit the final output of the program. They may want to modify only a single definition. As shown in the Koch example (§2.3.6), SKETCH-N-SKETCH offers the ability to *focus* on a specific expression. The remainder of the program’s output disappears, and drawing operations on the canvas add to the focused definition rather than to the program’s final shape list. And for the Koch example, focused editing enables recursive drawing by drawing a function inside itself (a workflow previously shown in Recursive Drawing [130] and Apparatus [131]).

SKETCH-N-SKETCH provides three ways to focus an expression or definition. Call widgets on the canvas may be clicked to focus on that particular function call. Or, if a selected item can be interpreted as coming from the right hand side of a let-binding, then a FOCUS DEFINITION tool is displayed in the Output Tools panel. Finally, an enterprising programmer may text-edit special comments into their code directly.

The focused function, and the example call which provide example arguments for its execution, are specified by special comments automatically inserted in the program. Comments, unlike internal expression identifiers, are preserved across arbitrary text edits to the program:

```
...
-- *** Focused Definition ***
makeKochPts point point2 =
  let oneThirdPt2 = oneThirdPt point point2 in
  let oneThirdPt3 = oneThirdPt point2 point in
  equiTriPt oneThirdPt3 oneThirdPt2

equiTriPt2 = -- *** Example Call ***
  makeKochPts point point2
...
```

The focused display is implemented by aborting execution early. Execution aborts when the function call marked by `*** Example Call ***` returns. If there is no example call but there is

a ***** Focused Definition ***** or a ***** Focused Expression *****, execution aborts after either of those instead. For tooling that requires examining the final execution environment (*e.g.*, determining which drawing tools to display), the execution environment from the focused item is used rather than the environment at an example call.

Shape drawing respects the focused context. The guess-and-check process for determining where to add new shape definitions only attempts to add shapes to lists within the focused context, and the “output” that the process examines for success is the output of the focused context.

2.4.10 Group & Abstract & Merge

Traditional graphics editors let users group items together. In a programmatic setting, a user may go further and turn a group into an abstract, reusable design (a function). Below we describe the implementation of SKETCH-N-SKETCH’s tools for grouping (*i.e.*, gathering items into a list) and then abstracting a new function. The abstraction step may build a new function based on a single copy of a design—via the ABSTRACT tool—or by diffing between multiple copies of a design—via the MERGE tool.

Group The GROUP tool gathers the selected items into a list, and is useful for grouping more than just shapes. A new list literal containing the selected items is placed in the program and bound to a variable. Internally, this process uses value holes (§2.4.5) to create the new list expression containing all the selected items. Once the new list definition is inserted an attempt is made to add that list to the output. Naively, adding the group to the output duplicates all the shapes on the canvas, as they were already also individually part of the output. To remove the old individual shapes from the output, for each variable in the new list other uses of that variable in the program are speculatively removed to find a removal that decreases the output size. Variable usages that do not affect the output size are retained. The process is repeated until as many individual shapes in the group have been removed from the output as possible.

Abstract The ABSTRACT tool attempts to turn *some* expression into a new function. The chosen expression is placed into the return position of the newly abstracted function, and the original expression is replaced by a call to the new function. As a heuristic, all bindings used only in the new function are recursively gathered into the function as local definitions, although bindings with no free variables are not gathered to ensure there are free variables left to abstract over. After gathering local definitions into the new function, free variables in the function body become arguments to the function, with the exception of free variables referring to functions (as determined by type inference). If the user is unhappy with the default parameterization, they may add and remove arguments using SKETCH-N-SKETCH’s refactoring tools (§2.4.12).

The Juno [111] and Juno-2 [64] constraint-oriented programming environments (§1.2.2) also feature a workflow for creating user-defined procedures in a manner similar to the ABSTRACT workflow above. In these systems, after constructing a design using a mix of text edits and mouse-based drawing actions, the programmer may invoke a command to turn the current design into a procedure parameterized by its input points. Like in SKETCH-N-SKETCH, the new procedure becomes available as a drawing tool in the interface.

Merge The MERGE tool performs a syntactic merge of the selected expressions, producing a shared function. Differences between the expressions become arguments to the function created, and the original selected expressions are each replaced by a call to the new function with appropriate arguments. In practice, MERGE operates by running clone detection over the entire program and offering those clones for elimination that contain at least one expression touched by the selected item’s entire “Based On” provenance. Potential clones must have twice as many AST nodes as the number of arguments that will be introduced when the clone is extracted into a function. Clones are replaced by calls to the derived function.

2.4.11 Repetition

SKETCH-N-SKETCH offers two workflows for producing repetitive designs. Users may either repeat a design over a list of points, or may manually lay out a repetitive design and ask SKETCH-N-SKETCH to infer the differences and math for the repetition—a programming by demonstration workflow. We describe both of these workflows below.

Repeating over a list of points A selected shape may be repeated either over an existing list of points in the program or over a new call to any function that returns a list of points (there are several such functions in the standard library, shown at the bottom of the toolbox in Figure 2.20). For example, suppose the programmer would like to repeat the logo design. With the `nPointsOnSegment` tool, they can draw on the canvas to produce a list of colinear points. If they select the logo design and choose to REPEAT OVER EXISTING LIST, the logo is attached to each of the colinear points via the following code:

```
logoFunc2 [x, y] =  
  let w = 100 in  
  logoFunc x y w 0 5  
  
nPointsOnSegment2 = nPointsOnSegment 3{0-10} [75, 326] [430, 379]  
  
repeatedLogoFunc2 =  
  concatMap logoFunc2 nPointsOnSegment2
```

The REPEAT tool creates a function (`logoFunc2`) that produces a copy of the logo given a single point, and maps that function (in `repeatedLogoFunc2`) over the list of points that were drawn earlier (`nPointsOnSegment2`). As with the GROUP tool (§2.4.10), the repeated group is added to the output and the original single shape is removed from the output (indeed, the original single shape definition is no longer available at the top level, it is subsumed into the repetition function—`logoFunc2` above).

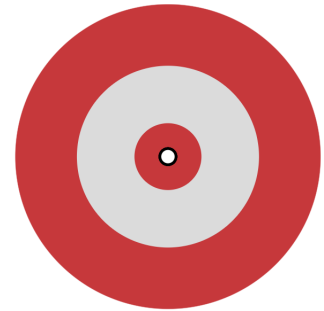
Instead of repeating over an existing list, the Output Tools also offer options to repeat over any of the standard library functions that return a list of points. A new call to the appropriate function will be added to the program, foregoing the need to draw such a call first as in the workflow above.

Repeating by “Indexed Merged” & PBE Holes The repetition workflow above only allows shapes to differ in their positions. To allow shapes to differ in other attributes, SKETCH-N-SKETCH

offers the programming demonstration workflow shown in the Target example (§2.3.8). The user lays out the first few copies of their design (perhaps via the DUPLICATE tool), roughly styles the shapes as desired, and then invokes the REPEAT BY INDEXED MERGE tool on the shapes, instructing SKETCH-N-SKETCH that the user believes the differences between the shapes could be derived from an index value i that takes on the values $0, 1, 2, \dots$ etc.

The REPEAT BY INDEXED MERGE tool syntactically merges the selected expressions, with the syntactic differences replaced by what we call *programming-by-example (PBE) holes*. A skeleton map $(\backslash i \rightarrow merged)$ $(zeroTo\ n\{0-3n\})$ is inserted into the program, where *merged* is the merged expression including PBE holes, n is a numeric literal of the number of items selected, and $3n$ is a literal three times larger. The tool also offers an optional extra result that reverse the indices. The $\{0-3n\}$ is a range annotation that allows the user to change iteration count via a slider on the canvas [25]. Recall from the target example that the generated code was as follows:

```
...
circles =
  map (\i ->
    circle \
      ??(1 => 0, 2 => 466, 3 => 0) \
      point \
        ??(1 => 114, 2 => 68, 3 => 25))
    (reverse (zeroTo 3{0-15}))
...
```



The merged expression contains PBE holes. Unlike value and location holes, PBE holes appear in the programmer-visible code. Each is written $??(1 \Rightarrow e_1, 2 \Rightarrow e_2, \dots, n \Rightarrow e_n)$, and contains a number of example expressions which represent what the hole is expected to evaluate to each successive time the hole expression is evaluated during a program run. The evaluator thus evaluates the hole accordingly, executing the next example expression on each successive encounter with the hole. If a PBE hole is encountered too many times during evaluation—*i.e.*, all its example expressions have already been used—the program crashes.

To enable filling the hole by program synthesis, the evaluator additionally logs the execution environments at the hole on each successive encounter. The PBE hole filling algorithm examines these environments to see what variables have changed and what their values are. This information is used to offer suggestions to fill (*i.e.*, replace) the PBE hole expression. The filling suggestions are offered persistently in the Output Tools panel so long as the program contains unfilled PBE holes (as shown in the inset). The hole fillings are currently generated based on a short list of built-in simple expression sketches [137] (*i.e.*, template expressions), each of which is either a simple mathematical skeleton (*e.g.*, $var + num$), or a simple if-then-else skeleton (*e.g.*, $if\ var == valFromVarDomain\ then\ exampleVal1\ else\ exampleVal2$). The hole filler re-

Output Tools ✕	
Fill Hole 1 (fill)	▶ if i == 1 then 466 else 0
Fill Hole 2 (r)	▶ if mod i 2! == 0! then 0 else 466

Output Tools ✕	
Fill Hole 1 (fill)	▶
Fill Hole 2 (r)	▶ 22 + i * 46

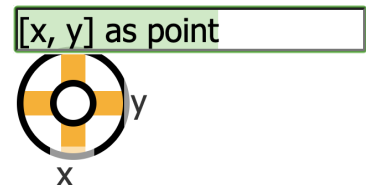
places the various terms in the sketch based on the examples in the PBE hole, perhaps dispatching the external mathematical solver [58], and offers the finished expression if all the numbers in the evaluated expression value are within 20% of their original values in each of the example execution environments. In order for this nearness requirement to be evaluated, PBE hole filling only operates if all examples return numbers or lists of numbers.

In this work, PBE holes are only generated by the REPEAT BY INDEXED MERGE tool. In future work, we imagine PBE holes might also be useful to facilitate an interaction that allows a repetitive design to be modified after-the-fact, to, *e.g.*, change the color of one shape in a series. PBE holes might also be useful in an interactive programming by example setting, as the holes themselves are very similar to structures used in the internal state of a traditional programming by example synthesizer such as Myth [118].

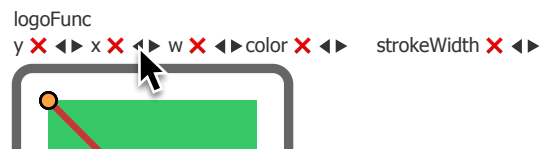
2.4.12 Refactoring Tools

Both programmers and computers rarely produce ideal code the first time. SKETCH-N-SKETCH offers several tools to help the programmer clean up their program. In keeping with SKETCH-N-SKETCH’s goal of enabling the user to perform as much programming as possible by interacting with the output, these tools are invoked by interacting with the canvas rather than with the code directly.

Rename Names aid human program comprehension [87]. In SKETCH-N-SKETCH, items on the canvas are labeled with an associated name or program expression. SKETCH-N-SKETCH attempts to provide high quality default names in its automatically-generated code (§2.4.6), but the user may still want to rename items. Names shown on the canvas may be clicked to immediately rename the item. Specifically, if the associated expression for an item is a variable usage or the left hand side of a binding, the pattern for the binding ($[x, y]$ as point) is displayed and clicking the text allows the user to rename all the variable(s) in the pattern (as shown in the inset).



Add/Remove/Reorder Arguments When a function call is focused, its arguments are shown with buttons next to each to REMOVE or REORDER the argument (as shown at right).



Additionally, while in a focused call, selecting a shape or widget feature reveals an option in the Output Tools panel to add the feature as an argument to the function. More specifically, every expression inside the function that appears in the transitive “Based On” provenance (§2.4.2) of the selected feature is offered as a candidate expression to be replaced by an argument. For example, if the programmer selects the right edge of a rectangle, ADD ARGUMENT will separately offer the rectangle x coordinate and the rectangle width each as a possible new function argument.

When the programmer adds/removes/reorders a function argument, all statically-determinable calls to the function are modified accordingly. When an argument is removed, the argument’s expression at the first function call is used as the concrete replacement inside the function and the

argument is removed from all call sites. When an argument is added, the replaced expression in the function is copied to all call sites. In both cases, possible scoping issues are ignored.

Reorder in List A selected item may be reordered within a list literal via a REORDER IN LIST tool in the Output Tool panel. Four options are offered: the selected item may be moved to the list head, to the list end, one space headwards, or one space endwards. These options were used in the Koch Curve example (subsection 2.3.6) to ensure points were returned in the correct from the recursive function. In practice, list reorderings are often just as easy to accomplish with text edits, although the REORDER IN LIST tool may be used to affect the “Move to front,” “Move to back,” etc. actions of a traditional graphics editor (albeit backwards—the front of a shape list is the shape drawn at the bottom).

Delete Pressing the Delete key interprets the selected values into expression(s) and attempts to remove them. If the last variable usage of a binding is removed, the binding is also removed.

As discussed in §2.4.2, which expression to delete is determined by transitively examining the “Based On” provenance of the selected items. Implicitly, items *not* selected should *not* be deleted, so their provenance is also examined to determine what expressions *not* to delete. When appropriate expressions are found that are unique to the selected item(s), DELETE successively attempts to remove those expressions from the program.

2.5 Evaluation (Case Studies)

To explore the expressive breadth of bimodal programming in SKETCH-N-SKETCH, we implemented 16 parametric designs, shown in Figure 2.21. These designs exercise different features: 6 designs are parameterized functions that appear as drawing tools at the end of construction, 7 involve repetition, and 1 uses recursion (the von Koch fractal). All 16 programs, spanning 427 lines of code total, were built entirely via output-based direct manipulations, without any text editing in the code box.¹³

To provide some external basis for assessing SKETCH-N-SKETCH’s expressivity, several of the examples are taken from the PBD test suite proposed in *Watch What I Do: Programming by Demonstration* [125]. The WWID: PBD suite spans diverse domains: 15 of its 32 tasks may be interpreted as parametric drawings. Of those 15, our work is able to complete 4 and partially complete another 2. These six are underlined in Figure 2.21.

Below we report on our authoring strategies for implementing the designs, including how often the various tools were used, and then discuss what SKETCH-N-SKETCH would need in order to complete the remaining WWID: PBD tasks without text edits.

¹³Videos of these examples are at <https://youtube.com/playlist?list=PLhhQAdToI5I4Qva5FRg0li3ZHLceaslY>

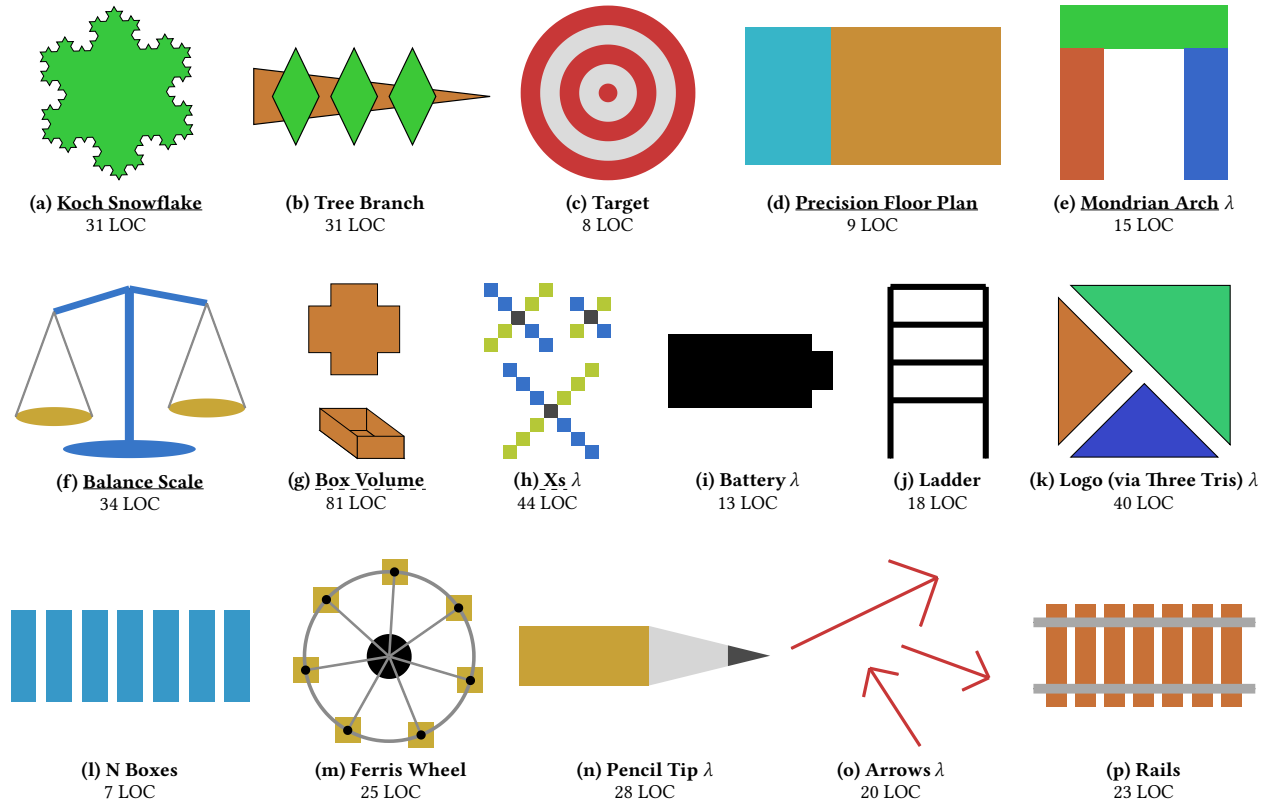


Figure 2.21: Examples created in SKETCH-N-SKETCH entirely through output-based interactions. LOC = Source lines of code. λ = Final design is a function that appears as a drawing tool. **Source of examples:** Tasks marked with underline (dashed = only partially completed) are from WWID: PBD [125]. (i) *Battery* is from Lillicon [13]. (j) *Ladder* is from QuickDraw [21]. (l) *N Boxes* and (m) *Ferris Wheel* are from the original, live-sync-only SKETCH-N-SKETCH [25].

2.5.1 Authoring

The lambda logo walkthrough in the overview framed the SKETCH-N-SKETCH workflow as a five stage *draw-relate-group-abstract-refactor* authoring process. In practice, design construction cannot always be cleanly delineated into precise stages in a fixed order. For example, while constructing the designs of Figure 2.21, we often specified the relationships *before* drawing the shapes: we laid out the desired parameterization of the design using points and offsets (as in, e.g., §2.3.7) and afterwards attached shapes using snap-drawing.

The #Ex column of Table 2.1 lists how many of the Figure 2.21 examples utilized each tool. Indeed, besides shape drawing and variable renaming, the most widely used functionality was SKETCH-N-SKETCH’s snap-drawing ability—not surprising given that the goal was to create parametric designs.

It is notable that to encode spatial constraints we more often preferred to snap-draw rather than use the MAKE EQUAL tool. MAKE EQUAL is more flexible, but not only does it require extra clicks compared to snap-drawing, MAKE EQUAL can offer a large number of different but hard-to-distinguish ways to enforce a constraint (Figure 2.3a is a tame example). To help, MAKE

Drawing	#Ex	Abstracting	#Ex
DRAW SHAPE	16	GROUP	8
DRAW CUSTOM SHAPE	4	ABSTRACT	9
DUPE	2	MERGE	0
DRAW POINT	3	REPEAT OVER FUNCTION CALL	2
DRAW OFFSET	9	REPEAT OVER EXISTING LIST	3
DELETE	4	REPEAT BY INDEXED MERGE	2
FOCUSED DRAWING	1	FILL HOLE	2
SNAP DRAWING (UI)	15	LIST WIDGETS (UI)	9

Relating	#Ex	Refactoring	#Ex
MAKE EQUAL	12	FOCUS EXPRESSION	3
RELATE	2	RENAME IN OUTPUT	15
DISTANCE FEATURES (UI)	6	ADD/REMOVE/REORDER ARG.	6
		REORDER LIST	4
		ADD TO OUTPUT	1
		SELECT TERMINATION COND.	1

Table 2.1: Program transformations and user interface features (UI) in SKETCH-N-SKETCH. The #Ex column indicates the number of examples in Figure 2.21 in which the feature was used.

EQUAL ranks results, preferring changes that rewrite terms near each other and later in the program (§2.4.5), which works well in practice. The least used tool for specifying relationships was the RELATE tool for guessing mathematical relationships—we only used it for constraints involving thirds.

Offsets plus snaps and MAKE EQUAL was sufficient, but not always convenient, for creating reasonable parameterizations of the designs. Laying out offsets beforehand requires forethought. A future SKETCH-N-SKETCH might benefit from tools to break constraints or invert dependencies after the fact.

As indicated in Table 2.1, we did not use the MERGE tool and three tools were used only once, all on the most challenging example, the Koch snowflake. The MERGE tool merges multiple copies of a shape into a function—we prefer ABSTRACT instead because it requires only a single example. The three tools only used for the Koch fractal all played a role in the workflow to specify recursion.

2.5.2 Remaining WWID Tasks

Of the 15 tasks in the *Watch What I Do: Programming by Demonstration* [125] benchmark suite that may be interpreted as parametric drawings, our system is able to fully complete four. What is needed to complete all the tasks without text edits?

Two of these remaining WWID: PBD tasks can be partially completed in this work (Figure 2.21g,h). To fully complete them, “Box Volume” would require an interaction to compute and display the numeric volume of the folded box, and “Xs” would require more precise control over what definitions are abstracted. (Not all uses of a `squareWidth` parameter are pulled into

the abstraction, causing the design to render incorrectly when drawing an X with different sized squares.)

The remaining nine tasks are diverse; no single feature would help with more than two or three. A prominent missing feature is arbitrary text boxes, with other elements placed relative to the text size. Beyond this, several examples require various list operations, albeit different such operations (*e.g.*, sorting, pair combinations, repetition over *pairs* of points, finding and replacing a middle element, finding a maximum). Beyond minor omissions (rotation handling and drawing paths with cutouts), SKETCH-N-SKETCH would also need to reason about intersections of lines with shape edges, to offer ways to specify overlapping and containment constraints, and to solve different kinds of such constraints simultaneously. Finally, one example would require creating *if-then-else* branches outside of a recursive or hole-filling setting and expose a boolean flag on the canvas to swap between the branches.

2.6 Discussion

SKETCH-N-SKETCH's tools are expressive enough to create all the designs in Figure 2.21 without resorting to ordinary text editing. Why bother having code all, and what improvements can still be made? We discuss these questions below.

2.6.1 Can you hide the code?

Although we constrained ourselves to only perform manipulations on the canvas, it is *not* our future goal to hide the code box and only display the canvas. We do not expect that output-based manipulations will be optimal for all program transformation tasks and, even supposing they were, the text-based representation of the program describes the complete operations that construct the design with a degree of comprehensiveness that cannot be economically represented on the canvas. For example, consider the expression `if bool then rect else circle`. The contents of both branches can immediately be read in the textual code. How can this be represented on the canvas? We might either show the output of only one branch at a time, or show both branches (*e.g.*, side-by-side) at the cost of considerable screen space. In contrast, the textual code for the expression occupies less than half of a single line of this paragraph.

For this reason, we do not envision a future SKETCH-N-SKETCH with hidden code. Instead, we hope to simultaneously leverage the strengths of textual code—its parsimonious representation of computation and its tractability for free-form editing—along with the strength of manipulable visualization—its concrete presentation and opportunities for tangible alteration.

2.6.2 Limitations & Future Work

Although the tools presented in this work may be used to create a number of designs without needing to resort to text-based programming, a number of improvements are possible.

Widget Clutter Because program execution may involve a large number of intermediate evaluation steps, even simple programs can clutter the canvas with widgets, making the interface unusable. Therefore, SKETCH-N-SKETCH hides most widgets by default and uses heuristics to determine when to show them—generally, upon the mouse hovering over some associated shape. Additionally, widgets from intermediate expressions in standard library code—outside the visible program—are generally not shown. Even with these techniques, there is still often quite a bit of visual noise on the canvas. Reducing that visual noise may prove to be a tradeoff with allowing functionality to be discoverable, and there may be no clear optimum.

Novices SKETCH-N-SKETCH assumes the user is comfortable working in code to understand program operation. Bimodal interactions might also help those with little programming experience—such as domain experts or students—to quickly produce rudimentary programs. Design concerns for novices should be investigated.

Unified Provenance The different types of provenance in this work were sufficient to build the functionality shown here, but SKETCH-N-SKETCH’s various provenance mechanisms were added in an ad hoc fashion over time and should be unified. Transparent ML by Acar et al. [2] describes a generic tracing scheme that preserves almost the entire operational derivation tree [22]. More focused kinds of provenance can be projected out of the trace [2], and *unevaluation* can be used to recover discarded derivation tree information [120]. Transparent ML is a good prospective foundation for uniting SKETCH-N-SKETCH’s various forms of provenance.

Practical Considerations The goal of SKETCH-N-SKETCH is to explore the feasibility of bimodal programming for creating vector graphics programs. The system currently lacks many features—such as rotation attributes and a path tool—that would be required in practical tool for creating parameterized drawings. In particular, while SKETCH-N-SKETCH offers tools for creating constraints (*e.g.*, MAKE EQUAL), it lacks tools for *breaking* those constraints.

Additionally, our example programs are starting to get longer than a page. Simply hovering over a widget to see its expression highlighted in the code may not be sufficient on longer programs: the expression may be offscreen. Scrolling the code to the relevant expression may be reasonable, as is done in, *e.g.*, TouchDevelop [19] or jsdare [123]. But in the case that a value is interpreted into *multiple* expressions that are not all within one page of each other, some alternative UI for handling the off-screen expressions may need to be developed.

Finally, on larger examples, running the code can be sluggish. To generalize this work to other domains, more attention will need to be paid to efficient computation.

2.7 Summary

We presented new techniques for bimodal programming in the SKETCH-N-SKETCH SVG programming environment. SKETCH-N-SKETCH enables the programmer create picture-drawing programs in a general-purpose (functional) programming language with few or no text edits. With the new techniques, we constructed a variety of non-trivial programs entirely through direct manipulation.

We implemented and demonstrated direct manipulation tools for *drawing*, *relating*, *grouping*, *abstracting*, and *repeating* shapes, as well as tools for *refactoring* the constructed program. To facilitate tool operation, we used techniques for tracking value *provenance* to associate output selections with relevant expressions in the program. In the UI, we exposed various *intermediate execution products* on the canvas for manipulation, so the programmer was not limited to manipulating their final output, and we offered *focused editing*, allowing the programmer to insert nested definitions and create recursive functions.

The expressive power of these bimodal programming tools was demonstrated by creating programs for 16 non-trivial designs using only the bimodal transformations. Overall, SKETCH-N-SKETCH shows you can have both direct manipulation *and* programmatic flexibility.

In the long term, we envision that the programming process might become as immediate and visual as direct-manipulation-based creativity applications—not just for shape-drawing programs but for non-visual programs as well. SKETCH-N-SKETCH is an early but significant step on that journey. In the following chapters, we take further steps towards this vision.

Chapter 3

Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions)

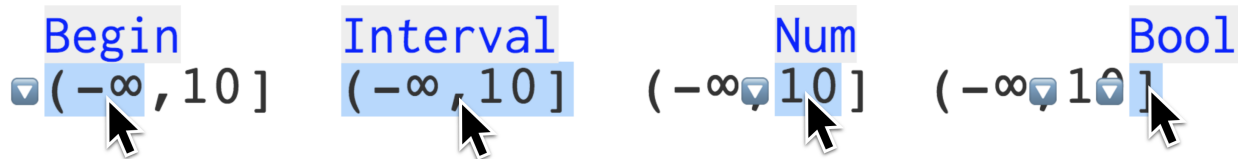


Figure 3.1: Selection areas in a structure editor automatically derived from a toString function.

3.1 Introduction

The vision of this dissertation is to apply bimodal programming not only to programs that output visual artifacts such as vector graphics, but also to discover how bimodal interactions can improve ordinary, general-purpose programming. Most ordinary programming, however, does not involve visual-spatial pictures. Instead, the data structures that a programmer works with are more abstract, and are custom to the program at hand. How should these custom data structures be visualized so they can be manipulated?

Imagine the programmer has defined a custom data type for ranges on the number line. They have a value representing the range with a lower bound of negative infinity and an upper bound of 10, inclusive. For the system to display this value, one option (Figure 3.2a) is to draw a pointer graph showing which data structures point to others in the running program (as in, *e.g.*, Python Tutor [54], FluidEdit [119], or Kanon [114]). The goal, however is not just to display the value but to also allow manipulation. The system might let the programmer grab and move the arrows representing pointers. Another option is to render a default textual representation for the value (Figure 3.2b). For manipulation, the system could overlay a UI on the string allowing the programmer to change different parts of the value’s internal structure.

Ideally, the system would offer visualizations that match the way the programmer thinks about the problem in their head. Neither a pointer graph nor the default textual representation are natural

This work was published as a short paper at VL/HCC 2020 (Hempel and Chugh [60]). Compared to [60], this chapter incorporates technical details that did not fit in the published version and were disseminated separately [61].

Excepting Section 3.4, portions of this intro and portions of Section 3.6, this chapter is ©2020 IEEE. Reprinted, with permission, from B. Hempel and R. Chugh, “Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions),” in IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020 [60].

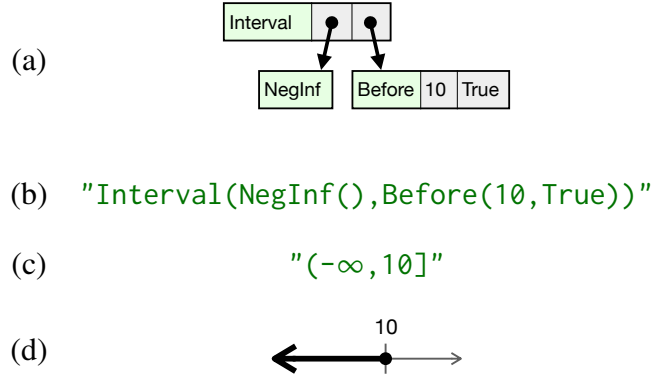


Figure 3.2: Ways to represent a custom data type. (a) Pointer graph, (b) default toString, (c) custom toString, and (d) custom graphical representation.

ways to represent contiguous intervals. Ideally the programmer could see *and manipulate* a natural representation for contiguous intervals, either the standard mathematical notation (Figure 3.2c) or a visualization on the number line (Figure 3.2d).

This ideal presents a challenge: the interval data type is a custom creation of the programmer. While it is fairly straightforward for the programmer write a toString (or a toSvg) function to produce a custom visualization, it is much more difficult to make a *manipulable* visualization. For manipulation, the programming environment might allow the programmer to code their own interactive UI. For common types, such as colors or regular expressions, this effort might be worth the trouble. Graphite [115] and mage [80] present the user with such customizable interfaces for editing subexpressions in their code, and livelits [117] shows how these customizable interfaces may be composed together by nesting (via type-compatibility). But, even for simple scenarios, interactive UI programming is not trivial, and creating a bespoke interface may not be worth the effort for less frequently used custom types such as the interval example here.

As noted, writing a toString function for a custom data type is usually straightforward—programmers often write toString functions as a matter of course. Could creating a *manipulable* visualization be as simple as writing a toString function?

Tiny Structure Editors (TSE)

In this work, we design a system, called TSE, that given a toString function for a custom data type, automatically generates *tiny structure editors* for manipulating values of that type.

To do so, TSE instruments the execution of the toString function applied to a value, and then overlays UI widgets on top of appropriate locations in the output string (Figure 3.5c). To determine these locations, TSE employs two key technical ideas: (a) a modified string concatenation operation that preserves information about substring locations and (b) runtime dependency tracing (based on Transparent ML [2]) to relate those substrings to parts of the input value.

While some prior systems [152, 133] trace string operations so that the programmer can edit strings in the output to thereby modify strings in the program, these systems can only modify literal strings in the code. TSE instead relates the output string to *any* original value of interest, not just

```

type Begin    = NegInf()          | After(Num, Bool)
type End      = Before(Num, Bool) | Inf()
type Interval = Interval(Begin, End)

valueOfInterest : Interval
valueOfInterest =
    Interval(NegInf(), Before(10, True))

```

Figure 3.3: ADT definitions for a custom interval type.

strings. Thus, TSE allows the user to modify not just strings but also numbers and custom data values.

In functional languages, custom data structures are represented using *algebraic data types* (ADTs), surveyed in the next section. Afterwards, we introduce TSE’s algorithm and discuss the editors produced by TSE for several common and custom data types.

3.2 Algebraic Data Types (ADTs)

Somewhat analogous to inheritance in object-oriented languages, *algebraic data types* (ADTs) enumerate the variants of a type and the data associated with each variant [121]. Unlike an object, an ADT value is raw data, separate from the functions that operate on it. Because ADTs succinctly describe the variants of plain data, ADTs are beginning to appear in mainstream languages: “enums” in Swift and Rust are ADTs, as are “case classes” in Scala and “discriminated unions” in Typescript.

Figure 3.3 shows three ADT definitions comprising a custom interval data type. The lower bound of an interval (Begin) has two variants representing whether the bound is negative infinity (NegInf()) or finite (After(...)). If finite, the bound records the finite boundary number and a boolean indicating whether the boundary is or is not included in the interval (is or is not *closed*). The type describing upper boundaries (End) is similar. An interval (Interval) is a lower and upper boundary together. The first word of each variant (NegInf, After, Before, Inf, Interval) is a *constructor* which acts as a function to create a *value* of the ADT. The last line of Figure 3.3 uses these constructors to create an interval value representing $(-\infty, 10]$.

Data inside ADT values is extracted using “pattern matching” in *case splits* (i.e., switch statements) which define the handling of alternative variants, as shown in the toString functions in Figure 3.4.

3.3 Algorithm

TSE’s automatic algorithm for generating tiny structure editors proceeds in three steps. The tracing evaluator relates substrings to portions of the original value, then 2D spatial regions over the rendered string are computed, and finally actions are assigned to the 2D regions.

```

toString : Begin -> String
toString(begin) = case begin of
  NegInf()          -> "(-∞"
  After(num, isClosed) ->
    (if isClosed then "[" else "(") ++ toString(num)

toString : End -> String
toString(end) = case end of
  Inf()          -> "∞)"
  Before(num, isClosed) ->
    toString(num) ++ (if isClosed then "]" else ")")

toString : Interval -> String
toString(interval) = case interval of
  Interval(begin, end) ->
    toString(begin) ++ "," ++ toString(end)

```

Figure 3.4: toString definitions for a custom interval type.

3.3.1 Dependency Tracing

TSE utilizes a custom evaluator that traces dependency provenance, following Transparent ML (TML) [2]. The value of interest and its subvalues are first tagged with *projection paths* (e.g., 2.2.●) indicating their location within the value of interest:

```
Interval(NegInf(){1.●}, Before(10{2.1.●}, True{2.2.●}){2.●}){●}
```

Based on the value's type, the appropriate toString function is invoked on the value of interest and the tracing evaluator propagates the dependency tags. Additionally, in TSE, string concatenation operations (++) do not produce a new, flattened string. Instead, the concatenation is deferred, resulting in a binary tree of substrings when evaluation completes (Figure 3.5a₁). Because of the tracing evaluator, each substring and each concatenation carries a set of projection paths, relating parts of the string to parts of the value of interest (Figure 3.5a₂).

3.3.2 Spatial Regions

In the final display, selection regions and UI widgets will be overlaid on top of the rendered string. To generate the selection regions, the string concatenation binary tree is translated into a binary tree of nested 2D polygons, with each polygon encompassing the spatial region of the associated substring (Figure 3.5b₁). Only regions associated with at least one path will ultimately be relevant (Figure 3.5b₂). Although a tree, the regions are nested tightly (Figure 3.5b₃), which can cause occlusion (discussed later). For a multiline string, the regions are shrunk to exclude whitespace, and each region may also exclude a portion of its first and last line (Figure 3.6).

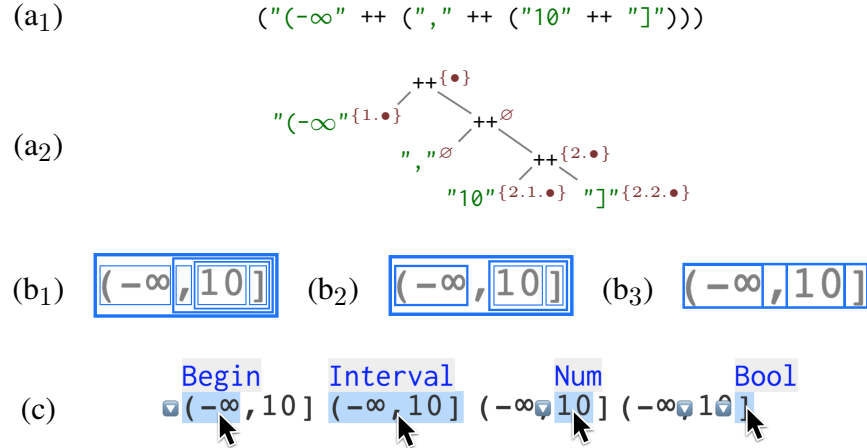


Figure 3.5: Steps in TSE’s generation of a structure editor. Instrumented execution with deferred concatenation produces a string (a₁) with each substring associated with parts of the value of interest (a₂). In the rendered UI, these regions are overlaid on top of the string (b) and the user may interact with these regions (c) to manipulate the original value of interest.

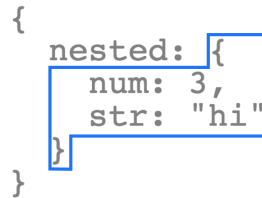


Figure 3.6: Multiline regions are contracted to exclude leading and trailing whitespace.

3.3.3 Selections and Actions

Once 2D regions of the displayed string are associated with corresponding locations in the value of interest, these 2D regions can be used to facilitate a number of interactions. The TSE prototype explores three: (a) *selection* of subvalues; (b) *base value editing* of numbers and strings; and (c) *structural transformations*, namely item insertion, item removal, and constructor swapping.

Selection regions When the user moves their cursor over the rendered string, the deepest (equivalently, smallest) region under their mouse is offered for selection/deselection. For the interval example, there are four possible selection regions, shown in Figure 3.5c. Selection is currently inert, but the selection regions are the basis for positioning UI widgets. In the future, selection might facilitate cut-copy-paste operations, as in Vital [56], or might open a floating menu of possible code transformations, as in SKETCH-N-SKETCH (Chapter 2).

Editing base values Literal numbers or strings from the value of interest may pass through to the output unchanged, for example the number 10 in the interval example. TSE lets the user manipulate these values. The user may click a number and drag up and down to scrub [147] the

```
type List<a> = Nil() | Cons(a, List<a>)
```

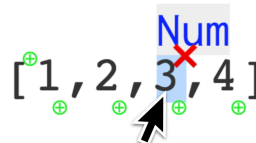


Figure 3.7: Traditional list ADT definition, along with a generated GUI for the list `Cons(1,Cons(2,Cons(3,Cons(4,Nil()))))`.

number to a different value. Both numbers and strings can be double-clicked to reveal a standard text box to text edit the value.

Structural transformations Because an ADT definition describes the allowable structures for a value, TSE is able to infer possible transformations on the value of interest. For the interval example, the `Begin`, `End`, and `Bool` types each have an alternative constructor which can be toggled by clicking the change constructor button (🔽) drawn to the left of the appropriate subvalue (Figure 3.5c). These buttons allow the user to, *e.g.*, change the lower bound from $-\infty$ to a finite bound (0 by default), or to toggle the boolean thus changing a finite boundary from closed ("`]`") to open ("`)`"). Which buttons to display are based on the selection region for the current mouse position—the deepest (smallest) region under the cursor. Since deepest regions may completely occlude some of their ancestors, TSE also displays the change constructor buttons for any such ancestor region that has no selectable area. For example, the `End` value "`10]`" is completely occluded by the `Num` "`10`" and the `Bool` "`]`", so when the cursor is over the `Num` or `Bool` TSE shows the change constructor button for `End` (the 🔽 over the comma in the right two cases in Figure 3.5c).

For recursive ADTs such as lists or trees, TSE additionally draws buttons to insert (⊕) or remove (✗) items from the data structure, as shown in Figure 3.7 for a list. Remove buttons are associated with item(s) to be removed. Insert buttons are trickier to position—TSE must predict where an item not currently in the data structure will appear. This prediction is occasionally imprecise, as discussed below.

Finally, in some cases, multiple buttons would be rendered in identical locations. Such overlapping buttons are coalesced into a single button that opens a menu offering the different transformations. The next section describes further implementation details; case studies of TSE follow in Section 3.5.

3.4 Implementation Details

As outlined above, TSE relies on dependency tracing and spatial region overlays to offer its interfaces. The technical mechanisms underlying these steps are described here in more detail. Our

Expressions e	$::=$	$x \mid f(x).e \mid e_1(e_2)$ $\mid C(e_1, \dots, e_n)$ $\mid \text{case } e \text{ of } C_i(x_1, \dots, x_n) \rightarrow e_i$ $\mid s \mid e_1 ++ e_2 \mid \text{strLen}(e)$ $\mid n \mid e_1 \oplus e_2 \mid \text{numToStr}(e)$ $\mid \text{basedOn}(e_d, e)$
Projection Paths π	$::=$	$\bullet \mid i.\pi$
(Tagged) Values w	$::=$	$v^{\{\pi_1, \dots, \pi_n\}}$
(Untagged) Pre-Values v	$::=$	$[E]f(x).e \mid C(w_1, \dots, w_n)$ $\mid s \mid w_1 ++ w_2 \mid n$ $\mid \text{dyncall}(f)$
Tagged Environments E	$::=$	$- \mid E, x \mapsto w$

Figure 3.8: Expressions, values, and, for dependency tracking, projection paths.

evaluator—combining tracing and dynamic dispatch—is described in §3.4.1. §3.4.2 and §3.4.3 describe cleanup steps to produce better interfaces, and subsection 3.4.4 details the heuristics for positioning insert and remove buttons.

3.4.1 A Dependency Provenance Algorithm

For tracing, TSE adapts the dependency provenance scheme of Transparent ML (TML) [2]. Provenance tracking in TML is ordinarily performed in two steps: first an execution trace is recorded during execution, then the desired provenance information is extracted from the trace. This two-step process enables TML to support multiple definitions of provenance. But because TSE only uses TML’s *dependency* provenance scheme, we can simplify this process: we collapse the two steps together and record the dependencies directly during execution, thus foregoing TML’s need to define a separate syntax of traces. The final provenance tags are still the same as in original TML (modulo the TSE-specific syntactic forms in Figure 3.8).

Figure 3.8 describes the syntax of TSE’s core language. The traditional constructs in the expression language are: variables x , recursive functions $f(x).e$ (where f is the function name), functional application $e_1(e_2)$, constructors with multiple arguments $C(e_1, \dots, e_n)$, case splits with multiple branches $\text{case } e \text{ of } C_i(x_1, \dots, x_n) \rightarrow e_i$, strings s , numbers n , and numeric binary operations $e_1 \oplus e_2$. Surface language if-then-else statements are desugared to case splits on True and False. To track provenance for substrings, string operations are primitives in TSE: string concatenation $e_1 ++ e_2$, string length inspection $\text{strLen}(e)$, and number to string conversion $\text{numToStr}(e)$ are part of the expression language. TSE also supports manual dependency addition via $\text{basedOn}(e_d, e)$, which allows the programmer to explicitly denote that the result of e should be considered dependent on that of e_d .

To track how output values are dependent on the input value of interest, Transparent ML (TML) assigns identifiers to each subvalue of the value of interest. These identifiers take the form of

Evaluation with Dependency Provenance

$$\boxed{E \vdash e \Downarrow w}$$

$$\begin{array}{c}
\text{[EVALVAR]} \quad \frac{}{E \vdash x \Downarrow E(x)} \quad \text{[EVALDYNVAR]} \quad \frac{x \in \{\text{"toString"}, \text{"showsPrecFlip"}\}}{E \vdash x \Downarrow \text{dyncall}(x) \{\}} \quad \text{[EVALFUN]} \quad \frac{}{E \vdash f(x).e \Downarrow ([E] f(x).e) \{\}} \\
\\
\text{[EVALAPP]} \quad \frac{E \vdash e_1 \Downarrow ([E_f] f(x).e)^{p_1} \quad E \vdash e_2 \Downarrow w_2 \quad E_f, f \mapsto ([E_f] f(x).e)^{p_1}, x \mapsto w_2 \vdash e_f \Downarrow v^p}{E \vdash e_1(e_2) \Downarrow v^{p_1 \cup p}} \\
\\
\text{[EVALDYNAPP]} \quad \frac{E \vdash e_1 \Downarrow \text{dyncall}(f)^{p_1} \quad E \vdash e_2 \Downarrow v_2^{p_2} \quad \text{typeDispatch}(f, v_2) = g \quad E, x \mapsto v_2^{p_2} \vdash g(x) \Downarrow v^p}{E \vdash e_1(e_2) \Downarrow v^{p_1 \cup p \cup p_2 \text{ if } f = \text{"toString"}}} \\
\\
\text{[EVALCTOR]} \quad \frac{E \vdash e_i \Downarrow w_i}{E \vdash C(e_1, \dots, e_n) \Downarrow C(w_1, \dots, w_n) \{\}} \\
\\
\text{[EVALCASE]} \quad \frac{E \vdash e \Downarrow C_j(w_1, \dots, w_n)^p \quad E, x_1 \mapsto w_1, \dots, x_n \mapsto w_n \vdash e_j \Downarrow v_j^{p_j}}{E \vdash \text{case } e \text{ of } C_i(x_1, \dots, x_n) \rightarrow e_i \Downarrow v_j^{p \cup p_j}} \\
\\
\text{[EVALSTR]} \quad \frac{}{E \vdash s \Downarrow s \{\}} \quad \text{[EVALCONCAT]} \quad \frac{E \vdash e_1 \Downarrow w_1 \quad E \vdash e_2 \Downarrow w_2}{E \vdash e_1 ++ e_2 \Downarrow (w_1 ++ w_2) \{\}} \quad \text{[EVALSTRLEN]} \quad \frac{E \vdash e \Downarrow w \quad \text{strLen}(w) = n \quad \text{allDepsDeep}(w) = p}{E \vdash \text{strLen}(e) \Downarrow n^p} \\
\\
\text{[EVALNUM]} \quad \frac{}{E \vdash n \Downarrow n \{\}} \quad \text{[EVALBINOP]} \quad \frac{E \vdash e_1 \Downarrow n_1^{p_1} \quad E \vdash e_2 \Downarrow n_2^{p_2} \quad n_1 \oplus n_2 = v}{E \vdash e_1 \oplus e_2 \Downarrow v^{p_1 \cup p_2}} \quad \text{[EVALNUMTOSTR]} \quad \frac{E \vdash e \Downarrow n^p \quad \text{numToStr}(n) = s}{E \vdash \text{numToStr}(e) \Downarrow s^p} \\
\\
\text{[EVALBASEDON]} \quad \frac{E \vdash e_d \Downarrow v_d^{p_d} \quad E \vdash e \Downarrow v^p}{E \vdash \text{basedOn}(e_d, e) \Downarrow v^{p_d \cup p}}
\end{array}$$

Figure 3.9: TSE’s adaptation of TML semantics.

projection paths π , which denote the tree-descent path from the root of the value of interest to the identified subvalue. Each (*tagged*) *value* w carries a set of these paths indicating the subvalues of the value of interest that w depends on.

Initially, the value of interest and its subvalues are tagged with singleton sets identifying their locations,¹ as in the example previously:

$$\text{Interval}(\text{NegInf}() \{1.\bullet\}, \text{Before}(10 \{2.1.\bullet\}, \text{True} \{2.2.\bullet\}) \{2.\bullet\}) \{\bullet\}$$

These projection paths are propagated during evaluation.

¹Figure 11 in Acar et al. [2] formalizes this initial tagging operation, although in their setting the operation is a little less trivial: their projection paths allow lookups into variables in the environment because they define program input to be a full execution environment of variable bindings which may include function closures with nested environments—for `toString` tracing in TSE we assume the input is a single value without closures and thus our projection paths do not need to support variable lookups.

Primitive values in TSE, called (*untagged*) *pre-values* v to distinguish them from their tagged forms w , include several traditional forms: recursive function closures $[E] f(x).e$ (where E is the captured environment of variable bindings and f is the function name), constructed ADT values $C(w_1, \dots, w_n)$, simple strings s , and numbers n . The TSE-specific forms are deferred string concatenation $w_1 ++ w_2$ (where w_1 and w_2 are each either deferred concatenations or simple strings), and a dynamic function call $\text{dyncall}(f)$ for late-bound type-based function dispatch to support multiple implementations of `toString` (discussed below).

Finally, for evaluation and closure capture, *tagged environments* E store a mapping from variable names to tagged values.

Figure 3.9 describes TSE’s adaptation of the tracing evaluation semantics of Transparent ML (TML) [2]. The tracing evaluation relation $E \vdash e \Downarrow w$ takes a tagged environment E and expression e for input and produces a tagged value w as output.

Variable names are simply looked up in the execution environment (EVALVAR). Values in the environment are already tagged with dependencies—these tags are retained unchanged. If the variable name is one of several special names that require type-based dynamic dispatch, instead of looking up the name in the environment, a $\text{dyncall}(x)$ value is produced representing the deferred function call (EVALDYNVAR). This value is assigned an empty set of dependencies. The variable name will be resolved to a function once the type of the argument is known (EVALDYNAPP).

Function definitions resolve to closures that capture the execution environment (EVALFUN). The closure value has no dependencies. Function application (EVALAPP) is standard. Functions are singly recursive²: after the argument expression is evaluated, both the argument value and the function closure are added as new bindings into the captured environment and the function body is executed. After a function result is produced, the dependencies p_1 of the closure are unioned with the dependencies p of the function result (although typically p_1 is the empty set). Notably, the dependencies of the argument value w_2 are not included in the union—if the argument was used in the computation of the result, these dependencies will already be represented in p .

Type-based dynamic function dispatch (EVALDYNAPP) is dynamically resolved to ordinary function application. Dynamic dispatch allows the same function variable name to be defined multiple times, with different type annotations on each definition. The appropriate closure is chosen when the function is called, based on the type of the argument. The implementation operates as follows: for those variable names considered dynamic³, a preprocessing step on the code (not shown) renames those (colliding) variable definition names to unique names. An internal dictionary (not shown) remembers the association between the type annotation on each definition and its unique name. At the call site, when the argument value v_2 is produced, its type is inspected and the dictionary for the non-unique function name f is consulted (*typeDispatch* in EVALDYNAPP) producing the unique name g of the implementation whose argument type matches. This function g is then applied normally to the argument. This scheme for dynamic dispatch means that multi-argument functions can only be dynamic in their first argument (although a fancy desugaring scheme might be able to work around this limitation without changing the core semantics presented here). The

²Before execution, mutual recursion is desugared to single recursion, following Exercise 9 of <https://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#Exo-9>

³Currently the dynamic names are hard-coded in our prototype. There are two dynamic names: `toString` and, for the GHC examples, `showsPrecFlip`.

dependencies for dynamic dispatch are propagated as in ordinary function application: the paths p_1 on the deferred function call `dyncall( $f$ )` are merged with those paths p from the function result. Finally, constant delimiters need to be associated the appropriate root value—*e.g.*, the opening “[” and closing “]” of a list should refer to the list. By the standard dependency rules, this would not ordinarily happen: constant delimiters are constant and therefore will not depend on the function argument. But, for calls to `toString` at least, it is reasonable to interpret the entire output as associated with the argument, so if the dynamic call was a `toString` function then the dependencies p_2 of the argument are added to those of the result value.

Constructor introduction (EVALCTOR) is standard—each argument expression is evaluated (the overline denotes multiplicity) and used for the arguments of the constructed value. The constructed value is assigned no dependencies.

Case splits (EVALCASE) are also standard. The scrutinee e is evaluated to a constructed value and the appropriate branch j is taken based on the scrutinee value’s constructor. The constructor’s arguments are bound to appropriate variable names for the branch and the branch expression e_j is evaluated. Finally, the dependencies p of the scrutinee value are unioned with the dependencies p_j of the branch result. This merger is key! Marking the case result as dependent on the scrutinee result is vital for TSE to work: this rule allows the UI to offer change constructor actions *e.g.*, clicking to toggle a boolean.

String literals are assigned no dependencies (EVALSTR). Deferred string concatenation (EVALCONCAT) is analogous to constructor introduction (EVALCTOR)—the string concatenation operator `++` is essentially an infix constructor with two arguments. The concatenation is assigned no dependencies but the dependencies on the left and right children are preserved. Inspecting the string length (EVALSTRLEN) is a built-in operation whose resulting number is marked as dependent on *all* the dependencies throughout the whole string concatenation tree (gathered by *allDepsDeep* in the rule).

Numeric operations are standard. Numeric literals (EVALNUM) are assigned no dependencies. Binary operations on numbers (EVALBINOP) produce a resulting value that is marked as dependent on both operands. Converting a number to a string (EVALNUMTOSTR) transfers dependencies from the number to the resulting string.

Finally, TSE supports manual dependency addition through the `basedOn( $e_d$ ,  $e$ )` built-in, which returns the result of its second argument e after adding the paths from the result of the first argument e_d (EVALBASEDON). Manual dependency addition is occasionally useful for the same reason that `toString` results are marked as dependent on the `toString` argument: constant delimiters, being constant, are not normally associated with the item being delimited. The dependency can be manually added.

3.4.2 Code Normalization

To avoid extraneous dependencies during tracing execution, prefix and suffix strings shared by all branches of a case split are pulled outside the case split.

Consider the following version of a list `toString` function:

```

toString(list) = "[" ++ elemsToString(list) ++ "]"

elemsToString(list) = case list of
  Nil()           -> ""
  Cons(head, tail) -> case tail of
    Nil()         -> toString(head)
    Cons(_,_)     -> toString(head) ++ ", " ++ elemsToString(tail)

```

In the final case split, the split on tail, both branches call toString(head). That head element's string will be marked as dependent on tail, and therefore, in the UI, moving the mouse over the head element will erroneously be interpreted as also referring to the tail—but the tail is the rest of the list *after* that element. To avoid this extraneous dependency, prefix and suffix strings shared by all branches of a case split are automatically moved out of the case split. The above code is translated to...

```

elemsToString(list) = case list of
  Nil()           -> ""
  Cons(head, tail) -> toString(head) ++ case tail of
    Nil()         -> ""
    Cons(_,_)     -> ", " ++ elemsToString(tail)

```

...which removes the extraneous dependency, so that the head is not associated with the tail. This normalization happens transparently before every execution and is not displayed to the user.

3.4.3 Concatenation Tree Normalization

After toString execution produces a tree of substring concatenations (Figure 3.5a₂), the projection path tags undergo normalization. Identical projection paths shared by adjacent substrings are recursively redistributed to their parent concatenation; afterwards nested occurrences of the same path are removed, retaining only outermost occurrences of a path. This normalization produces no change in the Interval example.

3.4.4 Positioning Remove and Insert Buttons

For recursive ADTs such as lists or trees, TSE draws buttons to insert or remove items from the data structure (Figure 3.7). Remove buttons (✗) are associated with the “contained” items that would disappear if a subvalue were replaced with one of its recursive children. For example, consider removing the item 2 from the list [1, 2, 3], which desugars to:

```
Cons(1, Cons(2, Cons(3, Nil())))
```

Removing the 2 means replacing the **bolded** subvalue above with its recursive child (underlined). Although the bolded **Cons(...)** is what is being replaced, that **Cons(...)** subvalue is itself a list and it would be inappropriate to imply that the whole sublist is what would be removed. Instead, the remove button is associated with that subvalue's non-recursive children, namely the **2**, that will disappear upon removal.

Insertion is roughly the reverse, and is similarly accomplished by looking for recursion in the ADT definition and using such locations as insertion points. Although generating candidate insertions is straightforward, positioning the insert buttons (⊕) is tricky because it relies on predicting where an item not currently in the data structure will appear. For this prediction, TSE uses the bottommost, rightmost point out of up to three candidate points: (1) the bottommost, rightmost point of the region(s) associated with the projection path immediately before the insertion location, (2) the topmost, leftmost point of the region(s) associated with the insert location projection path, and (3) the topmost, leftmost point of the region(s) associated with the projection path immediately after the insert location. Because a large, complicated data structure may include multiple kinds of containers of various types, these candidate points are subject to the additional constraint that they must be associated with the same container that is being inserted into. To enforce this constraint, container root paths are estimated by searching for projection paths whose parent value has a different type; all candidate points must be prefixed by the same container root. As shown in Figure 3.7, TSE’s positioning heuristic does not always place insert buttons in an aesthetically consistent location—the first insertion button is above the list while the others are below—but TSE’s heuristic needs to handle empty and multi-line data structures and we found the above heuristic, relative to other heuristics we tried, was least likely to place the buttons in confusing locations.

3.5 Case Studies

TSE’s goal is to provide low- to no-cost domain-specific value editors. We tested TSE on `toString` functions for a number of datatypes, measuring several properties of the generated editors as shown in Table 3.1. Table 3.1 reports the percentage of ADT subvalues that could be directly selected (*i.e.*, were not occluded, missing, or sharing a selection region with other subvalues). For data types representing containers (*e.g.*, lists or sets), Table 3.1 reports the percentage of contained items that can be selected. Compared to selecting any subvalue, the ability to select contained items is likely more important for downstream applications: the user might *e.g.*, select the items they want to extract from a container.

To evaluate TSE’s heuristic for insert button positioning, Table 3.1 also reports the percentage of insert transformations placed in reasonable locations. An insert transformation is considered *unreasonably* placed if either (a) the insert should be possible but is not assigned to any button in the UI, or (b) the insert shares a single button with other inserts, or (c) clicking the button inserts an item at a location other than the button’s position.

To provide evidence that TSE can operate on unmodified `toString` functions, we translated several `toString` functions from Haskell’s standard libraries to TSE’s Elm-like language, as shown in the bottom half of Table 3.1. These translations were performed as literally as possible.

Manual inspection of the case studies revealed a few issues to address in subsequent versions of TSE. Most notably, zero-width regions such as those from empty strings are ignored, which for some variants of list `toString` caused the final `Nil()` to be un-selectable. Additionally, selection region sharing and occlusion are sometimes troublesome. Two subvalues sharing the same selection region is a less of an issue—depending on the application, selecting a shared region could

Data Structure	Description	%Selectable		%Reasonable		Notes
		Subvalues	Items	Inserts		
Interval	"(-∞, 10]"	80% (4/5)				
Date	"May 9, 2020"	100% (4/4)				
JSON (multiline)	w/arrays, objects, strings, nums	33% (14/43)		81% (13/16)		Components represented separately. based on used 3x.
List	"[1, 2, 3]"	86% (6/7)	100% (3/3)	100% (4/4)		
List ("]" in base case)	"[1, 2, 3]"	100% (7/7)	100% (3/3)	100% (4/4)		
List (via join)	"[1, 2, 3]"	71% (5/7)	100% (3/3)	100% (4/4)		
List (via different join)	"[1, 2, 3]"	86% (6/7)	100% (3/3)	100% (4/4)		
Tree (S-exp)	"(2 (1) (4 (3) (5)))"	53% (10/19)	100% (5/5)	14% (2/14)		5 inserts missing; poor placements.
Tree (indented)	"2\n 1\n 4\n 3\n 5"	21% (4/19)	100% (5/5)	21% (3/14)		5 inserts missing; shared placements.
Pair [48]	"(10, "ten")"	100% (3/3)				
List [48]	"[1, 2, 3]"	100% (7/7)	100% (3/3)	100% (4/4)		
ADT (recursive) [48]	"Ctor4 (Ctor3 True "asdf")"	100% (4/4)		50% (1/2)		Bool region too long; same insert 2x.
Record [48]	"Record {field1 = ..., ...}"	100% (9/9)				Bool region too long.
Set [29]	"fromList [2, 3, 5, 7]"		100% (4/4)	0%		Not 1-to-1 w/ADT definition.

Table 3.1: Case studies of hand-written and translated toString functions.

offer to operate on any of the associated items. Occlusion, however, results in certain subvalues being un-selectable. One solution might be to expand ancestor regions by a few pixels. Finally, insert buttons were not placed in reasonable locations for tree-like data structures, but, as discussed next, how best to handle actions is a domain-specific consideration.

3.6 Limitations and Future Work

The TSE prototype demonstrates how simple interfaces can be derived from the toString functions that programmers already write. Before TSE can be incorporated into a larger system, however, several details still need to be worked out. In practice, TSE will also need to custom support actions beyond structural insert/remove, and will need to support more kinds of string operations. Additionally, how to handle nested pattern matches or how to apply TSE in an object-oriented setting are both open questions. All these considerations are discussed below.

Further Actions Although TSE is intended to support future bimodal programming systems, the TSE prototype here is a standalone demonstration system (TSE is not yet part of the MANI-POSYNTH system presented in Chapter 4). While TSE’s independent implementation highlights its key techniques, applying TSE to a particular application will require a number of further design decisions, particularly surrounding the handling of actions. For example, consider the set data structure in the last row of Table 3.1. The reference implementation [29] is based on a tree and maintains a number of invariants such as balancing, ordering, and non-duplication. None of these invariants are expressible in a standard ADT definition alone, and the internal tree structure is not exposed in the toString output ("fromList [2, 3, 5, 7]"). Therefore, only some of TSE’s selection regions are relevant—namely, the terminal items, as reported in Table 3.1—and the structural transformations generated by TSE are not meaningful because they do not enforce the set invariants. TSE does not yet provide an interface for specifying custom insert and remove functions, instead we imagine such an interface would be part of a larger, future IDE.

Tracing Substring Provenance As implemented here, the TSE prototype has another minor limitation. Systems that rely on string tracing [152, 133] provide custom implementations of string manipulation functions that correctly propagate dependencies. TSE currently only support string concatenation and string length—supplementing the language with additional string functions (*e.g.*, substring extraction) remains future work.

Dependence Ambiguity Under Nested Patterns All practical functional languages allow nested pattern matches, but our core language and TML do not. How dependency semantics should work for nested patterns is an open question: although a language’s compiler will unnest the patterns [9], different unnestings can result in different dependency traces. Consider the following nested pattern of tuples:

```
case boolPair of
  (True, True) -> a
  (_, _)       -> b
```

The pattern match compiler has two options for un-nesting the patterns. Either the first or the second element of the pair may be inspected before the other:

```
case boolPair of
  (fst, snd) -> case fst of
    False -> b
    True   -> case snd of
      False -> b
      True   -> a
```

```
case boolPair of
  (fst, snd) -> case snd of
    False -> b
    True   -> case fst of
      False -> b
      True   -> a
```

While both versions are semantically equivalent, they can result in different dependency structures. If, *e.g.*, `boolPair` is `(True, False)`, in the first version all case splits are executed and the result will be marked as dependent on all three scrutinees: `boolPair`, `fst` and `snd`. In the second version, the deepest case split is not executed and the result is thereby not marked as dependent on `fst`, only `boolPair` and `snd`.

This example suggests that relying on the pattern match compiler to determine the dependency structure may not be the right approach. An explicit dependency semantics for nested patterns might be required. Programmers think about case branches from top to bottom—if the first pattern does not match, try the second, and so on. Dependency semantics could match that intuition: a branch result might be marked as dependent on values corresponding to all nested patterns in the current *and prior* branches (of the same constructor).

While nested patterns are not uncommon in real-world code, such ambiguous cases did not occur in our examples.

TSE for Object-Oriented Code Finally, object-oriented (OO) programming is more common than the functional setting presented here—how might TSE be adapted to OO? Instead of ADTs, variants in an OO setting can be encoded as subclasses of a shared abstract superclass. With ADTs, differences between variants are handled via case splits; with objects, each subclass defines its own version of a particular named method and dynamic (*i.e.*, virtual) method dispatch chooses the appropriate implementation for a particular object. The key tracing rule in TSE, inherited from TML, dictates that the result of a case split is marked as dependent on the value split upon (EVALCASE in Figure 3.9). In the interval example, this rule is responsible for 3 of the 5 dependency tags in the result and this rule is what allows TSE to offer meaningful change constructor actions. In an OO setting, this rule would be equivalent to marking the result of a dynamic call as dependent on the receiving object. That is, if the result of a method call could vary between subclasses, then the result of the call should be marked as dependent on the receiver.⁴

Editors from toSvg Functions TSE instruments the execution of a toString function to produce an interactive editor for a custom data type. The same tracing scheme could instead be applied to a user-specified toSvg function, thus automatically turning a graphical representation, such as the interval on the number line in Figure 3.2d, into an interactive editor. A further intriguing direction is to also extend SKETCH-N-SKETCH (Chapter 2) so that it may be used to help construct these toSvg functions.

3.7 Summary

TSE generates structure editors based on the toString function for a data type, with little to no further programmer effort required. The automatically generated interfaces are sufficient for selecting items out of a container and, for list structures, the interfaces also facilitate item addition and removal.

Although presented here as a standalone prototype, TSE aims to be a mechanism used by a larger bimodal programming system. A user might use the interfaces generated by TSE to naturally specify changes on values and the bimodal system can reify those value changes into the code of the program. One such bimodal environment that, in the future, could profitably incorporate TSE is MANIPOSYNTH, presented next.

⁴Whether the result of a method call can vary between subclasses is only easy to determine in a language like C++ where such methods are explicitly marked `virtual`. In OO languages that more fully embrace late binding (*e.g.*, Smalltalk and Ruby), the evaluator would have to guess if the method is ever overridden anywhere at anytime, which is not generally possible in the fully dynamic setting.

Chapter 4

Maniposynth: Bimodal Tangible Functional Programming

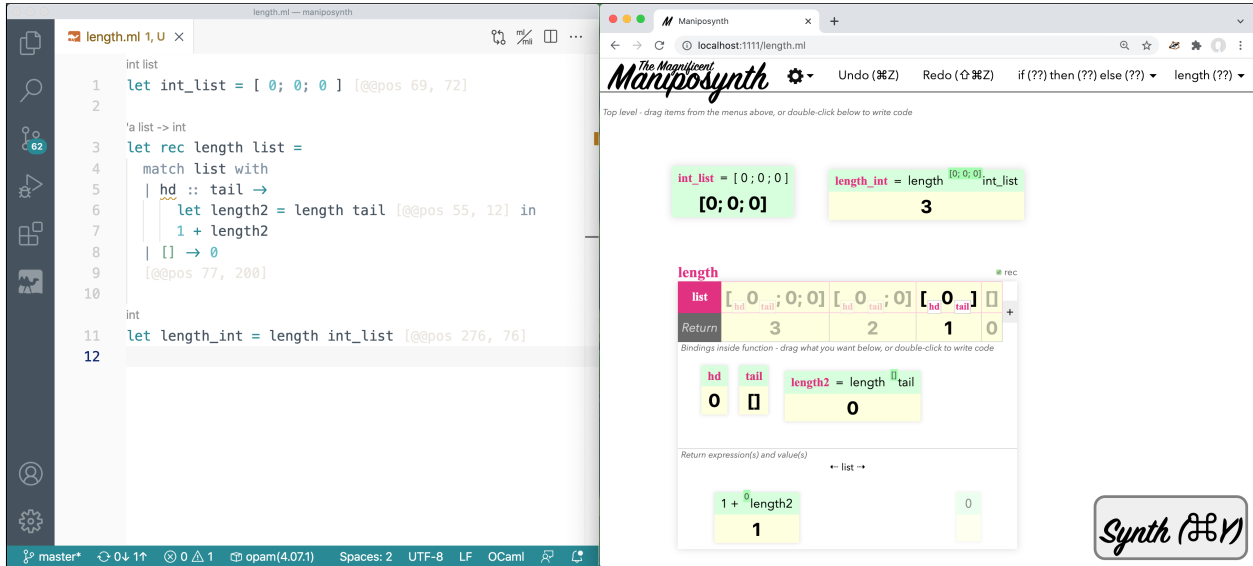


Figure 4.1: A list length function implemented in MANIPOSYNTH.

4.1 Introduction

SKETCH-N-SKETCH (Chapter 2) demonstrated that bimodal programming can be used to produce non-trivial vector graphics programs without the need to resort to text-based editing. But SKETCH-N-SKETCH can provide direct manipulation for vector graphics programs because there is something intuitive to manipulate: the graphical program's *graphical* output. What about *non-graphical* programs? Could similar interactions be applied to non-visual, general-purpose programming? Is there a way to write programs via direct manipulation on live program values?

The Eros environment by Elliott [38] demonstrated a compelling partial answer to this question. Eros reimaged the programming space not as a program in text (as in traditional coding), nor as a draftsman's drawing of operations connected by wires (as in nodes-and-wires programming [142]), but as a 2D canvas of manipulable values. These *tangible values* (TVs) were primarily partially applied functions, rendered with (graphically editable!) example arguments for their unapplied

The work in this chapter is in submission.

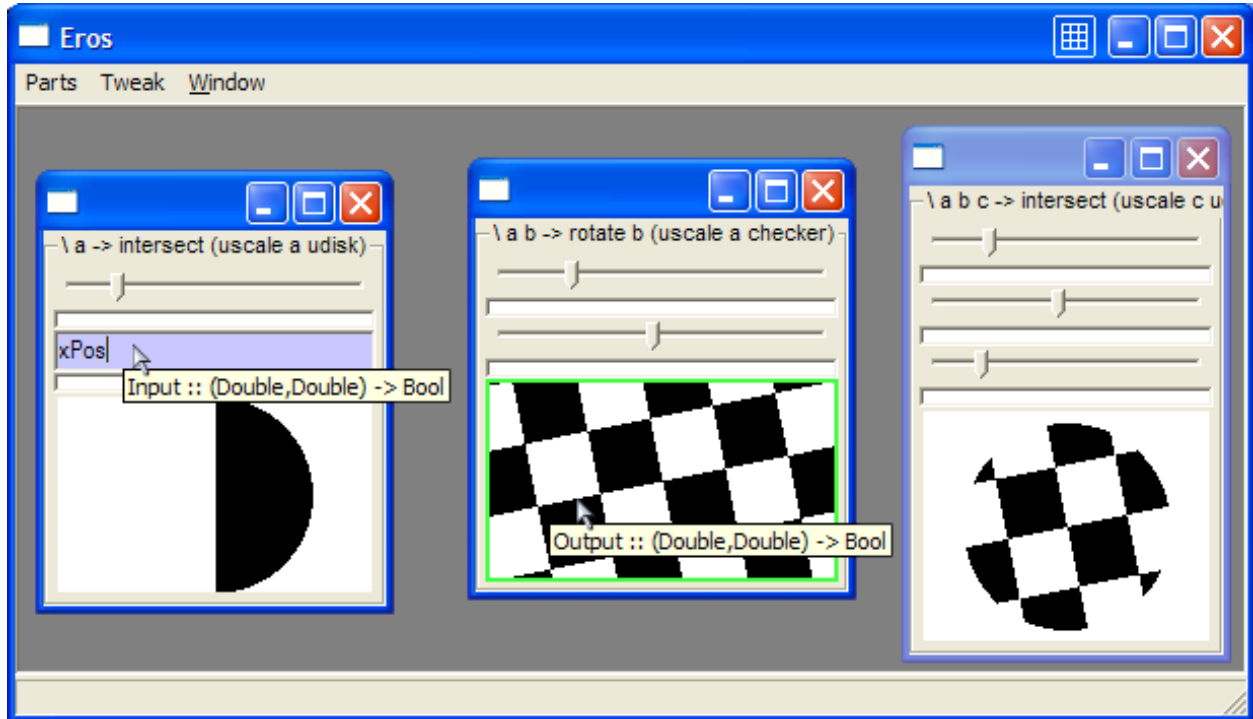


Figure 4.2: (Reproduction of Figure 14 of Elliott [38]). Three tangible values on the non-linear Eros 2D canvas. Each TV shown here is a partially applied function, with unapplied arguments on top and output below. Unapplied arguments are shown with (editable) example values. For example, the leftmost TV has two unapplied arguments: a numeric argument represented as a slider (corresponding to a scale factor), and an image input (the example input image is black for positive x values and white for negative x values). The example output (the intersection of the image with a disk) is shown below. The middle TV is a function producing a checkerboard pattern, with the scale factor and rotation angle still unapplied. TVs can be composed. The user has selected the output of the middle TV and the image input argument of the left TV. Composing these together results in the rightmost TV, in which the output of the middle TV has been used as the second argument for the right TV. The remaining unapplied arguments of both are the unapplied arguments of the result TV. (Eros is a strongly typed environment, only allowing composition between outputs and inputs of compatible type. As seen in the tooltips above, images are represented as functions of type $(\text{Double}, \text{Double}) \rightarrow \text{Bool}$, *i.e.*, coordinates to black/white.) Used with permission. Conal M. Elliott. 2007. Tangible Functional Programming. In Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07). Association for Computing Machinery, New York, NY, USA, 59-70. Fig. 14. DOI: <https://doi.org/10.1145/1291151.1291163>.

inputs, with the corresponding example output displayed below (Figure 4.2). The user could select the output of one TV, the input of another TV (of corresponding type), and compose the two together into a new TV.

Eros highlighted that there is a complementarity between *non-linear editing* and pure functional programming. Without side effects, the order of computation is negligible. The user may gather the parts they need, in any order they please, and worry later about how to assemble them. Alas, the standard practice of writing functional programs as linear, textual code obscures this fundamental opportunity for non-linearity. Placing values on a 2D canvas instead highlights it.

Non-linearity matters because not all humans are linear thinkers—not even all programmers are linear thinkers! A non-linear environment can offer a creative space more inviting to folks whose standard workflow naturally entails concrete exploration rather than abstract planning.

While Eros highlighted this complementarity between non-linear editing and pure functional programming, the Eros mechanism of composing TVs may tip the balance too far from the abstract in favor of the concrete. Once a value has been composed, it obscures *how* it came to be. The expression at the top of a TV gives some indication (Figure 4.2), but this one line is inadequate for any computation of modest size. Moreover, once composed, how does one change that computation that produced a TV? Value manipulation alone may be inadequate for carefully specifying abstract algorithms. Perhaps there is there a middle ground that allows both non-linear, concrete direct manipulation on values *and* traditional editing of ordinary code.

That middle ground is the subject of this chapter. Here, we seek an answer to the question: **How can the approachability of non-linear direct manipulation on concrete values be melded with the time-proven flexibility of text-based coding?** We would like to create a programming environment with the following four properties:

- (a.) **Value-Centric.** Like Eros, and unlike most visual programming environments, we want values—not AST elements!—to be centered in the display and, as much as possible, be the subject of the user’s direct manipulations.
- (b.) **Non-linear.** To support non-linear thinkers and exploratory programming, we want to allow the user to gather the parts they need out of order, and compose them together later.
- (c.) **Synthesis.** How to integrate recent advances in program synthesis into a practical workflow remains an open question. A value-centric interface is a natural environment to specify assertions on those displayed values, and thus also a natural environment to fulfill those assertions with a synthesizer—we want to explore this.
- (d.) **Bimodal.** Code is unavoidable: it is the language that describes computation. Ideally, a visual programming environment would not sacrifice the unique affordances of textual code—its concision and its amenability to an ecosystem of existing tooling (such as text editors, language servers, and version control). We want to offer a bimodal interface that simultaneously offers the non-linear graphical editing interface *alongside* a text-editable, traditional representation of the program’s code.

Contributions

To show how value-centric non-linear editing can meld with traditional text-based programming, we implemented a value-centric, non-linear, bimodal programming environment with synthesis features called *(The Magnificent) Maniposynth*. We demonstrate both how non-linear visual editing can integrate with linear code, as well as show novel editing features made possible by the value-centric display.

To gain an initial understanding of the system, we implemented an external corpus of 38 example programs.

For additional insights, we conducted an in-depth exploratory study with two external professional functional programmers, whose feedback informed the evolution of MANIPOSYNTH. We describe their experiences using the tool and discuss additional observations through investigative lenses from the Cognitive Dimensions of Notation framework [51].

Section 4.2 introduces MANIPOSYNTH with a running example. Section 4.3 describes the technical implementation of the tool and the synthesizer. Section 4.4 presents insights from implementing a corpus of examples and the qualitative user study. Section 4.5 presents related work, and Section 4.6 discusses avenues for continued exploration.

4.2 Overview Example

To provide an overview of interacting with MANIPOSYNTH, we follow a fictional programmer named Baklava as she re-implements the list length function from scratch.¹ Figure 4.1 shows the final result.

MANIPOSYNTH is a locally running web application designed to be opened in a web browser alongside the user’s preferred text editor. Baklava creates a blank text file named `length.ml` on her computer, starts MANIPOSYNTH in that directory, and navigates to `http://localhost:1111/length.ml` in her web browser. She positions her browser window side-by-side with Visual Studio Code and is ready to begin.

4.2.1 List length, without synthesis

To start, MANIPOSYNTH displays a blank white 2D canvas. Because MANIPOSYNTH is a live programming environment, Baklava starts by creating an example list so she can see the length operation on concrete data. Double-clicking on the canvas opens up a text box to add new code, Baklava does so and types an open bracket `[`. Because writing example data is common in MANIPOSYNTH, concrete literals up to a fixed size are offered as autocomplete options (auto-generated from the data constructors in scope, Figure 4.3). Baklava selects the list literal `[0; 0; 0]` from the autocomplete options and hits Enter.

In the code, a new `let`-binding for the list is inserted at the top level of `length.ml` and automatically given the name `int_list`. On the canvas, this binding is represented as a box display-

¹A video of this example, as well as an artifact if you would like to follow along yourself, are both available at <http://maniposynth.org>.

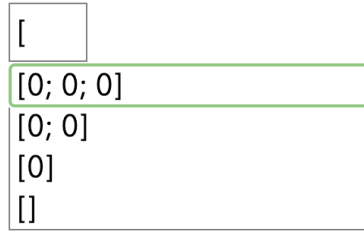


Figure 4.3: List literals offered as autocomplete options.



Figure 4.4: Tangible values (TVs) for the example list binding and the example call to length.

ing (in clockwise order, Figure 4.4, left) the binding pattern (`int_list`), the binding expression (`[ 0 ; 0 ; 0 ]`), and the result value below (also `[ 0 ; 0 ; 0 ]`, but bigger). These three elements together in a box form a *tangible value* in MANIPOSYNTH. The box may be repositioned on the 2D canvas, and the coordinates of the position are stored in the code as an AST attribute annotation on the binding, written `[@@pos 152, 49]` in the code. Arbitrary attribute annotations are supported by the standard OCaml AST which allow these properties to be preserved across program transformations. Baklava has installed a VS Code plugin to dim these attributes in the code to avoid becoming distracted by them.

To begin work on the `length` function, Baklava now creates an example call to the function: on the canvas, she double-clicks to add new code and types `length int_list`. As before, a new binding is inserted in the code (named `length_int`) and an associated tangible value (TV) appears on the canvas (Figure 4.4, right).

The `length_int` TV has two differences from the previous `int_list` TV. First, its result value (displayed as `?`, explained below) has a yellow background—this indicates the result is *not* simply a constant introduced in the expression: it came from computation elsewhere. Second, the `int_list` variable usage in the TV’s expression bears a superscript indicating the value of `int_list`, namely `[0; 0; 0]`.

In MANIPOSYNTH, using a variable that is not yet defined automatically inserts a new let-binding (TV) for the variable—in this case, `length` was not defined. Because Baklava used `length` as a function, a new function skeleton was inserted in the code (`let length x1 = (??)`).

Function TVs are displayed specially on the canvas (Figure 4.5). Immediately below the function name, a *function IO grid* displays the function input and output values encountered during execution. Immediately below the IO grid is a blank white area which is a *subcanvas* for the bindings (TVs) inside in the function, of which there are none yet. Below the subcanvas is a (non-movable) TV for the return expression and overall result value of the function. Currently the function return expression is a *hole expression*, written `(??)`. Hole expressions are placeholders, expected to be filled in later. For this reason, they are displayed larger than normal expressions

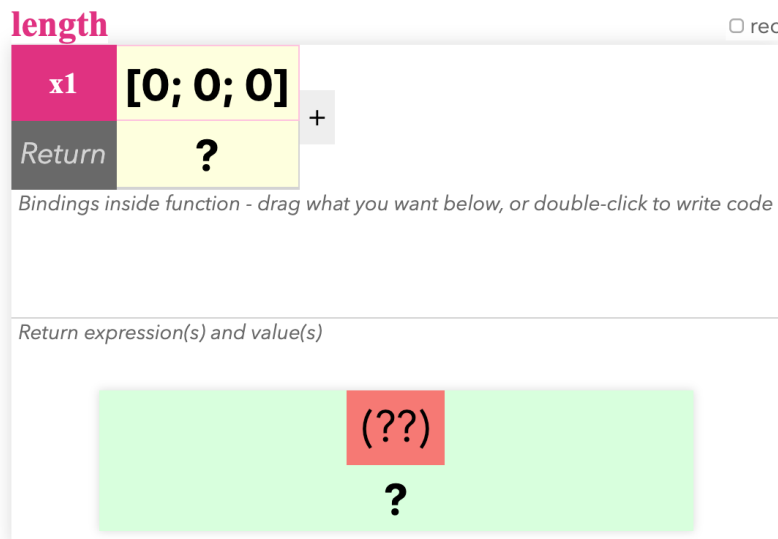


Figure 4.5: Tangible value for the function skeleton binding `let length x1 = (??)`.

(to make them easier targets for clicking) and have a slowly pulsing red background (to remind the user that the program is unfinished). While the `(??)` syntax is supported by OCaml’s editor tooling (Merlin² and its language server protocol wrapper³), programs with holes are ordinarily not executable. To continue to provide live feedback in the presence of holes, MANIPOSYNTH evaluates hole expressions `(??)` to a *hole value*, displayed as `?`. This hole value `?` is the current return value of the function shown below `(??)`—in green because it was introduced by the immediate expression above—and also shown in the “Return” row of the IO grid as well as, back on the main top-level canvas, in the result value of the `length int_list` function call.

Baklava does not like the default `x1` parameter name in the `length` function and wants to rename it. Most items in MANIPOSYNTH can be double-clicked to perform a text edit. Baklava double-clicks the pink-background `x1` to rename the variable (patterns are pink), and writes the name `list` instead. Figure 4.7a shows the code at this point.

A goal of MANIPOSYNTH is to allow non-linear editing—to not need to have all of a solution before making progress. Baklava knows she must make a recursive call to the `length` function, so, without thinking hard about what might come after, she decides to add `length (??)` inside `length`. She could double-click and type this code, but typing `(??)` requires some finger gymnastics. MANIPOSYNTH supports a large number of drag-and-drop interactions. Any green expression can be dragged to a new location to duplicate that expression—dropping on an existing expression (*e.g.*, a hole) replaces the existing expression, while dropping on a (sub)canvas inserts a new binding (TV). Values and patterns can also be dragged to expressions or (sub)canvases—when hovering over a value or pattern, a tooltip shows what expression will be inserted. Finally, a *toolbar* at the top of the window (Figure 4.6a) offers menus containing skeleton expressions: the first menu offers

²<https://github.com/ocaml/merlin>

³<https://github.com/ocaml/ocaml-lsp>

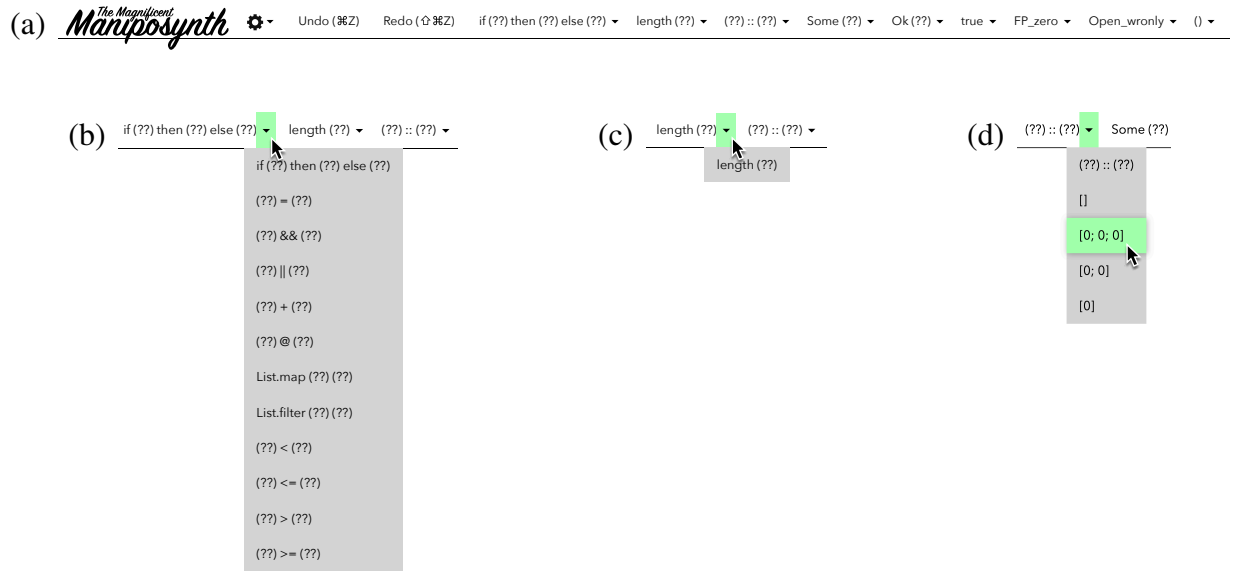


Figure 4.6: Toolbar (a), which includes a skeleton expressions menu (b), a menu of functions defined in the current file (c), and menus of expressions auto-generated from the data types in scope, such as the list expressions shown in (d).

common expressions such as `if (??) then (??) else (??)` etc. (Figure 4.6b), the second menu offers functions defined in this file (Figure 4.6c), and the remaining menus offer constructors and automatically generated example values of the types in scope (*e.g.*, Figure 4.6d; the expressions are the same as those offered by autocomplete)—if the user had any custom data types, each would appear as a menu as well.

Baklava drags `length (??)` from the toolbar into the subcanvas for her `length` function (Figure 4.7b). A `length2 = length (??)` binding is created in the code (Figure 4.7c) and an associated TV appears inside `length` (Figure 4.7d). MANIPOSYNTH also changes the top level `let length = ...` into `let rec length = ...`.

Because `(??)` produces hole value `?` instead of crashing, the `length` function is now diverging as `length (??)` calls `length (??)` which calls `length (??)` etc. MANIPOSYNTH uses fueled execution to cut off infinite loops and keep functioning. In the function IO grid, there are now extra columns showing these calls (Figure 4.7d), but other than understanding why these extra columns are there, Baklava need not pay any mind that her programming is momentarily divergent.

Baklava wants the recursive call to operate on the tail of the input list. When she moves the cursor over the input list in the IO grid, a “Destruct” button appears (Figure 4.8a), which she clicks. As shown in Figure 4.8b, a match statement (*i.e.*, case split) appears in her code, with holes for the return expression of each branch. On the display, there are a number of visual changes. In the IO grid, `hd` and `tail` pink subscripts appear inside the input list `[0; 0; 0]`, labeling the subvalues that are now bound to names by the match statement. To make these bindings even clearer, they are also represented as two new TVs in the function subcanvas. Finally, the function now has two possible return expressions: both appear as (non-movable) TVs at the bottom of the function, one is grayed out indicating it is not the branch taken when the input is `[0; 0; 0]`. Above the

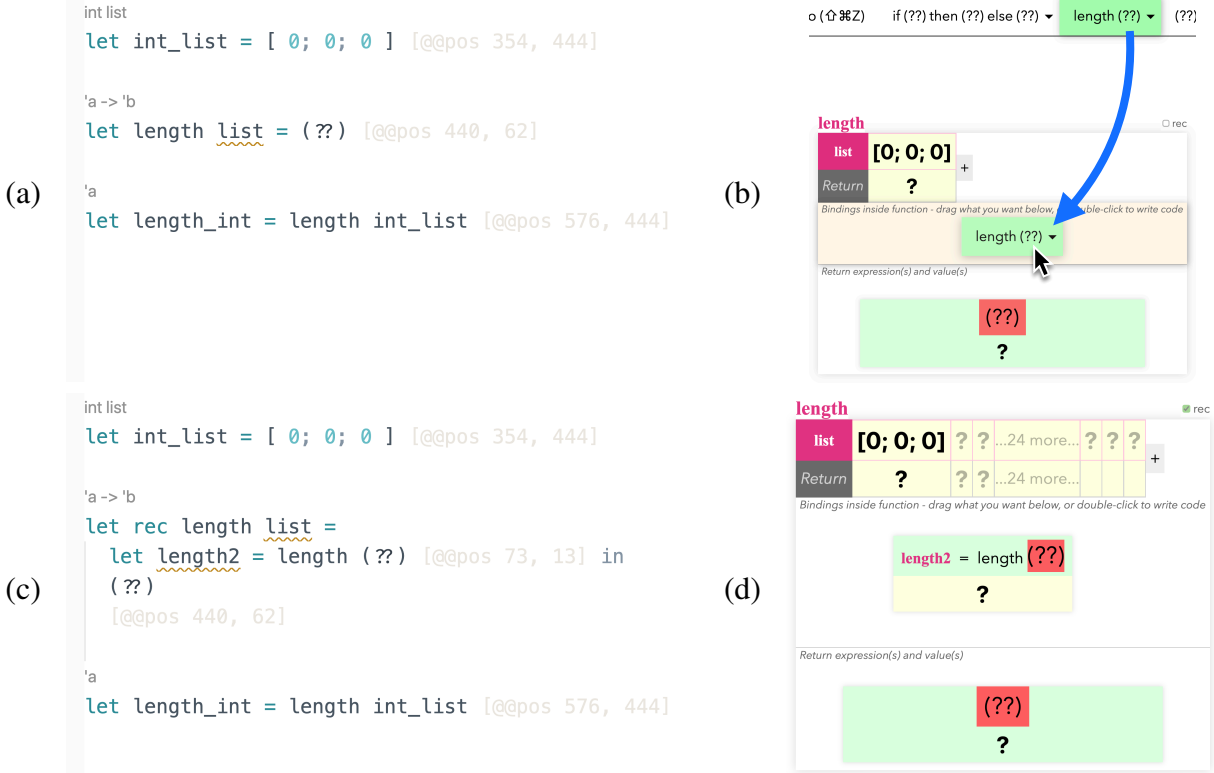


Figure 4.7: Creating a recursive call. (a) Code before. (b) Dragging a new length call into the length function. (c) Resulting code and (d) resulting length function TV.

two return TVs is an indication of the scrutinee, “ $\leftarrow$ list $\rightarrow$ ”, which allows editing of the scrutinee expression.

Now that the list tail is exposed on the subcanvas, Baklava drags it (either the pink tail name or the `[0; 0]` value below it) onto the hole in `length (??)`, transforming it into `length tail`. In her code, the binding is moved from the top level of the function into the branch in which tail exists (Figure 4.8c). Because MANIPOSYNTH embraces non-linear editing, the user should not have to worry about binding order—bindings will automatically be shuffled around as necessary to place items in the appropriate scope.

The additional calls from the recursion appear in the function IO grid, each still returning hole value `?` (Figure 4.8c, right). Baklava would like to edit the base case, so she looks for the column in the IO grid where the input is `[]`, and then clicks that column to bring that *call frame* into focus. Call frames are effectively equivalent to runtime stack frames. The TVs not executed on that call are grayed out (hd, tail, length2, and the return for the `hd::tail` branch). Baklava double-clicks the no longer grayed-out return expression `(??)` for the base case and sets it to constant `0`. (She could also have double-clicked the green-background hole value `?`; values are rendered with a green background when double-clicking them will effect an edit on the expression immediately above.)

Baklava now clicks the second-to-last call frame in the IO grid to bring into focus the call where the input is `[0]`. The return expression for this branch is still `(??)`. She notes that the TV for the



Figure 4.8: (a) A “Destruct” button appears when the cursor hovers over an input value that is an ADT. (b) Code and length function TV after destructing on the `[0; 0; 0]` input value. (c) After dragging the tail value `[0; 0]` to the red hole argument `(??)` for the recursive length call.

`length tail` call now displays a result value of 0. Baklava double-clicks the return expression `(??)` and, after typing “`1 +`” she pauses (Figure 4.9). When she began to type, MANIPOSYNTH recolored the displayed values in scope in different colors, and now, looking at the autocomplete options, she sees `1 + 0`, `1 + 0`, and `1 + 0` among the possible autocompletions—each with a different color `0` corresponding to a similarly colored `0` value elsewhere on screen. The maroon `0` is the return from `length tail`, so she chooses that. The branch return expression becomes `1 + length2`, and Baklava can now see in the IO grid that her function is returned the correct value for all inputs (Figure 4.1).

4.2.2 Undo and delete

MANIPOSYNTH supports undo/redo. Additionally, any expression may be selected by a single click and then transformed to a hole by pressing the Delete key. Entire let-binding TVs can similarly be selected and deleted, removing them from the program. Uses of the binding must be deleted before deleting the binding itself—otherwise MANIPOSYNTH will immediately recreate a binding to satisfy the otherwise unbound variable uses.

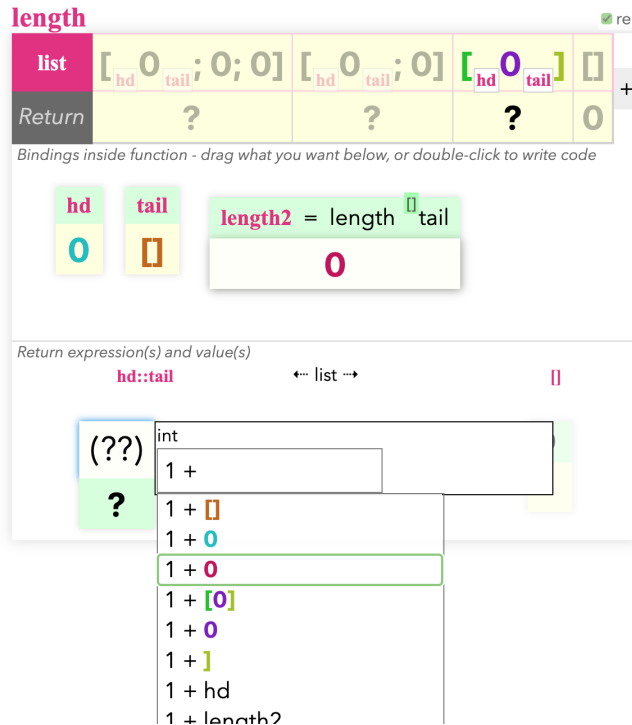


Figure 4.9: Autocompleting to a value in scope.

4.2.3 Value-centric shortcuts, and synthesis

There are usually multiple ways to complete a task in MANIPOSYNTH. Below are a few variations Baklava might have performed instead.

Drag-to-extract When Baklava needed to extract the list tail and use it for the recursive call to `length`, she clicked “Destruct” on the input value and then dragged the resulting `tail` name to her `length (??)` call. The explicit “Destruct” step can be skipped. Because MANIPOSYNTH’s goal, as much as possible, is to provide manipulations on values, *subvalues* can also be manipulated. Baklava might instead have hovered her mouse over the portion of the input list `[0; 0; 0]` that is the tail of that list, namely `; 0; 0]`, and dragged that subvalue directly to her `length (??)` call without pressing “Destruct”. The destruction will be performed automatically and the same code will result.

Autocomplete-to-extract Similarly, visible subvalues are also offered as autocompletions. Perhaps the fastest way to write list `length` is, immediately after the `length` function skeleton is created, to double-click the return hole expression, type “`1 + length`”, and then finish the new expression by selecting `; 0; 0]`, the tail of the input list, from the autocomplete options. The expression `1 + length tail` and the needed pattern match will be inserted, leaving only the base case to fill in.

Assertions Baklava started by creating an example call to `length`. To remind herself of the goal, she could have created an assertion instead: typing `length [0; 0; 0] = 3` on the top level canvas will create an `assert` statement instead of a binding with a name. Assertions are rendered in red when unsatisfied, and both the expected result (in blue) and the actual result (in black) are shown. When an assertion becomes satisfied, its result value is hidden and the assertion turns green (Figure 4.10).

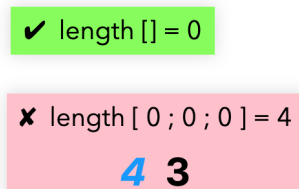


Figure 4.10: A satisfied and unsatisfied assertion.

Assertions can also be added via the function IO grid: clicking the “+” button at the right of the IO grid will create a new column in the grid, allowing Baklava to fill in the input values and expected output. Upon hitting enter, the column is reified by adding a new assertion is added at the top level, so that the function is indeed called with the specified arguments.

Program Synthesis Assertions facilitate *programming by example*, a workflow currently available in Microsoft Excel [52] but not yet available in ordinary programming. After asserting `length [0; 0; 0] = 3`, Baklava might have clicked the “Synth” button on the lower-right corner of the window. MANIPOSYNTH will use a type-and-example based approach (inspired by the MYTH [118] synthesizer) to guess hole fillings until the assertion is satisfied or the synthesizer gives up (after between 10 and 40 seconds). The synthesizer incorporates a simple statistics model and other heuristics to improve result quality (§4.3.4). In this scenario, with only the single assertion, the MANIPOSYNTH synthesizer instantly finds a filling that creates the proper case split, but places 3 as the return of the base case and `length tail` as the return of the recursive case (Figure 4.11). The result is incorporated into the code, but presented to Baklava with buttons prompting her to “Accept”, “Reject”, or “Try again”. When Baklava clicks “Try again”, in about a second the synthesizer produces the correct result, which Baklava accepts.

Subvisualizations MANIPOSYNTH offers users the ability to visualize the result of a function call on all subvalues of a displayed value—for example, the result of calling `length` on each sublist of `[0; 0; 0]` can be displayed as superscripts on the sublists. Assertions can also be specified on these visualized results, leading to the following workflow:

As before, Baklava first inserts `[0; 0; 0]` on her blank canvas. But now she clicks the `[0; 0; 0]` value to select it; a floating inspector window appears offering various type-compatible functions in scope to visualize atop `[0; 0; 0]`. Baklava foregoes these suggestions and navigates to the textbox that allows her to input a custom subvisualization (Figure 4.12a). She types “length” and hits Enter. The length function skeleton is automatically created, and each subvalue of `[0; 0; 0]` now displays a superscript `?`, the return result of the unfinished length function when applied to that subvalue. Baklava double-clicks the superscript corresponding to the whole `[0; 0; 0]`, which opens a textbox that allows her to assert on `length [0; 0; 0]`. She types “3”, hits Enter, and the appropriate assertion is created at the top level.

These subvisualizations allow users to quickly specify multiple assertions without manually creating new example values. To assert on a list of length 2, Baklava double-clicks the superscript

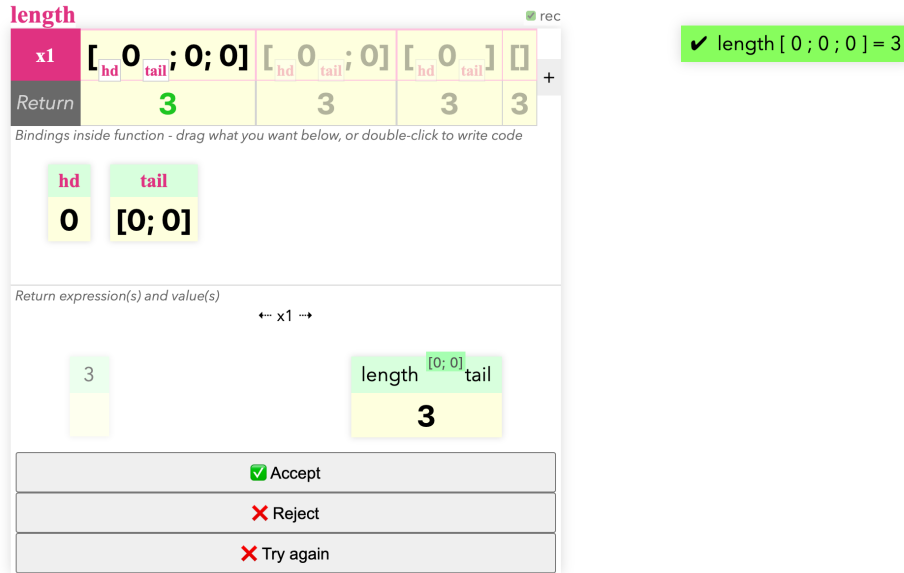


Figure 4.11: An undesired synthesis result—the next result (“Try again”) will be correct.

corresponding to the tail sublist and types “2” (Figure 4.12b). With these two assertions, the synthesizer finds the intended result in one try (Figure 4.12c shows the satisfied assets).

4.3 Implementation

With the main features of the tool demonstrated, we now describe the technical operation of MANIPOSYNTH.

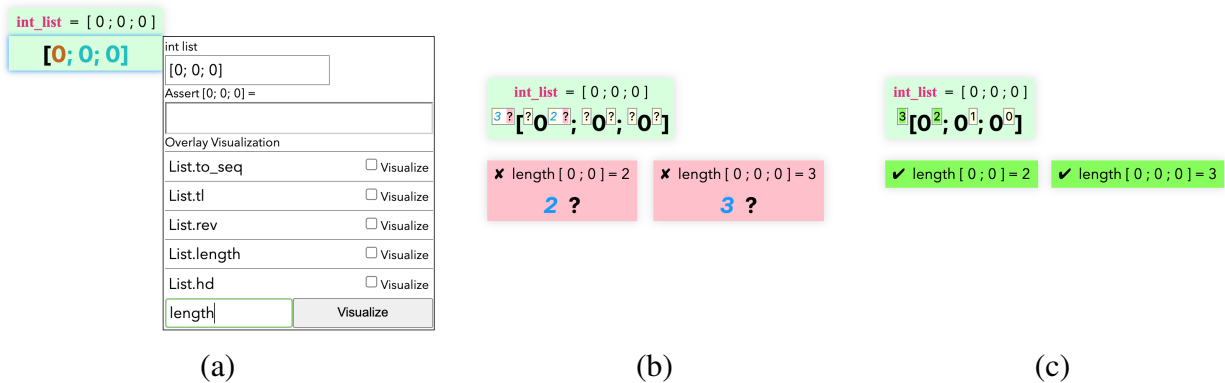


Figure 4.12: (a) Adding a subvisualization. assertions on a subvisualization, (b) before and (c) after satisfaction.

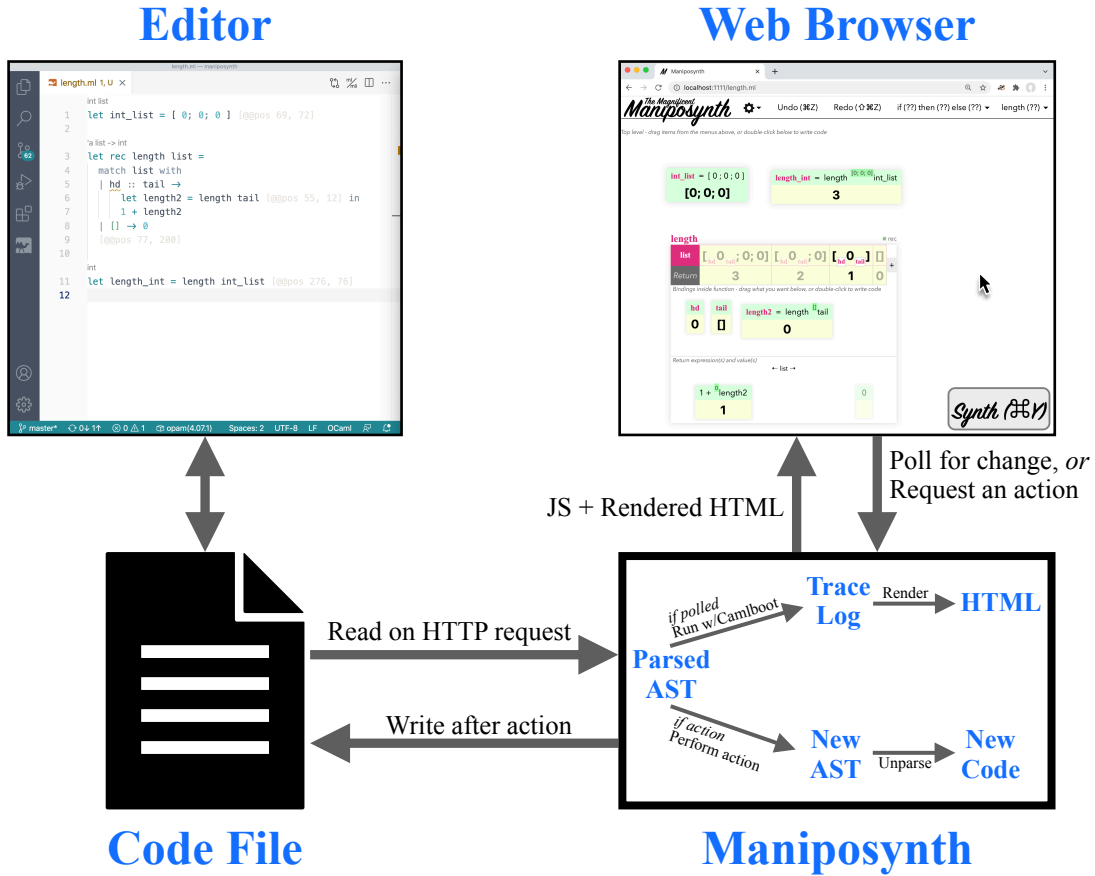


Figure 4.13: MANIPOSYNTH architecture overview.

4.3.1 Architecture Overview

MANIPOSYNTH is a web application written in about 8600 lines of OCaml (excluding the interpreter) and 2000 lines of Javascript. MANIPOSYNTH relies on OCaml’s provided compiler tools and AST data types to handle parsing, type-checking, type environment inspection, and pretty printing of modified code. Modified code is further beautified by running it through `ocamlformat`⁴ if the user has it installed. Comments are (unfortunately) discarded by OCaml’s parser.

For displaying live feedback, we need to run the program and log the runtime values flowing through the code. We modified the OCaml interpreter from the Camlboot [27] project to emit a trace of all runtime values at all execution steps. We also performed additional modifications to handle holes and assertions (described in the next section).

After MANIPOSYNTH runs the code via our modified Camlboot, MANIPOSYNTH associates runtime values from the logged execution trace with expressions in the program, and then renders HTML which is sent to the browser and displayed. Almost all OCaml-specific logic is handled server-side and baked into the HTML. The Javascript on the browser only handles TV positioning and standard GUI interaction logic. When the user performs an action, the JS sends the action to

⁴<https://github.com/ocaml-ppx/ocamlformat>

the server via HTTP, the code is modified on disk, and the server prompts the browser to reload the page to re-render the display. The browser also polls the server via HTTP so that when the file is changed on disk, the display will refresh. This overall architecture is outlined in Figure 4.13.

In the next sections we describe our modifications to the Camlboot interpreter, then how MANIPOSYNTH handles binding reordering to provide a non-linear experience, and then the mechanics of the synthesizer.

4.3.2 Interpreter

MANIPOSYNTH needs to provide live runtime values. Ordinary OCaml does have a bytecode interpreter in addition to its native code compiler—ideally we would modify this bytecode interpreter to emit a log of values during program execution. Unfortunately, OCaml performs type erasure and its runtime in-memory representation of values is ambiguous, rendering it impossible to inspect memory to recover a value’s type. Remembering the types at program locations would only partially alleviate this problem because many expressions have polymorphic (*i.e.*, generic) type, which, alas, occurs often during program construction: the function skeleton `length x1 = (??)` has type `'a → 'b`, but in the IO grid we want to be able to display any example input values as lists, not as unknown polymorphic values. Therefore, instead of trying to modify the standard OCaml interpreter or compiler, we base MANIPOSYNTH off of the OCaml interpreter in the Camlboot [27] project, an experiment in bootstrapping the OCaml compiler. The Camlboot OCaml interpreter is written in OCaml and represents all runtime values as members of an ordinary OCaml algebraic data type (ADT), which allows inspecting their type and structure at runtime, at the cost of somewhat slower execution. We modified Camlboot to handle holes and assertions, and to log runtime values during execution.

Supported subset Unmodified, the Camlboot interpreter will run a large subset of OCaml. The tooling and display in MANIPOSYNTH, however, currently only fully supports a smaller subset, shown in Figure 4.14. At the top level, programs in MANIPOSYNTH are expected to consist only of type declarations followed by (potentially recursive) let-bindings; assertions are only expected to occur at the top level. Only single-name patterns have full UI support (although internal operations such as free variable analysis will account for names in nested patterns). Supported expressions include holes, base value constants, argument-less, single-argument, and multi-argument constructors, variable usages, function introductions with an unlabeled parameter, multi-argument function applications, (potentially recursive) let-bindings, tuples, if-then-else, and pattern match case splits. Case splits are only fully supported on constructors.

Records do not have complete UI support. User-defined modules, opening modules, imperative functions, and object-oriented features are currently unsupported.

The swath of supported syntax was enough to cover the kinds of data structure manipulations we explored in our evaluation. During the user study exercises, participants rarely missed the unsupported syntax. Even so, for the tool to become practical for everyday use, the users noted it would definitely need to support modules and imperative programming.

Programs	P	$::=$	$\overline{\text{type } t = T} \ \overline{B}$
Types	T	$::=$	(standard OCaml, elided)
Top-level binding groups	B	$::=$	$\text{let } x = e$ $ $ $\text{let rec } x_1 = e_1$ $ $ $\text{let } () = \text{assert } (e_1 = e_2)$
Expressions	e	$::=$	$(??) \mid c \mid C \mid C e \mid C (e_1, \overline{e_i})$ $ $ $x \mid \text{fun } x \rightarrow e \mid e_1 \ \overline{e_i}$ $ $ $\text{let } x = e_1 \text{ in } e_2$ $ $ $\text{let rec } x_1 = e_1 \text{ in } e_b$ $ $ $(e_1, e_2, \overline{e_i})$ $ $ $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ $ $ $\text{match } e_1 \text{ with } \overline{p \rightarrow e_i}$
Case Patterns	p	$::=$	$C \mid C x \mid C (x_1, \overline{x_i})$

Figure 4.14: The subset of OCaml fully supported by MANIPOSYNTH. Overlines denote zero or more of the syntactic element. Unsupported expressions and patterns will still be displayed but will not have full UI support.

Holes and Bombs It is best for the user if live feedback is available even if the program is incomplete. While we could have the interpreter crash on the first hole, that may still be too restrictive, *e.g.*, if the expression is new and is still dead code then the presence of the hole should be inconsequential to the rest of execution. A thorough solution would be to adopt the Hazelnut Live semantics, which describe how to evaluate *around* holes [116]. When holes reach elimination position, terms become stuck (*e.g.*, what should hole plus hole be? Or which case branch should we take when the scrutinee is a hole?). Hazelnut Live evaluates around the term by, effectively, turning the stuck term into a value which is propagated until it causes another term to become stuck, and so on. While this can offer intriguing UI possibilities in its own right (outlined in [116]), it requires that we display the stuck terms to users as if they are values. MANIPOSYNTH may do so eventually, but our display is already full of elements to keep track of. Asking users to make sense of stuck terms, displayed far from their origin, might be confusing.

MANIPOSYNTH instead adopts a middle ground. We evaluate around holes but not around any other expressions. In practice, hole expression $(??)$ introduces a hole value $?$ that remembers the introduction location and captures a closure. (This closure is not displayed to the user but is occasionally used during synthesis when propagating assertions to constraints on holes.) Hole values propagate through evaluation—if unused, the evaluation can continue normally. If a hole value reaches elimination position (*e.g.*, $? + ?$), we resolve the expression to a special Bomb value (displayed as 💣). Similarly, if a Bomb reaches elimination position, another Bomb is produced. In this way, execution can continue and expressions unrelated to the unfinished code can continue to provide live feedback.

Finally, to prevent infinite loops from stalling the interpreter or inhibiting live feedback, MANIPOSYNTH uses fueled execution to abort when the right-hand side of a binding takes too long

to execute. Each top level let-binding is allocated 1000 units of fuel (execution steps), and each non-top level let-binding reserves 50 units for later execution in case the binding diverges. When the interpreter runs out of fuel, execution drops back to the let-binding, all patterns at the binding are bound to Bomb, and execution continues if there is any remaining fuel. Divergence is moderately common, because recursive call skeletons like `length` (??) from the Overview Example will repeatedly call the function with hole value. Thus it is important that execution bypasses the divergence with some fuel reserved for later bindings so that later TVs will still show live values in the display.

Assertion logging When an assertion is encountered during execution, ordinarily OCaml would evaluate the assertion and then throw an exception if the asserted expression returns false. Instead, MANIPSYNTH evaluates the assertion and logs the result for later, but never raises an exception. The assertion logging only supports equality comparisons for now; unsupported assertions are skipped. The expected expression (the right-hand side), the subject expression (the left-hand side), and the result values of both are logged. Logged assertions are used both for synthesis and to display blue expected values to the user wherever that same expression and value is encountered during execution (Figure 4.10).

Tracing In addition to logging assertions, MANIPSYNTH also logs other execution information needed to render its display. Our modified interpreter records information in two places: each execution step is entered into a global log, and we also tag side information onto runtime values.

At each execution step and at each pattern bind we add a log entry to a global trace, recording the current AST location, the call frame number (from a global counter incremented upon each function call), the result value or value being pattern matched against, and the execution environment of bound variables. When producing the display, this information is queried to discover which values flowed through which locations and under which call frames, and the appropriate values are rendered.

For convenience, we also store extra information on values as well. On values we log the type of the value when it is introduced (or returned from a built-in external primitive such as addition) so we have a concrete type associated with the value even if the value is later used in a polymorphic context. To the value we also attach a list of frame numbers and AST location of the expressions and patterns the value passes through, to, *e.g.*, conveniently interrogate where a value was first introduced. For example, to display function closure values, we find where the closure was bound to a name and display that name as the rendered closure value.

The above tracing mechanisms are sufficient to render the live feedback in the MANIPSYNTH display. Although the extensive logging might be expected to slow down execution, at present MANIPSYNTH is only applied to small programs and HTML rendering tends to take considerably longer than the initial execution, but the MANIPSYNTH server is generally able to provide a response in under 200ms.

4.3.3 Fluid Binding Order

A primary goal of MANIPOSYNTH is to offer a non-linear editing experience. The program is therefore rendered on a 2D canvas, which means we do not want users to have to worry about binding ordering. If the user sees a name on the canvas, they should be able to reference that name in the expression they are editing, even if, in the written code, that name is introduced later in the program.

Reordering bindings To support this non-linear workflow, only limited variable shadowing is supported. Nested bindings may shadow a top-level definition, but otherwise all names are assumed to be unique within each top-level definition. After every user action, MANIPOSYNTH leverages these assumptions to reorder bindings, move bindings into match statement branches, and to add a `rec` flag on bindings that refer to themselves. Only single recursion can be inferred (multiple recursion must be added manually in the text editor).

The overall consequence is that users rarely have to think about binding ordering in their code—they can continue to use the TVs on the display as if they are all appropriately visible to each other.

Inserting case splits Recall that users can grab any displayed subvalue and drag it into their program to induce a pattern match. Internally, the process works as follows. Whenever a user hovers their mouse over a value, a tooltip appears previewing the expression that will be inserted. For subvalues, the expression is an incomplete pattern match, such as `match list with | hd::tail -> tail`. Deeper extractions are also possible, e.g., `match (match list with | hd::tail -> tail) with | hd2::tail2 -> tail2`, but not often useful.

When the user drops the subvalue into their program, the expression is initially (internally) inserted as is, e.g., `let tail2 = match list with | hd::tail -> tail in ....` A series of program transforms then rearranges match statements as follows:

1. All `let`-bindings at the beginning of the function are pushed down and duplicated into each pre-existing case branch. They may be pulled back out at the end of the process below. This push-down has the effect that all newly inserted match statements are now children of any prior match statements already inserted.
2. If the user dragged a subvalue that has already been extracted (or has extracted a deeper subvalue and one of its parents has already been extracted), we do not want to insert superfluous pattern matches. We want to reuse the case splits that already exist. Relying on the prior step that ensured all bindings are now in scope of all variables introduced in cases, a static analysis pass simplifies nested case splits on the same variable: if a match on `list` already exists, then the copy of `let tail2 = match list with | hd::tail -> tail in ...` that was pushed into the pre-existing `cons` case will be simplified to `let tail2 = tail in ...` and the copy pushed into the pre-existing empty list case will be simplified to `let tail2 = match list with in ..., i.e.,` a match with no cases, which marks the binding for removal.
3. Each `let`-binding that has such an empty match anywhere in its left-hand-side is removed.

4. Any surviving match statement is not redundant, but still in a non-idiomatic position. A series of local rewrite rules floats the match statements up to the outermost level of the function, *e.g.*, `f (match list with hd::tail -> tail)` becomes `match list with hd::tail -> f tail`, etc.
5. All let-bindings, previously floated down into all case branches, are now floated back up as far as possible to the top level of the function: a binding is pulled up outside of match branches when both (a) the same binding exists in all branches, *i.e.*, it was valid in all branches and not removed, and (b) in all branches, the binding is not dependent (or transitively dependent) on any variables introduced for the case branch.
6. Newly inserted matches are now at the top level, but may still be missing cases. Incomplete pattern matches are filled in with the missing branches, with a hole expression (`??`) in each new case.
7. Bindings that are only simple renamings, such as `let tail2 = tail in ...`, are removed—these happen when the user performs an extraction of a subvalue that was already previously extracted.

The above algorithm produces idiomatic match statements, with the match wrapped in all the let-bindings that are not dependent on variables introduced in the branches. For functions with a single match, the above algorithm performed well in our evaluation—it was never a source of trouble. For nested matches with independent scrutinees, a current limitation of MANIPOSYNTH is there is no refactoring to flip the nesting order.

Inserting undefined variables Finally, in keeping with the goal of non-linearity, we also want to allow users to use a variable before it has been defined anywhere. Therefore, after the above processes, any remaining variables that are used but not defined are introduced in a new let-binding. Each new variable is either bound to hole or, if the variable is used as a function in an application, bound to a new function skeleton with the appropriate number of parameters. Thus, typing `length [0; 0; 0]` on an empty canvas results in the skeleton length function seen in the Overview Example.

4.3.4 Synthesizer

As discussed in the Overview Example, MANIPOSYNTH includes a programming by examples (PBE) workflow to help users finish their incomplete code. Here we detail the program synthesizer’s operation and our design choices in its implementation.

The MANIPOSYNTH synthesizer does not contain any new “big” ideas, but the design was carefully chosen for our setting. To be as practical as possible, we had four goals:

1. **Few examples.** To reduce the burden on the user, we would like the synthesizer to operate with few examples. For example, the MYTH synthesizer [118] also targeted a subset of OCaml, but required the user-provided examples include all needed recursive calls—*e.g.*, `length [0;0] = 2`, `length [0] = 1`, and `length [] = 0`. But because this “trace

Expressions	e	$::=$	fun $x \rightarrow e$
			match e_1 with $\overline{p \rightarrow e_i}$
			c
			x
			$x \overline{e_i}$
			$C \mid C e \mid C (e_1, \overline{e_i})$
			if e_1 then e_2 else e_3
Case Patterns	p	$::=$	$C \mid C x \mid C (x_1, \overline{x_i})$

Figure 4.15: An overview of the subset of OCaml the synthesizer can emit; also a subset of Figure 4.14.

completeness” requirement is burdensome, we would like our synthesizer to operate with only one or two examples.

2. **No type annotations.** Similarly, MYTH and its successor SMYTH [95] require holes to have types before synthesis, which requires manual annotation. We would like to relieve the user of this responsibility and operate without explicit type annotations.
3. **As simple as possible.** The primary goal of MANIPOSYNTH is to explore non-linear editing, not synthesis per se, so we wanted to keep our synthesizer as simple as possible. For now, we did not adapt SMYTH because, although it appropriately relaxes the trace-completeness requirement, SMYTH utilizes a complicated synthesis schedule and requires the Hazelnut Live machinery [116] for evaluating around holes. Even with the mechanisms described below, our synthesizer is around 1300 lines of OCaml, compared to more than 5000 for SMYTH [95].
4. **Quality results.** When given only a few examples, synthesizers are notorious for producing simple but undesirable results,⁵ which limits their utility. This problem is compounded when the synthesizer is asked to operate in practical environments with many variables in scope, rather than unrealistic bare minimal execution environments often used for synthesizer benchmarks. Our synthesizer should operate with the OCaml standard Pervasives library open in the execution environment so the synthesizer may choose to use *e.g.*, addition and subtraction. We adopt statistics and heuristics to make this tractable.

MYTH used type information to dramatically reduce the search space and to intelligently introduce case splits. So, to meet the above goals, we built a MYTH-like synthesizer which uses both types and examples to guide its guessing. Unlike MYTH, however, we relax the trace-completeness requirement and instead rely on a statistics model to guess more likely terms sooner. Our target language subset, the statistics model, and our other heuristic choices are described below.

⁵For example, “January, February, Maruary” <https://techcommunity.microsoft.com/t5/excel/flash-fill-wrong-pattern-for-filling-month-names/m-p/355213>

Expressions	e	$::=$	$52\%x$ $ $ $20\%e_1 \bar{e}_i$ $ $ $10\%\text{fun } x \rightarrow e$ $ $ $8.1\%ctor$ $ $ $6.6\%c$ $ $ $1.9\%\text{match } e_1 \text{ with } \overline{C...} \rightarrow e_i$ $ $ $1.3\%\text{if } e_1 \text{ then } e_2 \text{ else } e_3$
Names	x	$::=$	$73\%localName \mid 27\%pervasivesName$
Local names	$localName$	$::=$	$31\%MostRecentlyIntroduced$ $ $ $20\%2ndMostRecentlyIntroduced$ $ $ $11\%3rdMostRecentlyIntroduced$ $ $ $...etc...$
Pervasives names	$pervasivesName$	$::=$	$4.0\%(+) \mid 2.9\%(=) \mid 1.9\%(-) \mid ...other\ non-imperative\ names...$
Constructors	$ctor$	$::=$	$54\%pervasivesCtor \mid 46\%localCtor$
Pervasives ctors	$pervasivesCtor$	$::=$	$e_1^{24\%}::e_2 \mid 20\%[] \mid 15\%() \mid 13\%false \mid 8.2\%true \mid ...etc...$
User ctors	$localCtor$	$::=$	$...constructors\ defined\ in\ file\ (uniform\ probability)...$
Constants	c	$::=$	$52\%int \mid 45\%str \mid 2.2\%char \mid 0.43\%float$
Int literals	int	$::=$	$37\%0 \mid 26\%1 \mid 10\%2 \mid 3.4\%3 \mid 2.7\%4 \mid 1.6\%8 \mid 1.5\%5 \mid 1.1\%-1$
String literals	str	$::=$	$2.9\%"" \mid 0.62\%". " \mid 0.59\%")" \mid ...other\ 1-char\ strs\ from\ corpus...$
Char literals	$char$	$::=$	$9.1\%'\backslash n' \mid 8.4\%'\backslash ' \mid 6.2\%'\backslash \backslash ' \mid ...other\ chars\ from\ corpus...$
Float literals	$float$	$::=$	$25\%0.0 \mid 20\%1.0 \mid 18\%0.0 \mid 8.3\%0.5 \mid 5.0\%10.0$

Figure 4.16: The grammar used for the statistics model. Each production is associated with a probability.

Synthesizable subset Figure 4.15 describes the subset of OCaml that the synthesizer may produce as it attempts to fill holes in the program. The synthesizer can introduce functions, match statements, constants (drawn from a corpus), variable uses, function calls with a variable in function position, constructor uses, and if-then-else statements.

Statistics model Naively, guess-and-check will produce a large number of unlikely programs. Incorporating a statistics model guides the synthesizer to guess more likely programs sooner and can speed up synthesis by multiple orders of magnitude [88, 75]. It also has the potential to offer the user more reasonable results when fewer examples are given.

We model program likelihood using a probabilistic context-free grammar (PCFG). A PCFG assigns a probability to each production rule in a grammar. For our synthesizer, we derived the probabilities of the production rules from a corpus of OCaml code—namely, the source files required to build the OCaml native compiler. The overall probability of a program term is the product of the probability of the production rule of the term with (recursively) the probability of all its sub-terms.

Our PCFG grammar is given in Figure 4.16. We subdivided several kinds of terms into multiple production rules in order to provide more precise probabilities: constants are divided by the constant type, constructors are classified as either a user constructor (*i.e.*, defined in the same module or a parent module) or from elsewhere, and, most notably, names are classified as local (*i.e.*, defined in the same module or a parent module) or from elsewhere. User constructors are assigned equal probability with each other. Local names are denoted by how recently they were introduced into the execution environment—more recently introduced names are much more probable than less recently introduced names. The probabilities of constant literals and external (*e.g.*, standard library) names and constructors are derived directly from how often those names and constructors appear in the corpus.

After calculating these probabilities from the corpus, we further reduced the search space for the synthesizer. We unscientifically trimmed down the list of possible constant literals to the 5-10 most common in the corpus for each type. We also limited the initial execution environment to constructors and functions in the globally-imported Pervasives module, removing functions involving imperative features (they are not supported by MANIPSYNTH) as well as several floating point primitive operators unimplemented by Camlboot. Finally, we also excluded OCaml’s polymorphic compare, which, in practice, the synthesizer would often use in surprising ways to produce the numbers 0 and 1, *e.g.*, compare `x x` evaluates to 0. For simplicity, we did not renormalize probabilities after the above trimmings to the production rules.

As an example, the most probable term (*i.e.*, what the synthesizer should guess first) is always the most recently introduced variable. The production rule for an identifier has probability 52%, the probability that the identifier is local is 73%, and the probability a local identifier is the most recently introduced variable is 31%, for an overall probability of 12%.

Type-based refinement MYTH divides synthesis into two processes. The *type-based refinement* process introduces program sketches—either function introductions or case splits—at holes based on the type at the hole and the types of variables in scope (to find an appropriate scrutinee to introduce a match). These sketches contain further holes to fill (*i.e.*, for the function body and the match branches). Type-based refinement alternates with the *type-directed guessing* process, which performs simple type-constrained term enumeration to guess a term to fill existing holes (guessing will not introduce functions or match statements).

As part of the type-based refinement process, MYTH will push the user’s examples to the frontier of synthesis. For example, if the user asserts that a hole should output 0 when given the input [], MYTH will refine the hole to `fun x -> (??)` and refine the example to note that (??) should resolve to 0 when `x` is bound to []. This allows MYTH to quickly verify when a hole filling satisfies all given examples.

However, MYTH’s implementation of this example refinement machinery requires that the user provide all assertions directly on holes. Users cannot write `assert (length [] = 0)`. Instead, they must write, essentially, `let length = ((??) such that { [] => 0 })`. It would be better if users could invoke synthesis on a partial sketch.

To allow assertions on program sketches rather than only on holes, SMYTH uses “Live Unevaluation” [95] to push top-level assertions down to constraints directly on holes. Pushing down the assertions is not fundamentally required to perform synthesis—a synthesizer may guess terms

at the holes and check the top-level assertions (indeed MANIPOSYNTH does so)—but pushing the assertions down to the holes provides information about the hole. MANIPOSYNTH adapts an effectively⁶ identical approach to SMYTH, and refines the examples through the sketch yielding constraints on holes. MANIPOSYNTH uses these hole constraints for two purposes:

1. Refining a hole into a function requires knowing that the hole is at arrow type. This can easily be determined if the program is explicitly typed, as required in MYTH and SMYTH. MANIPOSYNTH, however, allows untyped sketches which often start at polymorphic type. For example, in an initial sketch `let length = (??)`, the `length` variable has type `'a`. When `assert (length [] = 0)` is pushed down to the hole, we know the hole must satisfy the requirement `[]  $\Rightarrow$  0`, *i.e.*, must be a function that when given `[]` produces `0`. If all the constraints on a hole are of that form $v_1 \Rightarrow v_2$, then MANIPOSYNTH will attempt to refine the hole into `fun x -> (??)`.
2. To speed synthesis and produce more relevant results, MANIPOSYNTH tracks whether a generated (sub)term is allowed to be constant or not (*e.g.*, MANIPOSYNTH requires that at least one argument in a function call must be non-constant). If multiple different examples reach a hole, MANIPOSYNTH will not generate a constant term for that hole. Similarly, if a single example reaches a hole, MANIPOSYNTH will exclude all constants from consideration for that hole, except for the value asserted on the hole.

MANIPOSYNTH currently only performs at most one level of refinement—introducing only one function or one case split. Further (or initial) case splits can be introduced by the user with the “Destruct” button in the UI. Introducing functions is rarely needed in practice because, in the MANIPOSYNTH UI, undefined variables are inserted with a function skeleton.

Type-directed guessing Terms are enumerated (guessed) at holes up to a given probability [85]. During term enumeration, the probability bound is treated as a resource that is iteratively distributed between holes, and then between guessed subterms. When the probability is exhausted, no further enumeration occurs on a subtree. If a candidate subterm’s probability is above the final probability bound, the remaining probability is available for enumerating sibling terms.

Within a hole, term enumeration is type-directed, starting from the type of the hole. Leveraging OCaml’s type checking machinery, subterms are unified during the enumeration process to narrow the type. For example, if a hole has type `int` and the synthesizer guesses a call to `max` which has type `'a  $\rightarrow$  'a  $\rightarrow$  'a`, the return type will be unified with `int` and the synthesizer will only guess terms of type `int` for the arguments.

As discussed above, initial sketches often have polymorphic types unhelpful to the synthesizer. To tighten these bounds before term enumeration, the input and output types of functions are

⁶Some sketches prevent the propagation of constraints—for example, if a sketch has a hole in scrutinee position, it is impossible to know which branch to take. The push-down procedure is stuck and any holes in the branches will not be able to receive their constraints. SMYTH resolves this scenario by speculatively filling the scrutinee hole while pushing down the constraints. We opt to avoid this extra machinery, resulting in a simpler implementation, but at the cost that we cannot resolve holes in an iterative fashion—satisfying the requirements of one hole before moving on to the next—which would enable faster synthesis.

speculatively chosen based on the given examples. If the given examples differ in type, multiple speculative types are explored (terms are guessed in each). Future versions of MANIPOSYNTH may use anti-unification instead to be more precise. The speculative types are not included in the final synthesis result in case the inferred code has a more general type.

As briefly mentioned, in order to reduce the number of unnatural synthesis results MANIPOSYNTH limits where constant terms may appear. The term enumeration eagerly tracks whether a term may be constant or not. A term is considered to be non-constant if it uses any introduced function parameter, or any variable introduced under the outermost enclosing function. At most one hole may be constant, and, when introducing a function call, at least one argument must be non-constant. If a previously filled hole is constant, or we have reached the last function argument and all other arguments are constant, then the subterm enumeration will avoid enumerating constants. This speeds synthesis and produces more reasonable results.

Final heuristics Finally, when all holes have been filled with type-appropriate terms within the probability bound, the candidate program is accepted if:

1. All assertions are satisfied. (Fueled execution prevents divergence when checking assertions.)
2. At most one hole is filled with a constant.
3. All introduced function parameters are used.
4. The result at a hole has not previously been rejected by the user.
5. Execution of the examples encounters all filled hole locations (*i.e.*, the execution path does not somehow avoid a hole).

If no satisfying hole fillings are found at the initial probability bound and a 10 second timeout has not been reached, guessing is restarted with a new bound 1/20 of the old. If there is a valid candidate program, the highest probability such program is returned. Enumeration within a given probability bound is not precisely from highest to lowest probability, however, so timeout will not interrupt a round of synthesis until the full space of that probability bound is explored. Thus for the user, the timeout they experience varies between 10 and 40 seconds.

4.4 Evaluation

To evaluate to what degree MANIPOSYNTH meets its goal of providing value-centric, non-linear editing, we performed two evaluations. In one, an expert user (the dissertation author) used MANIPOSYNTH to implement 38 functions from the exercises and homework of a course on functional data structures [112]. In the second, to provide additional qualitative insights on the operation of the tool, we hired two professional OCaml programmers, and guided and observed them as they used MANIPOSYNTH to implement a subset of the exercises from the above course.

4.4.1 Study Setups

The first six lessons of the course [112] cover natural numbers (via an ADT), various list functions, leaf trees, binary trees, binary search trees, and a form of binary search tree that also records on each node the minimum value of all its descendants. We excluded the six functions on this specialized tree because of time constraints. The course exercises and homework spanned 38 functions on the remaining data structures. The first author implemented each of these functions in MANIPOSYNTH with the code editor hidden. MANIPOSYNTH was set up to log the number and kinds of actions that were preformed. We report on these in the next section. Our aim was to show that the MANIPOSYNTH interface was able to implement these exercises and to discover if there were any obvious trouble points.

For our user study, we advertised on <https://discuss.ocaml.org/> and hired two professional OCaml programmers to use the tool for three sessions each. Sessions were spread over three weeks, with each session lasting two hours. Participant 1 (P1) and Participant 2 (P2) had 5 and 11 years, respectively, of professional OCaml experience. The participants ran MANIPOSYNTH on their own computers alongside their preferred text editor (Vim for both). The study facilitator connected via video conference and recorded the sessions. Participants implemented their choice of exercises from the list, or suggested their own task to complete. The facilitator provided varying amounts of guidance throughout, starting with close guidance to teach the tool and transitioning to less intervention as participants became more comfortable. After each exercise and at the end of each session, participants discussed a series of questions posed by the facilitator. In concert with MANIPOSYNTH’s four design principles—value-centric operation, non-linearity, supporting synthesis, and bimodality—we aimed to gain insights about the following four research questions, along with three supplemental questions:

RQ1. How do users interact with the live values?

RQ2. How do users work non-linearly?

RQ3. How do users interact with program synthesis?

RQ4. How do users interact with their text editor?

SQ1. What are the pain points? How might the system be improved?

SQ2. How comfortable are participants with the tool—can they complete an exercise without guidance?

SQ3. Using the lenses of the Cognitive Dimensions of Notations [51], what additional insights do we learn about the tool?

For each participant, the first session introduced to the tool without synthesis, the second session introduced synthesis (before the synthesizer had a statistics model), and the third session concluded with the tool as presented here. Based on participant feedback, we fixed bugs and made improvements between each session.

Function	LOC	Asserts	Time	Mouse	Keybd	Un/Re/Del	TypeErr	Crash
nat_plus	5		0.8	6	5			
nat_minus	8		1.9	6	11			
nat_mult	9		1.4	8	6			
nat_exp	13		2.1	9	6			
nat_factorial	13		1.6	8	4			
nat_map_sumi	10		2.6	11	5		1	
count	9		1.9	9	11			
length	4		0.3	1	7			
snoc	8	1	2.4	8	12	2		
reverse	8		1.5	4	9			
nat_list_max	17		4.6	23	21			
nat_list_sum	13		1.1	9	4			
fold	9		3.2	14	6			
shuffles	14		14.5	25	28	2		
contains	9		2.2	10	13	1		
distinct	16		2.4	9	11	2		
foldl	10	1	1.5	10	6		1	
foldr	8	1	1.8	10	5			
slice	12	3	9.8	19	22	4		
append	8	1	1.4	7	9			
sort_by	21	3	6.2	17	29			
quickselect	13	1	13.1	19	38	1	1	
sort	16	3	5.6	11	32	2		
ltree_inorder	12	1	2.9	7	20	1	1	
ltree_fold	13	1	3.1	13	13			
ltree_mirror	11	1	4.4	12	6		1	1
bst_contains	14	3	6.6	11	32	1		
bst_contains2	17	5	10.4	20	41	2		
btree_join	34	2	61.7	82	64	51		2
bst_delete	36	2	14.4	31	24	4		
bstd_valid	29	3	32.2	63	100	4	1	
bstd_insert	18	2	8.0	38	23	3		
bstd_count	21	1	7.6	15	32	1		
bst_in_range	31	3	9.3	23	39	3		
btree_enum	29	3	19.2	31	51	6	3	
btree_height	15	1	1.9	11	14			
btree_pretty	14	1	3.7	4	21		4	
btree_same_shape	19	1	8.1	14	34	7		
Total	566	44	277.6	628	814	97	13	3

Table 4.1: Example exercises, with lines of code, number of assertions, time in minutes, number of mouse actions (excluding selection and undo/redo), number of keyboard interactions (*e.g.*, typing in a textbox), number of undo/redo/deletions, number of type errors encountered, and number of times MANIPOSYNTH crashed and the file had to be repaired in the text editor.


```
tree2 = Node (Node (Empty , 5 , Node (Empty , 6 , Empty )), 7 , Empty )
```

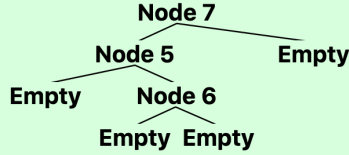


Figure 4.17: MANIPOSYNTH beautifies tree-like values.

4.4.2 Results

Example corpus implementation The expert implementer spent about 4.5 hours total implementing the 38 functions, resulting in about 550 lines of code (including AST annotations and examples written for live feedback, but excluding whitespace). A quantitative summary of these example exercises is shown in Table 4.1. There are often many paths to a correct implementation, so to constrain the workflow the implementer did not use synthesis, and did not use the ordinary text editor except to copy an earlier function into a later exercise (in case of dependencies) or when MANIPOSYNTH crashed on the given code. Note that for the functions operating on tree-like datatypes (ADTs with multiple children of the same recursive type), MANIPOSYNTH’s live display helpfully draws the trees as trees (Figure 4.17). Primarily, these 38 examples show that the MANIPOSYNTH UI is expressive enough to create these programs. We also noticed two qualitative takeaways from the exercises. First, although we believe bimodality is an important property for the grounding and long-term practicality of the tool, it is possible to hide the text editor and work entirely in MANIPOSYNTH. Second, even with live feedback available, it is not always used—a theme that reemerged in our user study. We now discuss these two observations in more detail.

The implementer used MANIPOSYNTH in fullscreen with their text editor hidden. The text editor was only used to initialize code by pasting from a previous exercise if prior code was needed, and in cases where MANIPOSYNTH could not run the code and crashed (*e.g.*, when the implementer tried to raise an exception in unreachable code, but instead raised the exception in reachable code!). Overall, MANIPOSYNTH operated well, even without the textual view, with a couple notable caveats.

The non-linearity machinery largely worked—the implementer did not have trouble with binding order. Even so, the implementer was careful to name extracted subvalues well, because the positioning of the extracted TVs on the 2D display did not (by default) reflect the items’ positions in the original data structure. Particularly for nested matches, there were sometimes a large number of these extracted values displayed and it was hard to keep track of them. The implementer found it helpful to reposition the extracted TVs (the TVs representing case split branch patterns) to reflect those original positions. Ordinary textual code for case split patterns would provide some of these positional cues without manual interaction.

On a few exercises where nested match statements were needed, MANIPOSYNTH initially created the wrong nested match structure; with the non-linear display, this is a bit hard to notice and requires thinking about the match nesting structure shown in the return TVs area. Regardless, the implementer was able to work around the trouble by undoing and triggering the destructions

differently.

The second observation from these examples is that, when working with MANIPOSYNTH, the implementer noticed that they seem to flip between two mental modes: these modes correspond roughly to focusing on displayed values versus focusing on expressions. In the *value-oriented* mode, the implementer would put their attention on the live values to consider if the code is operating correctly; in the *expression-oriented* mode, the implementer would read expressions and simulate the computer’s operation in their head.⁷ As a matter of discipline, the implementer was trying to push themselves to consider and use the live values, but still often found themselves reverting to thinking only about the expressions instead. We have three hypotheses for why there seems to be a tendency to revert to focusing on expressions instead of values.

Hypothesis A: Expressions are a concise language that represent abstractions, and programming is, fundamentally, abstract. Language is how humans handle abstraction and so our brains are good at language. The concrete values do not immediately represent the abstraction.

Hypothesis B: Seasoned programmers have years and years of experience reading code and simulating the computer in their head, our brains have adapted to it and it feels natural.

Hypothesis C: MANIPOSYNTH did not provide enough live feedback and forced the implementer to consider the expressions. In some cases this was immediately true: MANIPOSYNTH currently only displays the first and last three call frames, with no option to see the others. Despite the implementer’s resolve to try to work with values, sometimes those values were in unavailable call frames. Additionally, when initially trying to figure out what algorithm was needed at all, the implementer found it easier to work out the initial algorithm sketch in their head rather than guess and check in MANIPOSYNTH.

Most likely, all three of these reasons contributed towards a tendency to put attention back on expressions rather than values. A similar theme was observed in the user study, which we now discuss.

RQ1. How do users interact with the live values? This theme of value-oriented focus versus the “old way” of expression-oriented focus appeared in several of the participant’s interactions with the tool. For example, despite the values featuring prominently in the display, it took until after the entire first exercise for P2 to fully realize they were looking at and working with live *values*. In another scenario, P1 and the facilitator together spent an embarrassingly long time trying to find a bug in an `insert_into_sorted_list` helper. After finding the bug they realized that, had they inspected the live values more closely, they might have found the bug much sooner. Additionally, P2 observed that they are so used to reading trees as long lines of serialized text (*e.g.*, `Node (Leaf 2, Node (Leaf 2, Leaf 3))`) that they were subtly repelled by the beautified 2D rendering of tree values: “I see the splayed-out [rendered] tree and I’m like, ‘Oof, I can’t read this,’ even though it’s much more readable!”

Even so, participants still did use the live display and expressed appreciation for it. P2 also noted that, when working with trees in ordinary programming, if their function didn’t work they would be forced to write large amounts of tree pretty-printing code to perform `printf`-debugging;

⁷In PL, the term “value-oriented” can refer to the programming paradigm centered around immutable values, in contrast to an “imperative” or “object-oriented” (OO) paradigm [96]. Here we use “value-oriented” to refer to a focus on execution results rather than program expressions, regardless of the language paradigm.

the live display ameliorated that issue. (And P2 wished we had introduced trees earlier in the study so they would have had more time to play with them.)

Live values require the user to switch call frames to see other example function calls, or calls that hit a different branch in the code. This was not always natural for participants. In the first session, P2 admitted to sometimes being confused about what branch they were looking at. And, despite gaining moderate proficiency with the tool by the end of the study, P2 still remarked that it was hard to think about how you can flip between frames. How to modify the display to help clarify this operation remains an open question.

RQ2. How do users work non-linearly? We want to know how programmers adapt to MANIPOSYNTH’s non-linear style. The tool requires a number of “inside-out” (P1) changes in thought, such as creating an example before defining a function, providing expressions *without* naming them first, and not worrying about let-binding order but instead just using an out-of-scope variable and letting MANIPOSYNTH move the binding. By the end of the study, participants were familiar with these concepts but did not necessarily start out that way. For example, in the first session P1 had trouble remembering to create functions by first providing an example call, but by the end of the study was doing so without any prompting from the facilitator. P1 also initially had trouble finding particular variable definitions on the screen but felt more comfortable by the second session. Near the end of the second session P2 expressed, “I want a let binding. . . I don’t have any confidence I can make let bindings,” despite having successfully done so many times by double-clicking the subcanvas or dragging values into the subcanvas. P2 instantly understood after a quick reminder from the facilitator, but it is notable that even after around three hours with the tool it hadn’t quite sunk in that most TVs are let-bindings.

At the end of the study, we asked the participants their thoughts about writing expressions without naming them first. P1 expressed they would more likely prefer to instead always have to provide a name; P2 was unsure, but noted that MANIPOSYNTH’s default names had improved from the first version we had them try. In particular, at P2’s behest we hard-coded the default names for list destruction to be `hd: : tail` instead of the original type-based `a2: : a_list2`. Even so, we rediscovered that naming was important in programming. Function skeletons are still inserted with generic parameter names, *e.g.*, `fun x2 x1 -> (??)`, which are both unhelpful and backwards. This indeed resulted in user mistakes during the study, and is a point to improve in future versions of MANIPOSYNTH.

Despite a few troubles, both participants were positive overall about the non-linear workflow. P1 noted the non-linear style “fits a lot more with how I like to write code,” and P2 said, “I like it, I’m excited about it.”

RQ3. How do users interact with program synthesis? We introduced participants to the synthesizer in the second session, at which point the synthesizer lacked a statistics model (instead enumerating terms from small to large) and did not offer the “Accept / Reject / Try again” buttons (instead requiring the user to Undo on an undesired synthesis result); these were added for the final session. We wanted to know how comfortable users were with providing assertions and using the synthesizer.

Participants were familiar with thinking in assertions. In the first session, the facilitator only

introduced participants to providing example function calls, not asserting on their results. Despite this, unprompted, both participants wanted to make assertions once they had an example to work with. When assertions were formally introduced, participants were generally comfortable providing examples, although P1 would occasionally write assertions in a polymorphic form, *e.g.*, `foldl f acc [] = acc`, which would insert new blank bindings for `f` and `acc` on the canvas and P1 would have to recover from the mistake. Even so, P1 appreciated that MANIPOSYNTH encouraged them to write in a test-driven development (TDD) style, and suspected it prevented them from making simple errors. When asked if they had trouble writing assertions, P2 responded, “I had trouble *not* making assertions,” because P2 enjoyed toying with the synthesizer, but P2 did observe that constructing trees was a little tricky. MANIPOSYNTH only beautifies tree values, not tree literal expressions. In the future, MANIPOSYNTH may beautify tree expressions in addition to tree values. Overall, we asked participants to rate how laborious it was to create examples on a scale of 1 to 10, P1 and P2 responded with 2 and 4, respectively. Providing assertions was not a bottleneck.

The facilitator introduced synthesis to the participants with the list length example, which left a positive first impression on the participants. Synthesis was somewhat less helpful after the length example. By the third session, the synthesizer was usually able to finish participants’ functions that were mostly sketched-out (if sketched out correctly!), but occasionally still failed. Even so, participants appreciated the synthesizer when it succeeded and were not bothered when it did not.

A prior study of synthesizer users revealed that users will sometimes accept synthesis results they do not understand, but in that study it did not lead to correctness errors [43]. Our study does provide one counterexample: when P2 invoked synthesis to fill out the final else-branch for BST insert, P2 examined the resulting expression and did not notice that it erroneously duplicated the right subtree; had they written the expression by hand, it is possible they would not have encountered this mistake. It would have been helpful if the synthesizer highlighted the ambiguity at that location, perhaps by finding solutions within a certain probability bound of the most likely solution and leveraging an interface similar to [99] to allow the user to choose between alternative subexpressions.

When initially introduced to the participants, the synthesizer did not have the “Accept / Reject / Try again” buttons, instead participants were required to Undo, but often forgot to do so. Without those buttons, there was also no feedback in MANIPOSYNTH that clearly indicated what had changed—P1 admitted to looking at their Vim window to ascertain what the synthesizer produced. The addition of the “Accept / Reject / Try again” interface was appreciated by participants and P1 noted this did keep their focus more on the MANIPOSYNTH window.

Overall, the facilitator’s impression was that the participants were comfortable trying to use synthesis, but did not necessarily obtain mastery of it, in part because synthesis is opaque. P1 noted, “It is really hard to know whether synthesis is failing because I have posed the problem in an incorrect way or synthesis is failing because I haven’t given it a lot of information. But the process of trying to give it more information is very illuminating in terms of whether my conception of the problem is wrong.” P2 as well initially felt that working with the synthesizer was unfamiliar but remained intrigued by its potential, saying, “It was kind of awkward at first. It sort of seemed like a cool trick but there were parts where it would actually complete the program which was kind

of nice even though it was not like a very trivial program. That’s a neat feature.” These experiences suggest that synthesis in this setting is a viable workflow, despite its initial unfamiliarity.

RQ4. How do users interact with their text editor? Participants were not forbidden from using their text editor, but the heavy focus on learning MANIPOSYNTH meant that they only did so only as a last resort. When asked, P1 estimated they spent about 40% of their time looking at Vim when they were trying to figure out what was going on, but, by the end of the third session, only felt the need to edit in Vim on particularly tricky errors. P2 also felt more comfortable in Vim, “When I was really stuck, I felt self-conscious and I was like, ‘Alright I’ll just figure this out in Vim quickly.’ It’s faster, probably, I’ve got years of experience doing that.”

Part of the promise of bimodal editing is that one *can* do this! Even so, participants performed the vast majority of their editing in the MANIPOSYNTH display. As noted, the participants only occasionally needed to edit in Vim, but even when operating MANIPOSYNTH they did seem to rely on looking at the textual display to understand what was happening. As noted below, MANIPOSYNTH may be over-reliant on shapes and colors to differentiate different kinds of elements and it can be confusing, which may have driven the participants to look at their Vim window instead of relying solely on the MANIPOSYNTH display.

SQ1. What are the pain points? How might the system be improved? The participants had trouble keeping track of what everything was in the MANIPOSYNTH display. MANIPOSYNTH relies on colors and shapes to distinguish the multitude of different UI elements: expressions, values, function parameters, assertions, expected values, return expressions, patterns, let-bindings (TVs), and different (sub)canvases that hold let-bindings. Both participants expressed a desire for more explicit labeling of what all these different elements were. After the first session, we added labels on the various subcanvases (“Top level”, “Bindings inside function”, “Return expression(s) and value(s)”) which P1 expressed appreciation for. We had hoped those would obviate the need for more labeling, but even by the end of the final session the participants still expressed a desire for more indication to explicitly notate element kinds.

SQ2. How comfortable are participants with the tool—can they complete an exercise without guidance? After each exercise we asked participants if they felt comfortable completing the next task without assistance from the facilitator. By the end of the final session P2 was comfortable with minimal assistance, whereas P1 still felt the need for help. Although P1 understood the tool well, they still stumbled over different small issues such as UI corner cases and accidentally trying to edit a parent expression in the subexpression editor (discussed in SQ3 below).

SQ3. Using the lenses of the Cognitive Dimensions of Notations, what additional insights do learn about the tool? The Cognitive Dimensions of Notations [51] is a framework of thirteen lenses for qualitatively assessing design trade-offs. Below, we report a subset of our observations from considering these lenses.

Diffuseness (How noisy is the display?) MANIPOSYNTH stores extra information, such as 2D binding coordinates and previously rejected synthesized expressions, as annotations in the OCaml code. P1 opined that, “All the annotations do make it less attractive to try to do stuff in Vim,” and

these annotations were a source of confusion. The rejected synthesized expressions were particularly confusing because the whole discarded expression was in text in the code, albeit wrapped with `[@not ... ]`, and participants would sometimes read these large expressions without realizing it was not the code they cared about. After the user study, we modified MANIPOSYNTH to store a short hash of the rejected expression rather than a full copy. Additionally, while MANIPOSYNTH includes a syntax highlighting rule that will gray out AST annotations, the rule only works in VS Code with the Highlight extension installed [138].

Secondary Notation (Is there non-semantic notation to convey extra meaning?) MANIPOSYNTH does not currently support comments. P1 missed having comments, while P2 did not.

Viscosity (How hard is it to make changes?) Three main scenarios arose where changes were difficult. First, editing a base case requires that some execution hits the base case, otherwise the base case can never be focused; this was occasionally a hindrance and might be addressed either by adding a “phantom call frame” that focuses the case without a concrete execution or by automatically synthesizing an example that hits the case. Second, once an expression was in the program, it was hard to wrap the existing expression with some new expression; it would be better if there were a mechanism to indicate whether a new drag-and-dropped expression should replace or wrap the old. Finally, (sub)expressions could be text-edited by double-clicking them on the display. However, sometimes participants (and even the first author) would double-click a subexpression but instead want to edit a parent of that expression, which required a different interaction. Future versions of MANIPOSYNTH may, upon double-click, open the entire expression for editing but with the clicked subexpression initially selected out of the whole line of code.

Visibility (Is everything needed visible? Can items be juxtaposed?) Element positioning in MANIPOSYNTH proved tricky, because elements will change size based on the size of the values in the TVs—multiple large trees in the function IO grid, for example, can make a function take up the whole window. Participants did have to move assertions around. P2 used a large screen and expected their functions to grow rightward: P2 would position assertions far to the right of their nascent function. P2 also expressed the desire for snap-to-grid so they could align their TVs perfectly. P1 used a smaller screen which may have caused trouble: at one point P1 was trying to debug and realized after-the-fact that they had scrolled the IO grid offscreen—had it been onscreen and they looked at it, they might have found their mistake quicker. One possible mitigation is to scale down large values.

4.5 Related Work

Before returning to the above observations and their implications for future work, we now discuss a number of systems that share MANIPOSYNTH’s goal of centering the programming workflow around live program values.

Programming by demonstration (PBD) Programming by demonstration (PBD) is an interaction paradigm in which the user demonstrates an algorithm to the computer step-by-step, resulting in a program. The first PBD system, Pygmalion [136], targeted generic programming and, like

MANIPOSYNTH, displayed the live values in scope as the subject of the user’s manipulations. For example, a function call with missing arguments was represented as an icon on the canvas. When all arguments to a function call were supplied, the icon for the function call was replaced with a display of its result value. To use that result value, the user dragged the value to where they wanted to use it. Recursion was supported. Although the 2D canvas was non-linear, Pygmalion treated the program as an imperative, step-by-step movie over time and did not offer a corresponding always-editable text representation.

Like Pygmalion, Pictorial Transformations (PT) [67] also offered program construction via step-by-step manipulation of a display of the live program values. PT allowed the user to customize visualizations, and was in general more expressive than Pygmalion, supporting more complicated algorithms including those involving lists. Later PBD systems were usually more domain-specific [28, 91], although ALVIS Live [70] targeted the construction of iterative array algorithms by demonstration, and, notably, represented the resulting program in editable text. Unlike MANIPOSYNTH, ALVIS Live aimed at imperative code and could not offer non-linear editing—buttons in its UI were needed to allow users to move backwards and forwards in the timeline.

Some empirical evidence for the possible benefits of a value-centric workflow was provided by the Pursuit PBD system for creating shell scripts [106]. In evaluating Pursuit, it was discovered that a comic-strip style representation of a program—with before and after values represented in the frames of the comic-strip—enabled users to more accurately generate programs compared to a more textual representation. On the other hand, when Adam et al. [3] compared student performance between Python Tutor [54], providing editable code + live output, and AlgoTouch [44], providing non-editable code + PBD on values, they found students performed comparably in either environment. An analogous comparison in ALVIS Live also found similar overall student performance when using text or PBD [71]. These results can be interpreted either way: pessimistically, that value-centric manipulation is not clearly better; or optimistically, that despite non-editable code, value-centric editing performs as well as ordinary programming. Even so, a PBD environment may aid in avoiding initial fumbling with syntax and in discovering what a tool can do: Hundhausen et al. [71] found that on the first task, users in the PBD condition completed the task faster and more accurately with less time consulting documentation.

Manipulable live objects In the object-oriented paradigm, the Self [146] language and environment displayed live objects graphically and allowed a user to send messages to those objects via direct manipulation (demonstrated in video form in [140]). Although value-centric, the interactions provided by Self and related systems like the Morphic UI framework [97] differ from all other systems discussed here in that manipulations in Self-like systems modify *state*, not the algorithm.

Like Self, Edward’s “Direct Programming” prototype [36] allowed the user to directly invoke actions on displayed values, but, unlike Self, reified these actions into lines in a script, blurring the line between running a program and modifying it. Also blurring the line between runtime interaction and coding, Boxer [31] was a non-linear programming environment displaying nested boxes on a 2D canvas. A box could contain a comment, code, a value, or serve as a graphics buffer for drawing. Boxes can be written to via code or by user interaction, enabling a workflow that mixes program runtime interaction with program creation. Boxer aimed for its interface to be an

approachable computational medium, resulting in design choices that differ from MANIPOSYNTH. Boxer is not bimodal—the displayed boxes are the program—and state and code are mixed. Also, box results are not automatically rendered. Code boxes must be manually invoked and must write their results to another box, but Boxer includes mechanisms for configuring keys or mouse buttons to trigger particular boxes.

Live nodes-and-wires In 2D nodes-and-wires programming [142], nodes usually represent transformations (expressions) and the wires represent dataflow (values). Consequently, nodes-and-wires environments do not necessarily display live values, although some systems do output live values below the nodes (*e.g.*, `natto.dev` [134]). Among these environments, Enso [39], formerly known as Luna, is also bimodal like MANIPOSYNTH, offering both textual and graphical representations for editing the program.

PANE [66] flips the usual node-and-wires paradigm and instead uses example values for the nodes and locates transformations (expressions) on the wires, placing values more at the center of attention compared to its peers. Examples values can be clicked to invoke operations on them. PANE does not, however, maintain an editable text representation of the program.

Live programming Like MANIPOSYNTH, traditional live programming research seeks to augment ordinary, text-based coding with display of live program values, although the displayed values are read-only. There are a growing number of such systems. Python Tutor [54] is popular teaching tool for visualizing Python program state. Bret Victor’s *Inventing on Principle* presentation [148] demonstrated several live programming environments and served as inspiration for later work [77, 90]. Edwards [35] showed how examples can be incorporated into the IDE for live execution, and Babylonian-style Programming [126] explored how to better manage multiple examples—individual examples could be switched on and off, an interaction we could adopt in MANIPOSYNTH to selectively reduce the number of values shown in the function IO grids.

In-editor PBE/PBD Like MANIPOSYNTH’s programming by examples (PBE) synthesizer, recent work has begun to explore offering PBE and PBD interactions within a traditional, textual programming environment.

Several systems generate code within a computational notebook via manipulations of visualized values. Wrex [34] adapts the FlashFill [52] PBE workflow to Pandas dataframes in Jupyter notebooks—after demonstrating examples of a desired data transformation in a dataframe spreadsheet view, Wrex outputs readable Python code. Similarly, the PBD systems B2 [155], *mage* [155], and Mito [30] transform step-by-step interactions on displayed notebook values into Python code in the notebook. For a Haskell notebook environment, Vital [56, 57] offers copy-paste operations on visualized algebraic data type (ADT) values, which are realized by changing the textual code in the appropriate notebook cell. Like MANIPOSYNTH, graphical interactions in Vital can extract subvalues via pattern matching, although Vital’s workflow focuses on modifying single values in place rather than building up computations like in MANIPOSYNTH. While these notebook systems provide some manipulation of intermediate program values, none offer fine-grained non-linearity as in MANIPOSYNTH.

For a more ordinary IDE setting, CodeHint [46], SnipPy [43], and JDial [68] provide program synthesis interactions in the live context of the user’s incomplete code. With CodeHint, users set a breakpoint in their Java program and describe some property about a value they want—CodeHint will enumerate method calls in the dynamic execution environment at the breakpoint to find a satisfying expression. Like MANIPOSYNTH, CodeHint leverages a statistics model to rank results. Notably, users with CodeHint were significantly faster and more successful at completing given tasks than users without. For Python, SnipPy [43] adapts the Projection Boxes tabular display of runtime program values [90] to perform PBE in the context of live Python values. The authors validated that users were able to successfully use the synthesizer to complete portions of given tasks. JDial [68] records variable values during execution of an imperative Java program and allows the programmer to directly change incorrect values in the execution trace. The program is repaired to match the corrections via sketch-based synthesis [137]. JDial, however, is limited to small program repairs and does not offer program construction features.

Bidirectional, bimodal programming Some systems represent programs as ordinary text, but also allow direct manipulation on program *outputs* to be back-propagated to change the original code. Usually, these changes are “small” changes to literals in the program—such as numbers [25, 84, 100, 45], strings [152, 133, 84, 100], or lists [100]. More full-featured program construction via output manipulations is available in a few systems for programs that output graphics [101, 132], including SKETCH-N-SKETCH (Chapter 2) .

Although MANIPOSYNTH also centers values as subjects for manipulation, we do not yet apply bidirectional techniques to deeply back-propagate a change on a value—direct changes on a value are only allowed when the value was introduced as a literal in the immediately associated expression. An earlier version of MANIPOSYNTH did have limited back-propagation abilities, but we disabled these when we noticed they caused trouble in the user study—manipulation on a value would inconspicuously change a literal in a very different part of the program. Determining an understandable meaning of such direct changes on a value remains an avenue for future work. While the bidirectional programming community offers the “least change” principle [104], that minimal changes should be performed to maintain a constraint, in the context of a full program a change may cause confusion not because of its magnitude but because the item changed is far from the user’s focus. Revealing that far-away code by popping open a “bubble” [16] or “portal” [17] may be one way to help make the change understandable.

4.6 Future Work and Conclusion

How close is MANIPOSYNTH to achieving its goals of providing a value-centric, non-linear programming environment? Based on the examples we implemented and feedback from our study participants, MANIPOSYNTH largely succeeded at providing useful live values. The non-linear features functioned moderately well—users rarely had to think about binding order—but MANIPOSYNTH was not immediately learnable and would benefit from more explicit labeling of the various kinds of elements on the canvas. Through our observations, we hypothesize that expression-oriented and value-oriented modes of thinking are distinct states of mind, and experienced pro-

grammers tend towards the former. An intriguing possibility for future work is to experimentally validate that expression-oriented and value-oriented thinking are actually modes—*i.e.*, the activity of considering values discourages considering expressions, and vice versa. More immediately, there are possible changes to MANIPOSYNTH that might encourage more value-focused interaction.

One experiment we would like to try is to change the display of variable uses so that, instead of the name of the variable, the current value of the variable is shown instead, with the name as a tooltip or subscript. This change might nudge users out of the expression-oriented mode of thinking back towards value-oriented thinking.

An intriguing corollary experiment was requested by P1. To keep track of where values came from, P1 wanted values to be drawn with unique colors all the time, rather than only when the autocomplete options were open. Another possibility is, when the cursor is over a variable usage, to highlight the TV where the variable is defined. We would like to explore these display choices.

Finally, while dragging items onto an expression is quite useful, in the current version of MANIPOSYNTH dragging items onto values is less-so. When working through the examples, the implementer dragged some item onto a value on only 4 occasions; dragging onto an expression happened 209 times. In the future, dragging a value to a value might open a menu of possible ways to combine the values, further increasing the utility of interacting with values. Ideally, programmers should be able to customize the available actions, as in Vital [57] which includes an API for this purpose.

MANIPOSYNTH currently focuses on interactions on relatively small values. Larger data sets might be more conveniently displayed and manipulated in a spreadsheet-style view. Flow-sheets [23] demonstrates how expression outputs might be visualized using a spreadsheet layout, albeit without program synthesis.

In conclusion, building on the insight from Eros [38] that non-linearity complements functional programming, MANIPOSYNTH showed that non-linearity can be maintained even when the program is ordinary code. Our focus on supporting textual code has resulted in the current MANIPOSYNTH being somewhat more expression-centric than Eros, but the above are possible ways MANIPOSYNTH might become more value-centric. Our overall vision is to make programming feel like a tangible process of molding and forming. MANIPOSYNTH points a way forward to that goal.

Chapter 5

Conclusion

In the prior chapters we explored the expressive power of output-based interactions within bimodal programming environments. To conclude, we reflect on overall lessons learned (Section 5.1) and consider future directions for bimodal programming systems (Section 5.2) before briefly summarizing the work (Section 5.3).

5.1 Lessons Learned

Several themes emerged throughout the implementation and evaluation of SKETCH-N-SKETCH, TSE, and MANIPOSYNTH. At the implementation level, all three systems leveraged an interpreter with dynamic provenance tracing and all three utilized information from a static type system. From the user perspective, our initial experience with the systems suggest takeaways: labeling on the canvas display is important for understandability, and a bimodal interface with editable code does work to provide explanation and flexibility. These observations lead us to offer the below recommendations for future implementers of bimodal programming systems.

5.1.1 Rely on Dynamic Provenance Tracing

SKETCH-N-SKETCH, TSE, and MANIPOSYNTH all run the user’s code through an interpreter that dynamically logs a trace of program execution steps. The logged information forms the basis for the output-based interactions: the trace correlates output values to program locations. The three systems show that this basic approach—dynamic provenance tracing—is an effective means for facilitating bimodal programming.

In contrast to dynamic tracing in an interpreter, rewriting approaches to provenance are possible via source-to-source translation. With a rewriting approach, the code can be run through an existing language compiler or interpreter at a possible speed benefit compared to a tracing interpreter. For example, Transmorphic [132] programs run in ordinary ClojureScript. Rewriting approaches, however, can complicate the association of output values to source code locations, *e.g.*, Transmorphic requires users to manually write code to add IDs to shapes produced in loops. Polymorphism also complicates the story. Because the specific concrete type of the value at a polymorphic location is unknown, any logging added there by source-to-source translation cannot log anything of consequence about the value—neither its structure or concrete type—so long as the language performs type erasure, as does OCaml. Monomorphization [145]—duplicating and specializing the polymorphic function for each set of argument types the function is called with—can dodge this trouble, and an early prototype of MANIPOSYNTH did use monomorphization to log runtime values via source-to-source translation instead of using an interpreter. Tracing via rewriting can

work fine in a language with better value reflection capabilities, *e.g.*, Java [124]. But if the implementer hopes that, somehow, source-to-source translation might offer a simpler route compared to writing an interpreter, the required monomorphization is an important consideration. Our experience has been that, once there is a functioning interpreter, modifying the interpreter to log tracing information is quite straightforward. Unless there is a need to operate under existing tooling, or there is some need for speed of the executed code, we advise against relying on source-to-source translation.

Although our experience with creating bimodal systems leads us to suggest relying on dynamic tracing via an interpreter, our experience does not yet suggest a standard form for the traces. SKETCH-N-SKETCH, TSE, and MANIPOSYNTH all use different—and sometimes contradictory—tracing schemes. For example, TSE marks a case split result as dependent on the scrutinee (EVALCASE in Figure 3.9), whereas SKETCH-N-SKETCH effectively does not (ITE, fst, and snd in Figure 2.15). TSE needs to know where to offer “Change Constructor” actions (a modification of the scrutinee) whereas SKETCH-N-SKETCH is concerned with selecting a shape’s sub-elements (which should *not* be conflated with the whole shape itself, even if deconstructed from the shape). Our experience is that it is difficult to anticipate in advance what specific kinds of tracing will be needed for the desired scenarios. The entire operational derivation tree could be considered the most thorough form of provenance tracing [22], so familiarity with a rich provenance scheme like Transparent ML [2] is a good place to start thinking about provenance tracing. TML traces preserve most of the derivation tree, and simpler forms of provenance can be projected out of them [2]. But again, anticipating the exact forms of provenance that a bimodal environment will need is difficult in advance, so we recommend implementers start with a tracing interpreter for its flexibility and only consider optimizations after their provenance needs are clarified.

5.1.2 Leverage Types

The goal of programming environment research is to leverage all the information that can be gleaned about a program to offer the most intelligent interface for editing that program. It is no surprise, then, that type information can play a key role. All three systems in this work leverage static type information to support the bimodal interface.

Type inference in SKETCH-N-SKETCH is used to expose ordinary user-defined functions as drawing tools. Brands [101] distinguish which arguments are, *e.g.*, coordinates or distances, so that the a new shape’s (*i.e.*, a new function call’s) position and size are matched to the user’s mouse movements (§2.4.7, §2.4.8).

In TINY STRUCTURE EDITORS, types are part and parcel of the interface: TSE generates editors for the values of particular algebraic data types. The types dictate which transformations are possible. Without types, TSE could not provide meaningful data transformation actions.

In addition to guiding synthesis, MANIPOSYNTH uses static types to offer default example values that the user can easily add to their program via autocomplete (Figure 4.3) or the toolbar (Figure 4.6). Also, OCaml’s type inference obviates the need to provide static type annotations while constructing programs, freeing the user to concentrate on the values they see. In the future, MANIPOSYNTH may also use types to refine the autocomplete options offered.

Overall, we recommend bimodal environment implementers leverage languages with robust

static type systems with strong inference capabilities. While live environments like Self [146] show how *dynamically* interrogating objects can provide a UI for invoking operations on those objects, only a static type system enables the bimodal environment to confidently answer the questions “What is this?” and “What can go here?” for all program locations.

5.1.3 Use Labels

With the visualization of runtime program values *and* a plethora of direct manipulation tools for operating on those values, clear labeling in the UI is key for usability in bimodal programming systems, both for associating output with its code and for explaining the user interface.

To assist in associating output with its code, SKETCH-N-SKETCH draws variable names next to associated items in the output. To avoid clutter, these names only appear when the cursor is over the item. These labels also allow items in the output to be renamed, which we found was instrumental to producing final programs that looked like they could have been written manually by a human (as in the Logo and Koch Snowflake examples, Figure 2.5 and Figure 2.7). Additionally, when the user hovers their cursor over an item in the output, SKETCH-N-SKETCH will highlight an appropriate expression in the program (based on an estimate of what would be deleted were the user to DELETE the output item). Together, the labels and expression highlighting help the user to associate appropriate code locations with items in the output.

MANIPOSYNTH similarly labels items on its canvas with their full expressions and patterns. The labeling is thorough enough that MANIPOSYNTH can be used with the text editor hidden—unlike SKETCH-N-SKETCH. But because MANIPOSYNTH is not as fully integrated with a text editor, code expressions in the editor are not highlighted when the cursor on the canvas.

Labels may also help explain the user interface. While we added some labeling based on user feedback, labels explaining the MANIPOSYNTH interface are still minimal, which may have contributed to user confusion. Interface explanations are perhaps less needed in SKETCH-N-SKETCH, as its interface is similar to an ordinary graphics editor. Even so, the entire suite of SKETCH-N-SKETCH transformations is never enumerated in the interface—only once items are selected are relevant transforms shown. A complete listing of available transformations could help users discover which transforms are available (a hypothesis supported by the refactoring interface study in Appendix A). In addition to interface labels, both MANIPOSYNTH and SKETCH-N-SKETCH might benefit from more tooltips explaining items.

Because bimodal systems are new and unfamiliar to users, we recommend bimodal environment implementers pay special attention to labeling. Labeling helps users by disclosing possible actions the user may perform and shows the correspondence between expression in the code and items on the canvas.

5.1.4 Embrace Code

The primary goal of this work is to expand the possibilities of output-based interaction in a bimodal programming environment. Although we have not performed extensive user testing, the example programs we implemented do showcase capabilities and promise of bimodal programming. From these initial experiences we offer several additional suggestions for implementers of future bimodal environments.

You can make your interface bimodal Although building an extensive toolset like that offered in SKETCH-N-SKETCH is quite an undertaking, adding just a little bimodal interaction is not as hard as one might expect. A little tracing can go a long way. Once there is any association between output and code locations, it is straightforward to offer simple manipulations of the code from the output canvas. The manipulations could be as simple as revealing a text box to edit the associated code (as in MANIPOSYNTH), or dragging an item around to change the associated numeric literal in the code. Many systems have demonstrated output interactions to enact these kinds of “small” code changes [45, 25, 84, 78, 152, 133, 100]. Traces could be as simple as tagging all values with the single expression ID at which a value was introduced: even this simple scheme would allow the user to manipulate literals that pass through to the output.

Aim for human-like code An ideal bimodal environment will produce code that looks as if a human could have written it. Towards this aim, two important considerations are choosing variable names and formatting whitespace.

Good names are key for generating comprehensible and human-like code. SKETCH-N-SKETCH uses a multi-step algorithm for choosing quality default names (§2.4.6). Statistical methods for choosing names are also possible [4, 5, 127, 74]. MANIPOSYNTH’s naming algorithm is simpler. A name for an expression is usually chosen based on the first two “words” of the expression’s code: *e.g.*, `fold f [] list` will be named `foldF`. Though simple, this scheme usually produces more informative names than just using the type of the expression. Unsurprisingly, we discovered that uninformative names such as `x1`, `x2`, etc. should be avoided. The one scenario where MANIPOSYNTH generates such names—new function skeletons—did indeed cause user confusion. At a minimum, renaming variables should be easy. Rename refactorings can be invoked from both the SKETCH-N-SKETCH and MANIPOSYNTH canvases.

In the SKETCH-N-SKETCH implementation, we paid careful attention to intelligently handling whitespace and comments in the program. Each expression in SKETCH-N-SKETCH stores its leading whitespace; and comments are stored as AST nodes. Each transformation tool is responsible for preserving and inserting whitespace appropriately. Although this carefulness helps preserve the “naturalness” of the user’s program, it came at some cost to the implementer. To speed development, and because the OCaml AST does not explicitly store whitespace, for MANIPOSYNTH we opted to ignore whitespace and instead rely on the whitespace choices made by `ocamlformat`¹. Relying on `ocamlformat` has external validity: automatic code formatting is sometimes part of professional development workflows. The main weakness of MANIPOSYNTH code handling is that comments are erased—OCaml’s parser discards them—which is a limitation that must be remedied if MANIPOSYNTH is to be used for practical work.

Do not be afraid to store state in the code While more transient state—such as the currently selected items—can be handled in-memory in the browser or with HTML local storage², both MANIPOSYNTH and SKETCH-N-SKETCH choose to persist certain state in the user-visible code. SKETCH-N-SKETCH stores the currently focused expression using special comments (§2.4.9), and

¹<https://github.com/ocaml-ppx/ocamlformat>

²<https://html.spec.whatwg.org/multipage/webstorage.html>

MANIPOSYNTH stores 2D positions for tangible values as AST attribute annotations. Although such annotations increase the noisiness of the code, they prevent the code from becoming out of sync were the state to be stored in a separate file. And because inline code annotations tag expressions directly, there is no need for the annotations to use expression identifiers, which may not be preserved across text edits.

Embrace—don’t replace—code While both SKETCH-N-SKETCH and MANIPOSYNTH demonstrated what is possible via edits performed entirely on the canvas, our experience does not suggest that code can be abolished. In SKETCH-N-SKETCH, the expressivity of output-based manipulation was only accomplished by implementing a large number of tools (Table 2.1), a multi-year undertaking. And even though so much of the program can be constructed on the canvas, the canvas display does not explain the computation clearly enough to hide the code. In MANIPOSYNTH, we can hide the text editor only because almost all the code is directly represented on the canvas. These observations point to the original motivation of bimodal programming, which we continue to endorse: because textual code is familiar, time-proven, and powerful, it may be better to augment code rather than replace it.

5.2 Future Work

Although SKETCH-N-SKETCH, TSE, and MANIPOSYNTH demonstrate the expressive power of bimodal programming, none of the systems are ready for practical use. A number of improvements are possible.

Scaling up The systems in the work have only been run on small programs. Running larger programs requires managing traces efficiently, and handling practical programs requires managing side effects such as input and output. The tracing performed by SKETCH-N-SKETCH and MANIPOSYNTH consumes a considerable amount of memory: every execution step produces a new portion of the trace. Currently, these traces are all stored in RAM. Future tracing could instead write to persistent storage. And for a large program, only a small portion of the trace will likely be needed at a time. Tracing can be skipped for code outside the region of interest. When needed, missing portions of the trace could be rebuilt on demand via program replay, which can be accomplished efficiently by periodically dumping the program state during the initial execution and replaying from a checkpoint as needed [14]. With careful logging of system calls, any program input/output can also be recorded so that replay is deterministic [113].

Managing visual space Effectively using the limited space on the screen is always a concern in graphical programming, and the story is no different for bimodal environments. A small expression in the program, say a variable usage x , might resolve to a large value, such as a deep tree, that must be displayed on the canvas. The live values for a small program fragment can quickly overwhelm the canvas (*e.g.*, Figure 5.1). For large structures like trees or long strings, one possibility is to hide uninteresting portions of the data structure or to replace those portions by a variable name representing the hidden portion [102, 33]. Standard techniques such as scaling, zooming, and

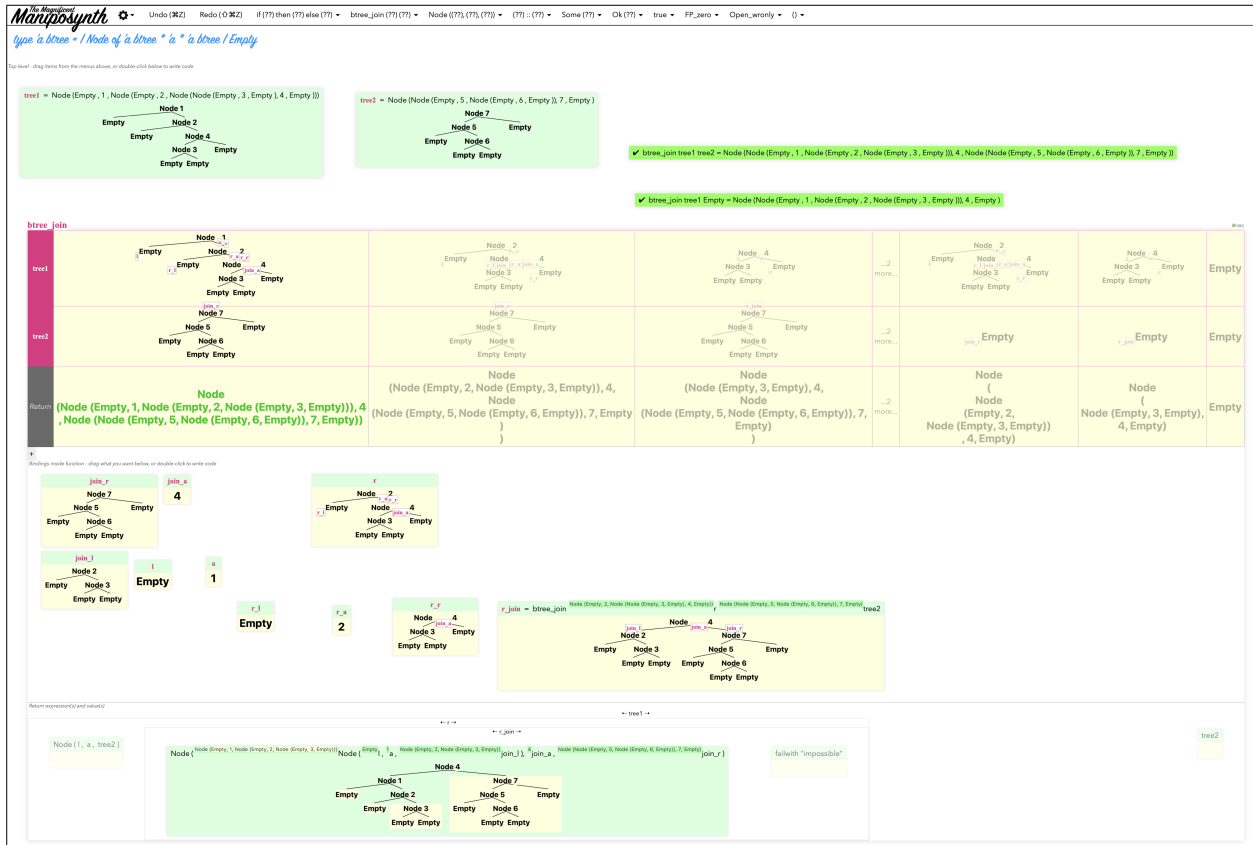


Figure 5.1: The MANIPOSYNTH interface at the end of the `btree_join` exercise. Live values fill up the space on the screen.

selectively toggling element visibility (akin to code folding in IDEs) are other strategies that may be employed to better manage visual space on the canvas.

Moving canvas and code closer together MANIPOSYNTH and TSE present the code and the output in two separate panes, with a visible division between the two. Partly this was a disciplined choice by the author as a way to limit text-oriented thinking from “contaminating” the creative process of working out how bimodal programming can operate. But now that these systems have been created and have demonstrated the expressive extent of output-based interactions, it may be time to consider dissolving the hard separation between the code and output displays. The dividing line could be removed and the live values could be drawn near the relevant code expressions, perhaps even inline with the code (as in [139] and [65], or as is possible when debugging in the Chrome DevTools³ or VS Code⁴). Moving the output next to its code has several advantages. It becomes easier for the user to associate code with output, and they do not have to move their eyes as

³<https://developers.google.com/web/updates/2015/07/preview-javascript-values-inline-while-debugging>

⁴<https://www.youtube.com/watch?v=nWW09971u6I&t=63s>

far. Additionally, it could encourage practical adoption of bimodal programming: programmers are used to working in text buffers, placing live values in the text buffer may feel more like a augmentation of their ordinary workflow rather than an entirely new paradigm. Finally, placing code and output together in a bimodal interface opens opportunities for direct manipulation not only of values but also of expressions. (Appendix A presents an interface for direct manipulation of expressions.)

Customization For bimodal programming to reach its goal of the user *feeling* like they are tangibly touching and molding the values in their program, the user will need access to both domain-specific visualizations and domain-specific interfaces for values. Graphs should be editable as graphs, trees as trees, images as images, sets as sets, etc. A fully realized bimodal programming system will need to support custom visualizations and custom user actions on those displayed values. TSE offers a low effort way to author custom interactive textual representations, but custom graphical representations, as well as custom actions on those representations, are not implemented in this work. One intriguing future possibility is to allow users to craft custom visualizations using a SKETCH-N-SKETCH-style interface rather than by manual coding. Ideally, these custom visualizations could be composed together by nesting. Livelits [117] shows how to nest different custom interfaces within each other based on type compatibility, although in the program the Livelits can only be used as expressions (as in other visual macro systems, *e.g.*, [6]). There may be a way to evolve this approach to apply to values as well.

User validation This work primarily focused on demonstrating the expressivity of bimodal programming, leaving much to learn about how users interact with such systems. In particular, there are open questions about how bimodal programming might assist novices, about how non-linearity might affect the programming workflow, and about the possible dichotomy between expression-oriented and value-oriented modes of thinking.

Part of the vision of bimodal programming is that it might help novices more rapidly create useful programs. The (few) studies to date do not find a large benefit to bimodal interaction in a learning setting, nor do they find a detriment. In the ALVIS Live bimodal editor for creating array iterative algorithms, Hundhausen et al. [71] found that students taught exclusively to use direct manipulation tools (with code visible) were able to transfer their learning when asked instead to use only text editing—these students performed similarly to those taught to exclusively use text editing. As well, Do et al. [32] found no clear differences between students’ textual coding performance whether or not direct manipulation interactions were available during their system’s guided tutorial. Likewise, although comparing two different environments (Python Tutor [54] and AlgoTouch [44]), Adam et al. [3] found that students performed comparably using only text editing versus using only PBD. All these results do not necessarily mean there is no benefit to offering direct manipulation interactions. Hypothetically, direct manipulation can help novices in two related ways: (a) it might reduce fumbling with unfamiliar textual syntax and (b) “drawing tools” self-document what actions are available. Hundhausen et al. [71] found evidence for both of these hypotheses: on their very first task, users with direct manipulation tools completed the task faster and more accurately, and appeared to spend considerably less time consulting documentation in the process. These preliminary studies indicate that bimodal programming is neither a silver

bullet for teaching novices, nor is it a hindrance. How best to leverage direct manipulation to assist learners remains an open question, as is the question of whether direct manipulation can speed up program development once users are familiar with the language and environment.

MANIPOSYNTH presented an initial exploration of non-linear editing in a bimodal environment. While the two users were positive about non-linear editing, and binding order was rarely troublesome, we have not performed an explicit user comparison to a more linearly constrained environment to see if, *e.g.*, different users prefer one environment over the other, or how much trouble it is when there are limits on variable shadowing, or if non-linear editing can promote exploratory problem solving strategies (using the canvas as a scratchpad). These are all relevant questions to investigate.

Finally, from the experiences with MANIPOSYNTH we hypothesize that when experienced programmer use a bimodal interface, they intermittently switch between two modes of thinking: either focusing on expressions or focusing on values, to the exclusion of attention on the other. How to encourage programmers to avoid ignoring the displayed values, whether these two modes of thinking are indeed distinct and mutually exclusive, and what the implications of that might be for the programming experience are all open questions to address in future work.

5.3 Summary

Bimodal programming augments conventional text-based programming, allowing users to directly manipulate the output of the program in order to construct and modify their code.

Despite its promise, the bimodal paradigm is under-explored. In this work we seek to explore and expand the expressive scope of output-based interaction in bimodal environments by building and demonstrating two bimodal systems and one supporting mechanism.

SKETCH-N-SKETCH is a development environment for creating programs that output vector graphics. With SKETCH-N-SKETCH it is possible to create parametric designs without ordinary text editing—although ordinary text editing is always available. The ability to create these programs without text editing is made possible by a large suite of output-based tools. We described the design and implementation of these tools and created sixteen parametric designs without the use of textual coding.

In the long term, bimodal programming environments will need to support custom visualizations of data types that, despite being custom, can still be directly manipulated to change the user’s code. To provide a low-cost mechanism for users to create custom manipulable visualization for their program’s data types, we created TINY STRUCTURE EDITORS (TSE), a mechanism for automatically generating structure editors for a custom data value, given only a `toString` function for that type. By tracing the execution of the `toString` function, TSE overlays selection regions and edit buttons on top of the `toString` output which allow the user to select and manipulate corresponding parts of the original value. We described the tracing mechanism and briefly explored the editors generated by TSE for several existing `toString` functions translated from Haskell and for a handful of `toString` written from scratch.

Finally, to expand beyond shape-drawing programs and employ the bimodal paradigm in a general-purpose setting, we built MANIPOSYNTH, a bimodal programming environment for creat-

ing functional data structure algorithms in OCaml. Because side-effect free functional programs do not require a concept of time, we utilized the 2D canvas to present the user’s program non-linearly, allowing the user some flexibility in the order in which they choose to build their code. As well, the live example-centric environment is a fitting setting for program synthesis, so we designed and incorporated a program synthesizer into the MANIPOSYNTH interface. We implemented 38 example exercises and also trained two professional programmers in the operation of the system to solicit their feedback. We discovered that, although bimodal programming remains promising, professionals often habitually focus their attention on program expressions rather than on the live program values. Attending to expressions versus attending to values may even be two distinct modes of thought—more research on this hypothesis is needed.

In total, these software systems justify the thesis:

Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-based interactions in a bimodal programming environment.

Having demonstrated the expressive extent of bimodal programming, future work may merge and expand these systems, as well as perform user evaluations to identify the tradeoffs in the bimodal design space. These efforts will serve the overall goal of making programming feel like an approachable and tangible process of manipulating live values.

References

- [1] Live Programming Workshop (LIVE). <https://liveprog.org/>.
- [2] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A Core Calculus for Provenance. *Journal of Computer Security*, 2013.
- [3] Michel Adam, Moncef Daoud, and Patrice Frison. Direct Manipulation versus Text-based Programming: An Experiment Report. In *Conference on Innovation and Technology in Computer Science Education (ITiCS)*, 2019.
- [4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning Natural Coding Conventions. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014.
- [5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting Accurate Method and Class Names. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.
- [6] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. Adding Interactive Visual Syntax To Textual Code. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2020. URL <https://doi.org/10.1145/3428290>.
- [7] Rolf Bahlke and Gregor Snelting. Context-Sensitive Editing with PSG Environments. In *Proceedings of the International Workshop on Advanced Programming Environments*, pages 26–38. Springer, 1986.
- [8] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software*, 2015.
- [9] Marianne Baudinet and David MacQueen. Tree Pattern Matching for ML. 1985.
- [10] Michel Beaudouin-Lafon and Wendy E. Mackay. Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Working Conference on Advanced Visual Interfaces (AVI)*, 2000.
- [11] Andrew Begel. LogoBlocks: A Graphical Programming Language for Interacting with the World, 1996. Advanced Undergraduate Project, MIT Media Lab.
- [12] Ben Fry and Casey Reas. Processing. <https://processing.org/>.
- [13] Gilbert Louis Bernstein and Wilmot Li. Lillicon: Using Transient Widgets to Create Scale Variations of Icons. *Transactions on Graphics (TOG)*, 2015.
- [14] Bob Boothe. Efficient Algorithms for Bidirectional Debugging. In *Conference on Programming Language Design and Implementation (PLDI)*, 2000. URL <https://doi.org/10.1145/349299.349339>.

- [15] Alan Borning. The Programming Language Aspects of ThingLab. *Transactions on Programming Languages and Systems (TOPLAS)*, October 1981.
- [16] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Conference on Human Factors in Computing Systems (CHI)*, 2010. URL <https://doi.org/10.1145/1753326.1753706>.
- [17] Alexander Breckel and Matthias Tichy. Embedding Programming Context into Source Code. In *International Conference on Program Comprehension (ICPC)*, 2016. URL <https://doi.org/10.1109/ICPC.2016.7503732>.
- [18] Neil Christopher Charles Brown, Amjad Altadmri, and Michael Kölling. Frame-Based Editing: Combining the Best of Blocks and Text Programming. In *Conference on Learning and Teaching in Computing and Engineering (LaTiCE)*, 2016.
- [19] Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It’s Alive! Continuous Feedback in UI Programming. In *Conference on Programming Language Design and Implementation (PLDI)*, 2013.
- [20] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping Distributed Hierarchical Web Data. In *Symposium on User Interface Software and Technology (UIST)*, 2018.
- [21] Salman Cheema, Sumit Gulwani, and Joseph LaViola. QuickDraw: Improving Drawing Experience for Geometric Diagrams. In *Conference on Human Factors in Computing Systems (CHI)*, 2012.
- [22] James Cheney, Umut A. Acar, and Roly Perera. Toward a Theory of Self-explaining Computation. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, 2013.
- [23] Glen Chiacchieri. F-F-F-Flowsheets v2 🍰🍰🍰 demo. <https://www.youtube.com/watch?v=y1Ca5cz0Y7Q>, 2017.
- [24] Chris Granger. Light table kickstarter. <https://www.kickstarter.com/projects/ibdknox/light-table>, 2012-2014.
- [25] 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.
- [26] Stephen Cooper, Wanda Dann, Randy Pausch, and Randy Pausch. Alice- A 3-D Tool for Introductory Programming Concepts. In *Journal of Computing Sciences in Colleges*, 2000.
- [27] Naëla Courant, Julien Lepiller, and Gabriel Scherer. camlboot, 2020. URL <https://github.com/Ekdohibs/camlboot/>.

- [28] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [29] Daan Leijen. Data.Set. In *Haskell Containers Library*, 2002. <https://hackage.haskell.org/package/containers-0.6.2.1/docs/src/Data.Set.html>.
- [30] Jacob Diamond-Reivich. Mito: Edit a Spreadsheet. Generate Production Ready Python. In *LIVE Workshop*, 2020. URL <https://liveprog.org/live-2020/Mito-Edit-a-Spreadsheet-Generate-Production-Ready-Python/>.
- [31] Andrea A. diSessa and Harold Abelson. Boxer: A Reconstructible Computational Medium. *Communications of the ACM (CACM)*, 1986.
- [32] Quan Do, Kiersten Campbell, Emmie Hine, Dzung Pham, Alex Taylor, Iris Howley, and Daniel W Barowy. Evaluating ProDirect Manipulation in Hour of Code. In *SPLASH-E Symposium*, 2019. URL <https://doi.org/10.1145/3358711.3361623>.
- [33] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. Synthesis with Abstract Examples. In *International Conference on Computer Aided Verification (CAV)*, 2017. URL https://doi.org/10.1007/978-3-319-63387-9_13.
- [34] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. *Conference on Human Factors in Computing Systems (CHI)*, 2020.
- [35] Jonathan Edwards. Example Centric Programming. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2004. URL <https://doi.org/10.1145/1028664.1028713>.
- [36] Jonathan Edwards. Direct programming. 2018. URL <https://vimeo.com/274771188>.
- [37] Jonathan Edwards, Jodie Chen, and Alessandro Warth. Live End-User Programming. In *LIVE Workshop*, 2016.
- [38] Conal Elliott. Tangible Functional Programming. In *International Conference on Functional Programming (ICFP)*, 2007. <http://conal.net/papers/Eros/>.
- [39] Enso. Enso. URL <https://enso.org/>.
- [40] Evan Czaplicki. Elm. <http://elm-lang.org>, 2012-2020.
- [41] Johan Fabry. The Meager Validation of Live Programming. In *International Conference on Art, Science, and Engineering of Programming*, 2019.
- [42] Tim Felgentreff, Alan Borning, Robert Hirschfeld, Jens Lincke, Yoshiki Ohshima, Bert Freudenberg, and Robert Krahn. Babelsberg/JS - A Browser-Based Implementation of An Object Constraint Language. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

- [43] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-Step Live Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2020.
- [44] Patrice Frison. A Teaching Assistant for Algorithm Construction. In *Conference on Innovation and Technology in Computer Science Education (ITiCS)*, 2015.
- [45] Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. CapStudio: An Interactive Screencast for Visual Application Development. In *Conference on Human Factors in Computing Systems (CHI), Extended Abstracts*, 2014.
- [46] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Code-Hint: Dynamic and Interactive Synthesis of Code Snippets. In *International Conference on Software Engineering (ICSE)*, 2014.
- [47] Andrew Gelman and Jennifer Hill. Data Analysis Using Regression and Multilevel/Hierarchical Models. 2007.
- [48] GHC Team. Glasgow Haskell Compiler 8.6.5, 2019. <https://gitlab.haskell.org/ghc/ghc/tree/ghc-8.6.5-release>.
- [49] Michael Gleicher and Andrew Witkin. Drawing with Constraints. *The Visual Computer: International Journal of Computer Graphics*, 1994.
- [50] Chris Granger. Light Table—A New IDE Concept. 2012. <http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/>.
- [51] Thomas R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Comput.*, 7(2):131–174, 1996. doi: 10.1006/jvlc.1996.0009. URL <https://doi.org/10.1006/jvlc.1996.0009>.
- [52] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*, 2011.
- [53] Philip Guo. Ten Million Users and Ten Years Later: Python Tutor’s Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *Symposium on User Interface Software and Technology (UIST)*, 2021. URL <https://doi.org/10.1145/3472749.3474819>.
- [54] Philip J. Guo. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2013.
- [55] Christopher Michael Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.
- [56] Keith Hanna. Interactive Visual Functional Programming. In *International Conference on Functional Programming (ICFP)*, 2002.

- [57] Keith Hanna. A Document-Centered Environment for Haskell. In *International Workshop on Implementation and Application of Functional Languages (IFL)*, 2005.
- [58] Anthony C. Hearn. REDUCE: A User-Oriented Interactive System for Algebraic Simplification. In *Interactive Systems for Experimental Applied Mathematics*. Academic Press, 1968.
- [59] Brian Hempel and Ravi Chugh. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*, 2016.
- [60] Brian Hempel and Ravi Chugh. Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions). In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020.
- [61] Brian Hempel and Ravi Chugh. Tiny Structure Editors for Low, Low Prices! Technical Supplement. 2020.
- [62] 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.
- [63] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Symposium on User Interface Software and Technology (UIST)*, 2019.
- [64] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor. In *Technical Report 131a, Digital Systems Research, Digital Equipment Corporation*, 1994.
- [65] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting Code with In Situ Visualizations To Aid Program Understanding. In *Conference on Human Factors in Computing Systems (CHI)*, 2018. URL <https://doi.org/10.1145/3173574.3174106>.
- [66] Joshua Horowitz. PANE: Programming with Visible Data. In *LIVE Workshop*, 2018. <http://joshuahhh.com/projects/pane/>.
- [67] Yen-Teh Hsia and Allen L. Ambler. Programming Through Pictorial Transformations. In *International Conference on Computer Languages*, 1988.
- [68] Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D’Antoni. Direct Manipulation for Imperative Programs. In *Static Analysis Symposium (SAS)*, 2019.
- [69] Ruanqianqian Lisa Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2022.
- [70] Christopher D. Hundhausen and Jonathan Lee Brown. What You See Is What You Code: A live Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, 2007. doi: 10.1016/j.jvlc.2006.03.002.

- [71] Christopher D. Hundhausen, Sean Farley, and Jonathan Lee Brown. Can Direct Manipulation Lower the Barriers To Computer Programming and Promote Transfer of Training? An Experimental Study. *ACM Trans. Comput. Hum. Interact.*, 16(3):13:1–13:40, 2009. doi: 10.1145/1592440.1592442. URL <https://doi.org/10.1145/1592440.1592442>.
- [72] R. Nicholas Jackiw and William F. Finzer. The Geometer’s Sketchpad: Programming by Geometry. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [73] Jennifer Jacobs, Sumit Gogia, Radomír Mech, and Joel R. Brandt. Supporting Expressive Procedural Art Creation Through Direct Manipulation. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017.*, 2017.
- [74] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful Variable Names for Decompiled Code: a Machine Translation Approach. In *International Conference on Program Comprehension (ICPC)*, 2018.
- [75] Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. Guiding Dynamic Programing Via Structural Probability for Accelerating Programming by Example. *Proc. ACM Program. Lang.*, 4(OOPSLA):224:1–224:29, 2020. doi: 10.1145/3428292. URL <https://doi.org/10.1145/3428292>.
- [76] Hyeonsu Kang and Philip J. Guo. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Symposium on User Interface Software and Technology (UIST)*, 2017.
- [77] Saketh Kasibatla and Alex Warth. Seymour: Live Programming for the Classroom. In *LIVE Workshop*, 2017.
- [78] Matthew Keeter. libfive Studio, 2017. <https://libfive.com/studio/>.
- [79] Andrew John Kennedy. Programming Languages and Dimensions. Technical report, University of Cambridge, Computer Laboratory, 1996. URL <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf>.
- [80] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid Moves Between Code and Graphical Work In Computational Notebooks. In *Symposium on User Interface Software and Technology (UIST)*, 2020.
- [81] Amy J. Ko and Brad A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI)*, 2006.
- [82] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. How Live Coding Affects Developers’ Coding Behavior. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014. URL <https://doi.org/10.1109/VLHCC.2014.6883013>.

- [83] David Kurlander. *Graphical Editing by Example*. PhD thesis, Columbia University, 1993.
- [84] Kevin Kwok and Guillermo Webster. Carbide Alpha, 2016. <https://alpha.trycarbide.com/>.
- [85] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
- [86] Tessa Lau. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine*, 2009. URL <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262>.
- [87] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering (ISSE)*, 2007.
- [88] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating Search-based Program Synthesis using Learned Probabilistic Models. In *Conference on Programming Language Design and Implementation (PLDI)*, 2018.
- [89] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *International Conference on Software Engineering (ICSE)*, 2013.
- [90] Sorin Lerner. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. *Conference on Human Factors in Computing Systems (CHI)*, 2020.
- [91] H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc., 2001.
- [92] Henry Lieberman. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [93] Henry Lieberman. Tinker: A Programming By Demonstration System for Beginning Programmers. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [94] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes On the Web. In *Conference on Human Factors in Computing Systems (CHI)*, 2007.
- [95] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020. doi: 10.1145/3408991. URL <https://doi.org/10.1145/3408991>.
- [96] Bruce J. MacLennan. Values and Objects In Programming Languages. *ACM SIGPLAN Notices*, 17(12):70–79, 1982. doi: 10.1145/988164.988172. URL <https://doi.org/10.1145/988164.988172>.

- [97] John H. Maloney and Randall B. Smith. Directness and Liveness In the Morphic User Interface Construction Environment. In *Symposium on User Interface Software and Technology (UIST)*, 1995.
- [98] David L. Maullsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying Graphical Procedures by Example. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1989.
- [99] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. User Interaction Models for Disambiguation In Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2015.
- [100] Mikaël Mayer, Viktor Kunčák, and Ravi Chugh. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL)*, Issue OOPSLA, 2018.
- [101] Sean McDirmid. A Live Programming Experience. In *Future Programming Workshop, Strange Loop*, 2015. <https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcdryTyPiuW8>
<https://www.youtube.com/watch?v=YLrdhFEAiQo>.
- [102] Sean McDirmid. Tangible Abstraction. *SPLASH-I*, 2018. A video of the system is at https://www.youtube.com/watch?v=6VmA_whVxPc.
- [103] McDirmid, Sean. The Future of Programming will be Live. In *Curry On*, 2016. <https://www.youtube.com/watch?v=bnqkg1rSqrq>.
- [104] Lambert Meertens. Designing Constraint Maintainers for User Interaction, 1998. URL <https://www.kestrel.edu/people/meertens/pub/dcm.pdf>.
- [105] Robin Milner. A Theory of Type Polymorphism In Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4).
- [106] Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical Representation of Programs In a Demonstrational Visual Shell - An Empirical Evaluation. *ACM Trans. Comput. Hum. Interact.*, 4(3):276–308, 1997. doi: 10.1145/264645.264659. URL <https://doi.org/10.1145/264645.264659>.
- [107] Emerson Murphy-Hill. *Programmer Friendly Refactoring Tools*. PhD thesis, Portland State University, 2009.
- [108] Emerson Murphy-Hill and Andrew P. Black. Breaking the Barriers to Successful Refactoring. In *International Conference on Software Engineering (ICSE)*, 2008.
- [109] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. In *International Conference on Software Engineering (ICSE)*, 2009.

- [110] National Instruments. Labview. <https://www.ni.com/en-us/shop/labview.html>.
- [111] Greg Nelson. Juno, A Constraint-Based Graphics System. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1985.
- [112] Tobias Nipkow and Mohammad Abdulaziz. Functional Data Structures (in2347), 2020. URL https://github.com/nipkow/fds_ss20/tree/daae0f92277b0df86f34ec747c7b3f1c5f0a725c. Technische Universität München.
- [113] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering Record and Replay for Deployability. In *USENIX Annual Technical Conference*, 2017. URL <https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan>.
- [114] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2018.
- [115] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *International Conference on Software Engineering (ICSE)*, 2012.
- [116] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. 2019.
- [117] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling Typed Holes with Live GUIs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2021. URL <https://doi.org/10.1145/3453483.3454059>.
- [118] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.
- [119] Jibin Ou, Martin T. Vechev, and Otmar Hilliges. An Interactive System for Data Structure Development. In *Conference on Human Factors in Computing Systems (CHI)*, 2015.
- [120] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional Programs That Explain Their Work. In *International Conference on Functional Programming (ICFP)*, 2012.
- [121] Benjamin C. Pierce. *Types and Programming Languages*, pages 132–142. MIT Press, 2002.
- [122] Guy Pierra, Jean-Claude Potier, and Patrick Girard. The EBP System: Example Based Programming System for Parametric Design. In *Modelling and Graphics in Science and Technology*. Springer Berlin Heidelberg, 1996.
- [123] Jan Paul Posma. jsdare: A new approach to learning programming. Master’s thesis, University of Oxford, St Hugh’s College, 2012. <http://jsdares.com/>.

- [124] Guillaume Pothier, Éric Tanter, and José M. Piquer. Scalable Omniscient Debugging. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2007. URL <https://doi.org/10.1145/1297027.1297067>.
- [125] Richard Potter and David Maulsby (Eds.). A Test Suite for Programming by Demonstration. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
- [126] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming - Design and Implementation of An Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.*, 3(3):9, 2019. doi: 10.22152/programming-journal.org/2019/3/9. URL <https://doi.org/10.22152/programming-journal.org/2019/3/9>.
- [127] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting Program Properties From Big Code. In *Symposium on Principles of Programming Languages (POPL)*, 2015.
- [128] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives On Liveness. *The Art, Science, and Engineering of Programming Journal*, 2019. doi: 10.22152/programming-journal.org/2019/3/1.
- [129] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM (CACM)*, 2009.
- [130] Schachman, Toby. Recursive Drawing. Master’s thesis, New York University Interactive Telecommunications Program, 2012. <http://recursivedrawing.com/>.
- [131] Schachman, Toby. Apparatus, 2015. <http://aprt.us/>.
- [132] Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*. 2017.
- [133] Christopher Schuster and Cormac Flanagan. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*, 2016.
- [134] Paul Shen. natto.dev. URL <https://natto.dev/>.
- [135] Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, August 1983.
- [136] David Canfield Smith. *Pygmalion: A Creative Programming Environment*. PhD thesis, Stanford University, 1975.
- [137] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.
- [138] Fabio Spampinato. Highlight VS Code Extension, 2021. URL <https://marketplace.visualstudio.com/items?itemName=fabiospampinato.vscode-highlight>.

- [139] Matús Sulír and Jaroslav Porubán. Augmenting Source Code Lines with Sample Variable Values. In *International Conference on Program Comprehension (ICPC)*, 2018. URL <https://doi.org/10.1145/3196321.3196364>.
- [140] Sun Microsystems. Self: The movie;. <http://www.smalltalk.org.br/movies/self.html>, 1995.
- [141] Ivan Sutherland. *Sketchpad, A Man-Machine Graphical Communication System*. PhD thesis, MIT, 1963.
- [142] William Robert Sutherland. *The On-line Graphical Specification of Computer Procedures*. PhD thesis, Massachusetts Institute of Technology, 1966.
- [143] Steven L. Tanimoto. VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing*, 1990. URL [https://doi.org/10.1016/S1045-926X\(05\)80012-6](https://doi.org/10.1016/S1045-926X(05)80012-6).
- [144] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the ACM (CACM)*, 1981.
- [145] Andrew P. Tolmach and Dino Oliva. From ML To Ada: Strongly-Typed Language Interoperability Via Source Translation. *J. Funct. Program.*, 8(4):367–412, 1998. URL <http://journals.cambridge.org/action/displayAbstract?aid=44181>.
- [146] David M. Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 1987.
- [147] Victor, Bret. Scrubbing Calculator, 2011. <http://worrydream.com/ScrubbingCalculator/>.
- [148] Victor, Bret. Inventing on Principle. <https://vimeo.com/36579366>, 2012.
- [149] Victor, Bret. Drawing Dynamic Visualizations, 2013. <http://worrydream.com/#!/DrawingDynamicVisualizationsTalk>.
- [150] Markus Voelter. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, 2011.
- [151] Helge von Koch. Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. *Arkiv för Matematik, Astronomi och Fysik*, 1:681–704, 1904.
- [152] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.
- [153] K. N. Whitley and Alan F. Blackwell. Visual Programming: The Outlook From Academia and Industry. *Workshop on Empirical Studies of Programmers*, 1997.

- [154] E. M. Wilcox, J. William Atwood, Margaret M. Burnett, Jonathan J. Cadiz, and Curtis R. Cook. Does Continuous Visual Feedback Aid Debugging In Direct-Manipulation Programming Systems? In *Conference on Human Factors in Computing Systems (CHI)*, 1997. URL <https://doi.org/10.1145/258549.258721>.
- [155] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging Code and Interactive Visualization In Computational Notebooks. In *Symposium on User Interface Software and Technology (UIST)*, 2020.
- [156] Haijun Xia, Bruno Araújo, Tovi Grossman, and Daniel J. Wigdor. Object-Oriented Drawing. In *Conference on Human Factors in Computing Systems (CHI)*, 2016.
- [157] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. DemoMatch: API Discovery from Demonstrations. In *Conference on Programming Language Design and Implementation (PLDI)*, 2017.

Appendix A

Deuce: Bimodal Editing on Expressions

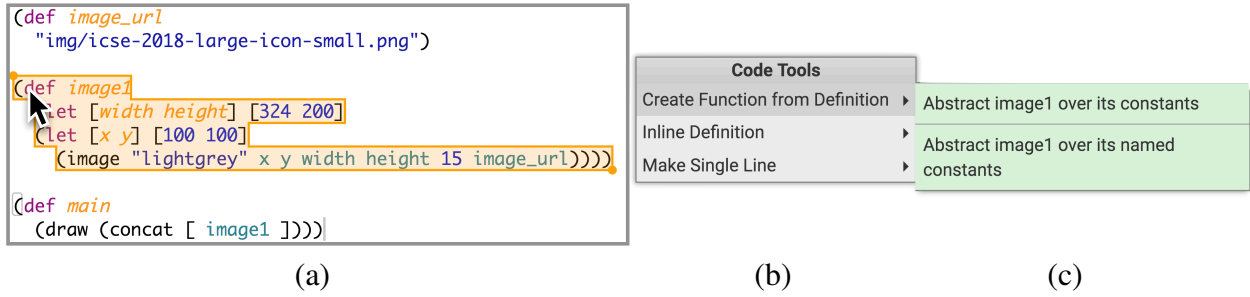


Figure A.1: To streamline refactoring, DEUCE provides (a) structural multi-selection, revealing (b) a short menu of context-sensitive refactorings configured with (c) reasonable defaults.

A.1 Introduction

The systems presented in the main chapters of this work explore leveraging direct manipulation interactions on program *outputs* without sacrificing ordinary text editing. Direct manipulation of program *expressions* is also possible, and has been the subject of a line of work.

Structure editors, Cornell Program Synthesizer [144], treat the program as a tree to offer edit actions that are syntactically valid. If designed appropriately, a structure editor can eliminate the possibility of syntax errors during program construction. In most structure editors, such as MPS [150], tree-structured navigation and editing is primary, with unstructured text editing circumscribed or imitated. Unstructured text editing, however, is flexible, convenient, and time-proven. Can we have the best of both worlds? Could we have bimodal editing of expressions—both (keyboard-based) unstructured *and* (mouse-based) structured interactions?

To explore direct manipulation structure editing on top of unstructured text in an ordinary code editor, we constructed DEUCE, a bimodal (keyboard- and mouse-based) text editor for specifying and invoking code refactorings. DEUCE’s default mode is ordinary text editing, but when the user holds the Shift key DEUCE switches to a *structural multi-selection* mode in which the user clicks to select one or more program expressions (Figure A.1a) and possibly a whitespace *target position* (Figure A.2) to serve as a result location for movement refactorings. Depending on the kind of item(s) selected, a short *context-sensitive menu* of possible transformations is shown (Figure A.1b), with several possible default configuration of the transform in a submenu (Figure A.1c). Hovering the mouse over the result previews its effect on the code, clicking the result applies the refactoring.

Portions of this supplementary chapter are excerpted from a paper at ICSE 2018 (Hempel et al. [62]), focusing on the dissertation author’s contribution to the work (the user study execution and analysis).



Figure A.2: Examples of selectable whitespace (“target positions” for refactorings).

We deliberately forgo a dialog box, because it has been shown that a configuration parameter in a refactoring dialog is only changed around 10% of the time [109].

Ideally, this DEUCE workflow would provide a streamlined interaction for quickly invoking refactorings while still providing a familiar, ordinary text editing buffer for writing code. We performed a user study to compare DEUCE to a more traditional mode of invoking refactorings (*i.e.*, by select and right-click or by finding the tool in a menu, and then configuring via a dialog). We describe this study and the results below.

A.2 Methodology

DEUCE is incidentally implemented in SKETCH-N-SKETCH [59], albeit in a version that predates the Elm-like syntax seen in Chapter 2. SKETCH-N-SKETCH’s output-based features were disabled and hidden for the user study.

To provide a baseline for comparison, we implemented a *Traditional Mode* to simulate the traditional process of invoking refactorings in IDEs such as Eclipse. Our Traditional Mode includes several ways to invoke a refactoring:

Tools menu first. Participants may explore a menu in the menubar that lists all the available refactorings (Figure A.3a). After choosing a menu item, prompted by a configuration panel (Figure A.3c) the user clicks to structurally select appropriate arguments for the transformation. The panel and structural selection is meant to simulate a configuration dialog in a traditional IDE. Once enough arguments are selected, the panel offers possible results.

Text-select first. Alternatively, the user may begin a refactoring via ordinary (non-structural) text selection. As in Eclipse, the whole item must be selected (module leading/trailing whitespace). The user may then choose a refactoring from the Code Tools menu (as in Eclipse, all refactorings are shown even if not valid for the selected item), or may instead right-click the selection to show a contextual menu (Figure A.3b; only valid refactorings are shown, as in Eclipse). After selecting a refactoring, as above the configuration panel directs the user to structurally select any remaining arguments.

Right-click first. As in Eclipse, constants and variables may be immediately right-clicked without text-selecting them first. The item is instantly selected and the contextual menu appears as above.

The experimental condition, DEUCE Mode, offered the same refactorings, except that the workflow proceeded as described in the introduction: the user holds down the Shift key, structurally selects one or more desired elements (Figure A.1a), and chooses the refactoring and a result from

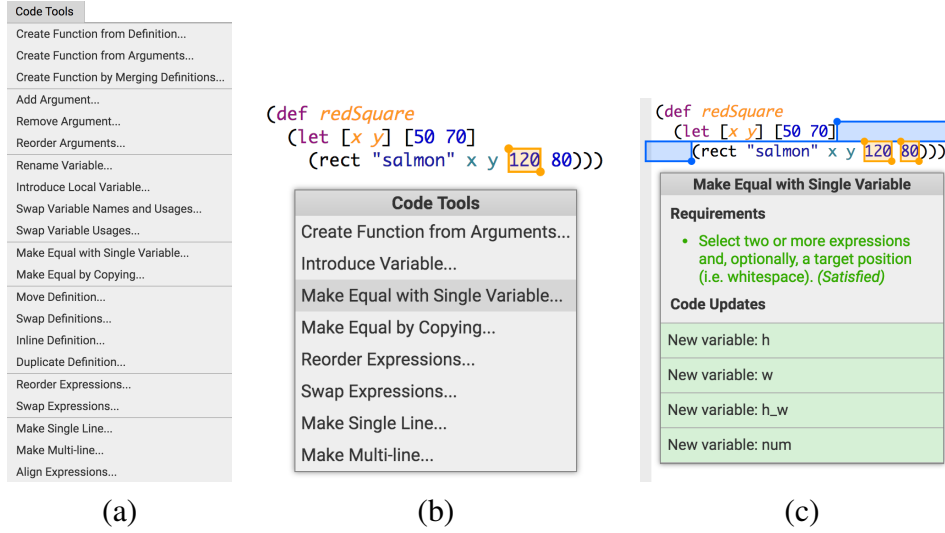


Figure A.3: Traditional Mode UI elements. (a) Code tools menu listing all the refactorings available. (b) Right-click menu after a single argument is selected. (c) Configuration panel explaining refactoring selection requirements and offering results.

a short, context-sensitive menu (Figure A.1b,c). This context-sensitive menu is often shorter than that shown via right-click in Traditional Mode (Figure A.3b), because DEUCE Mode only shows those refactorings that are valid for precisely the selected elements, whereas Traditional Mode must also show those refactorings that could become valid if more elements are selected.

Study Design

We sought to answer the following questions:

- Compared to Traditional Mode, is DEUCE more effective for (a) completing tasks, (b) rapid editing, or (c) achieving more with fewer transforms?
- Compared to Traditional Mode, is DEUCE preferred by users? In which cases?

We conducted a within-subjects study with 21 undergraduate and graduate students recruited from the University of Chicago. Each participant individually completed a 2 hour session and was paid \$50. The sessions were structured as follows.

After a guided tutorial in the system walked the participant through the interaction mechanisms for both Traditional and DEUCE Modes, the user attempted the six tasks summarized in Table A.1. The first four tasks were “head-to-head” tasks. Each was performed twice, once in each mode, resulting in eight trials. The order of the trials was randomized. The last two tasks were “open-ended” tasks. These tasks were performed once each, but with both Traditional and DEUCE Modes active simultaneously (their operation is orthogonal). The participant was free to mix-and-match between modes for these two tasks.

Before each trial, the participant was given a reading period to inspect the starting code. Once they began a trial, a textual description of steps to perform was shown (e.g., “move the ring

Name	#LOC	#Transforms	Example Tool Sequence (with minimum number of transforms required)
One Rectangle	9	3	Swap Expressions; Move Definition; Swap Definitions
Two Circles	11	2	Create Function from Definition; Reorder Arguments
Three Rectangles	11	2	Creating Function by Merging Definitions; Rename
Four Rings	7	4	Remove Argument; Rename; Move Definition; Add Arguments
Four Squares	9	7	Create Function by Merging Definitions; Create Function from Arguments; Rename (5x)
Lambda Icon	10	8	Make Equal with Single Variable (6x); Introduce Variable; Rename

Table A.1: Overview of the four head-to-head and two open-ended tasks. #LOC is non-blank lines of code in the starting program. A way to complete each task with a minimum number of tool invocations is indicated.

definition inside target”), as well as the expected final code. Lines differing between the current code and the intended solution were highlighted in the sidebar. Participants could click a button to “Give Up” at any time and were limited to six minutes per head-to-head trial and twelve minutes per open-ended trial, respectively. To focus on the refactoring interactions, text edits were disabled throughout. Participants completed an exit survey after the final trial.

A.3 Results

Participants reported between 2 and 10 years of programming experience (mean: 5.1), of which between 0 and 3 years involved functional programming (mean: 0.76). 10 participants (48%) reported no prior functional programming experience. 8 participants reported using tools that supported automated refactoring (Eclipse, IntelliJ, and PyCharm all received multiple mentions). 4 participants reported some prior exposure to previous versions of the SKETCH-N-SKETCH project, but none reported knowledge of the code tools presented in the study.

For the study itself, 8 users brought their own laptop, the remaining 13 used ours. 15 participants used a mouse, and 6 relied on their laptop’s trackpad. Each session took a mean of 1hr 44min (range: 1h 11m – 2h 27m). Users spent between 23 and 66 minutes on the tutorial (mean: 41) and 20 and 65 minutes on the tasks (mean: 44). The remaining time was spent on introductory remarks and the exit survey. All users attempted all tasks. Two trials were discarded because of tool malfunction, for a final total of 166 head-to-head trials and 42 open-ended tasks suitable for analysis.

The tasks proved moderately difficult. On average, each participant successfully completed

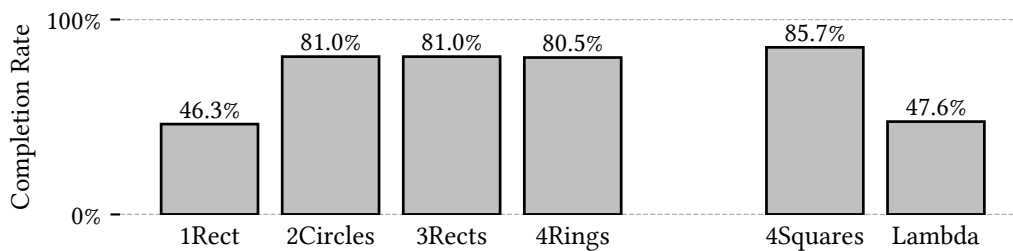


Figure A.4: Task completion rates pooled over both modes.

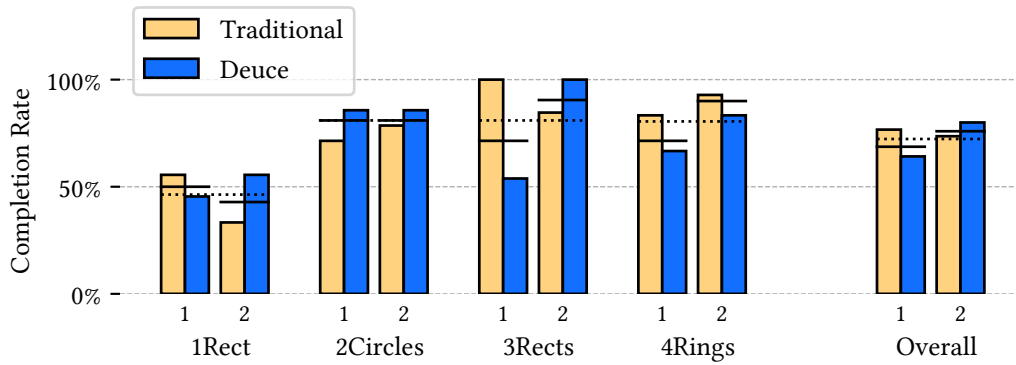


Figure A.5: Head-to-head task completion rates by mode and by subject's first/second encounter with task. Overlaid lines indicated pooled completion rates.

71% of the trials and open-ended tasks within the time limits, with 3 users completing them all and 1 user failing to complete any. Figure A.4 shows completion rates by task. The One Rectangle and Lambda tasks had particularly low completion rates. Based on videos of failed attempts, many users struggled with choosing appropriate tools—*e.g.*, many chose `INTRODUCE VARIABLES` rather than `MAKE EQUAL`, and some chose `INLINE` rather than `MOVE DEFINITIONS` in an attempt to create a tuple definition. The tutorial was not sufficient for everyone to remember and understand all the tools needed for the tasks. The task descriptions may have also presented obstacles—*e.g.*, for Lambda, the phrase “Define and use...”, along with `(def [x y w h] ...)` in the final code, may have led some to use `INTRODUCE VARIABLES`, which would then require several roundabout transformations to complete the task. We believe these difficulties are largely independent of the user interface features. We now address each of the research questions in turn.

Is either mode more effective for completing tasks? Figure A.5 breaks down completion rates for head-to-head tasks by mode. Because each was attempted twice, to assess possible learning effects from already completing a task in the other mode, Figure A.5 also differentiates between the user's first or second encounter with each task. Visually, the data suggest that on the first encounter with a task, Traditional Mode may better facilitate completion, and is also a better teacher for the subsequent encounter with DEUCE Mode. In contrast, a first encounter with DEUCE Mode may be less helpful for the second encounter with Traditional Mode.

To control for learning effects, a mixed effects logistic regression model [47] was fit with `lme4` [8] to predict task completion probability based upon fixed effect predictors for the mode (coded as 0 or 1), the trial number (1-8), whether the trial was the second encounter with the task (0 or 1), whether the participant used a mouse (0 or 1), whether the participant used their own computer (0 or 1), and the interaction of mode with the second encounter (0, or 1 when DEUCE Mode and a second encounter). To model differences in user skill and task difficulty, a random effect was added for each participant as well as each task, and a random interaction was added to model differences in the second encounter difficulty per task. Reported p-values are based on Wald Z-statistics.

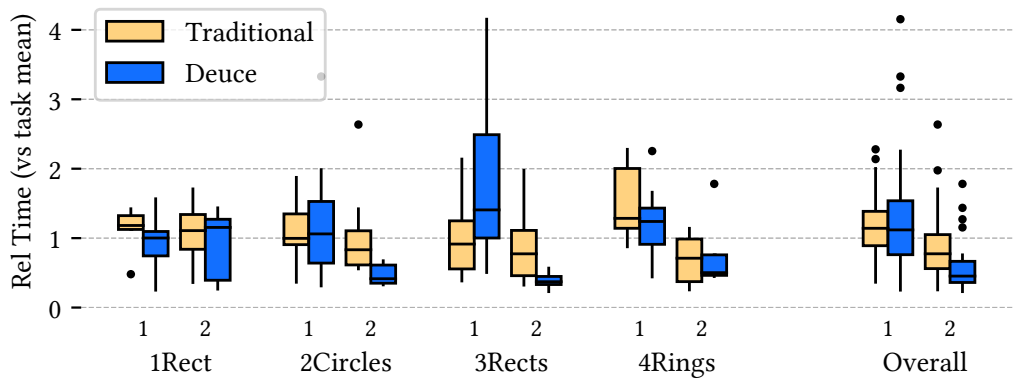


Figure A.6: Head-to-head task durations for successfully completed trials, scaled relative to the mean time per task.

In the fit model, the coefficient for mode was on the edge of significance ($p=0.057$), indicating that Traditional Mode did better facilitate task completion on the first encounter with a task. Given this, DEUCE Mode performed better than expected on the second encounter (interaction term $p=0.036$), but not enough to confidently say that DEUCE Mode was absolutely better than Traditional Mode for the second encounter ($p=0.17$). No other fixed effect coefficients approached significance.

DEUCE Mode therefore seems to present a learning curve, but may be just as effective as Traditional Mode once that learning curve is overcome. This interpretation accords with the surveys: 5 participants wrote that Traditional Mode might be better for learning, and 4 participants—including 3 of the previous 5—said DEUCE Mode was better when they knew the desired transformation. However, the data may be alternatively explained if DEUCE Mode on the first encounter is a poor teacher, actively misleading users on the second encounter with Traditional Mode.

Is either mode more effective for rapid editing? Among trials successfully completed, the duration of each trial was measured from the start of configuration of the first refactoring to the end of the final refactoring. The distribution of these timings is presented in Figure A.6, scaled relative to the mean duration for each task. Again, to tease out if any of these differences are significant, from the same predictors described above two linear mixed effects models were fit to predict (1) trial duration and (2) the logarithm of trial duration (*i.e.*, considering effects to be multiplicative rather than additive). Percentile bootstrap p -values for the fixed effect coefficients were calculated from 10,000 parametric simulate-refit samples.¹ For the first encounter with a task, Traditional Mode was insignificantly faster (by 13 seconds, $p=0.44$; or 9.2%, $p=0.52$). However, DEUCE Mode was on average 25 seconds ($p=0.13$) or 36% ($p<0.01$) faster for the second encounter with a task, suggesting that DEUCE Mode may be faster once users become familiar with the available tools. Most of the gain comes from less time spent in configuration—after discounting all idle thinking time between configurations, the model still reveals an 18 second difference.

¹See <https://www.rdocumentation.org/packages/lme4/versions/1.1-13/topics/bootMer>

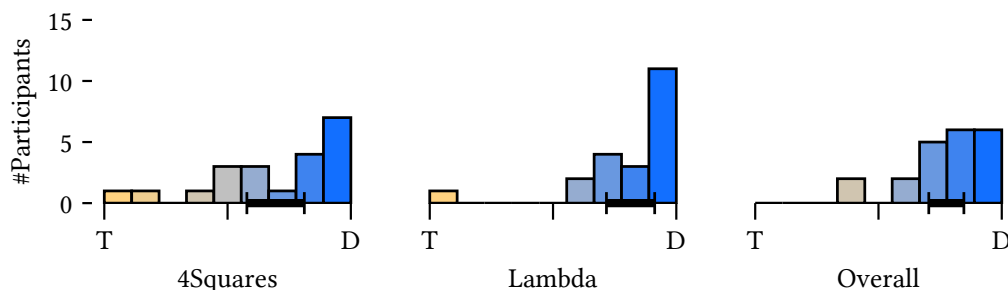


Figure A.7: Distribution of user preferences for Traditional vs. DEUCE Modes as measured by the ratio of refactorings performed by the user in each mode on the open-ended tasks. Far left represents all Traditional Mode refactorings; far-right indicates all DEUCE Mode refactorings. The 95% confidence interval for the mean preference across all users is indicated (via percentile bootstrapping, 10,000 samples).

Is either mode more effective for achieving more with fewer transforms? To determine if either mode facilitated more efficient use of interactions, the same mixed effects model was fit to predict the number of refactorings invoked during each successful trial, as well as the number of Undos. On the first encounter with a task, Traditional Mode accounted for an average of 2.0 fewer refactorings ($p < 0.01$) and 2.1 fewer Undos ($p < 0.01$), but on the second encounter no significant difference in number of refactorings or Undos was indicated. As a second encounter with DEUCE Mode is faster than Traditional Mode, the speed gain thus appears to be explained by faster invocations rather than fewer invocations.

Is either mode preferred by users? In which cases? The two final open-ended tasks allowed participants to mix-and-match the two modes as they pleased. As shown in Figure A.7, on both tasks the overwhelming number of users performed a greater share of refactorings using DEUCE Mode. We believe a main advantage of DEUCE Mode is that it simplifies the configuration of refactorings that require multiple arguments, as the user may select all the arguments together before choosing a transformation from a short menu. In Traditional Mode, the workflow is stuttered: the user must select a single argument, right-click to choose a transformation, then select the remaining arguments. However, for a refactoring requiring only a single argument, Traditional Mode is more streamlined: a user may simply select the desired transformation immediately after right-clicking on the first argument. Thus, for single-argument refactorings, DEUCE Mode's advantages may be limited. A breakdown of mode usage by popular tools (Figure A.8) lends support to this hypothesis. For the most commonly used tool, RENAME, which always takes only a single argument, participants used Traditional and DEUCE Modes with roughly even frequency. Most other tools showed strong preferences towards DEUCE Mode, with the notable exception of CREATE FUNCTION BY MERGING DEFINITIONS. Because the Four Squares task required invoking this tool with four expressions, according to the hypothesis, users should prefer DEUCE Mode. The videos revealed that several users were unable to discover how to structurally select a function call, which required hovering on the open parenthesis (not demonstrated in the tutorial). Several of these users were, however, able to invoke the tool by text-selecting a function call or by starting

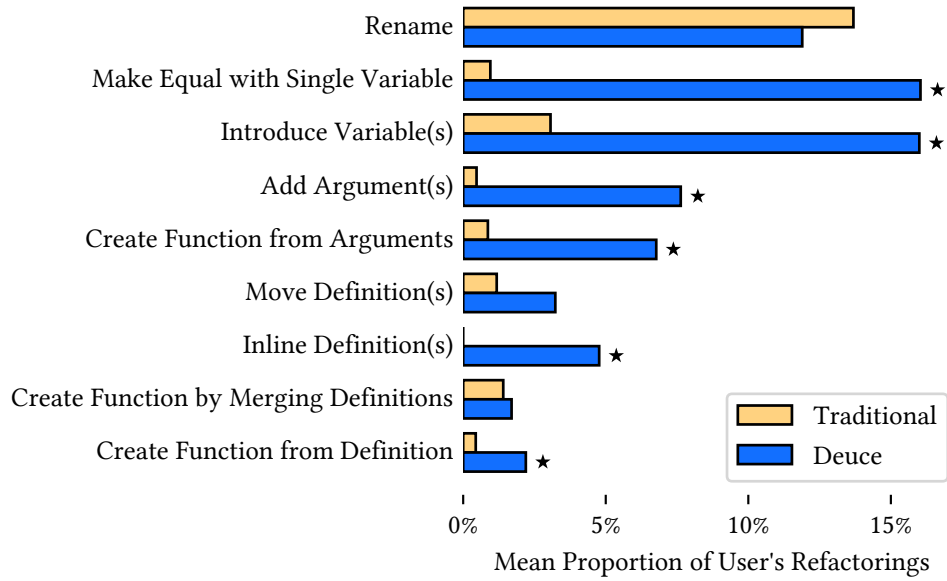


Figure A.8: Mode usage for tools used by at least half of participants on the open-ended tasks. Deuce mode is preferred for most tools. Stars indicate differences significant at the 95% level (via percentile bootstrapping, 10,000 samples).

from the full Code Tools menu.

Subjectively, the concluding survey asked whether DEUCE or Traditional Mode worked better for each head-to-head task, measured on a 5-point scale from “Text-Select Mode worked much better” to “Box-Select Mode worked much better”. For each participant, a random choice determined which mode appeared at each end of the scale. As shown in Figure A.9, on average a similar modest preference for DEUCE Mode was expressed for each task.

Altogether, users demonstrated a strong objective and modest subjective preference for DEUCE over Traditional Mode, suggesting that DEUCE accomplishes its goal to provide a more human-friendly interface to identify, configure, and invoke refactorings.

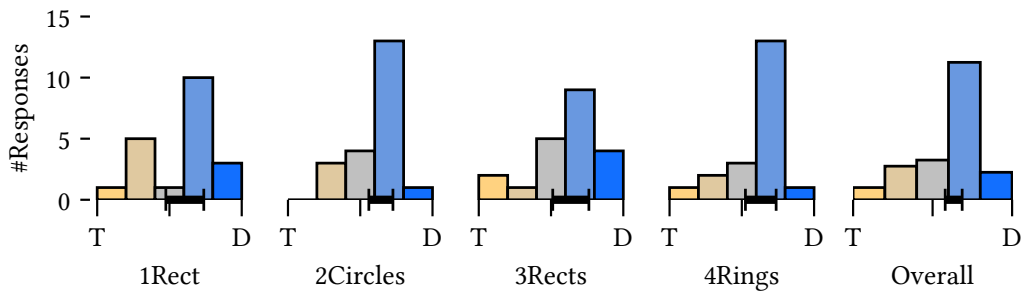


Figure A.9: Surveyed subjective preference for Traditional vs. DEUCE Modes for the head-to-head tasks. The 95% confidence interval for the mean preference across all users is indicated (via percentile bootstrapping, 10,000 samples).

A.4 Related Work

Murphy-Hill and Black [108] demonstrates two relevant mechanisms to help users make valid selections for invoking refactorings. The first, *Selection Assist*, shades a region of code under the cursor indicating the extent of the relevant statement, thereby hinting the user how they may begin and end their selection at valid expression boundaries. The second, *Box View*, displays a sidebar with miniature nested boxes corresponding to the nested expression structure of the adjacent textual code. Clicking a box text-selects the entire expression in the code. Murphy-Hill [107] presents a third mechanism, *Refactoring Cues*, similar to the Traditional Mode in our user study: after choosing a refactoring from a list, the user may click to structurally select expressions in the code. Unlike our implementation, once a refactoring is selected Refactoring Cues preemptively draws boxes (*cues*) around all expressions in the code rather than just the expression currently under the mouse.

Of the many structure editors of the 1980's, PSG [7] is notable for, like DEUCE, offering a hybrid interface with both textual and structural input supported. Unlike DEUCE, PSG's structural actions focused more on AST construction rather than refactoring. More recently, Barista [81] and Greenfoot [18] are structure editors that also imitate an ordinary text buffer. Code in Greenfoot is organized into *frames* which correspond to, *e.g.*, method bodies, if-statements, and looping constructs. Frames can be dragged and dropped or created via keyboard shortcuts. Statements within a frame are displayed as ordinary textual code but are still handled as structured elements, although Greenfoot attempts to make editing of these structured elements behave somewhat like ordinary text editing. Barista, as well, offers a mix of both box and text-like editing in a structured environment. Barista additionally enables customized views for certain expressions in a program, *e.g.*, math expressions may be rendered in a beautified, typeset style.

Lee et al. [89] identify a subset of Eclipse's refactorings that can be unambiguously invoked via drag-and-drop, forgoing the need for the configuration dialog step. DEUCE currently requires the user to select a target position (Figure A.2) for many refactorings, the addition of a drag-and-drop capability might make invoking these refactorings more convenient.

A.5 Conclusion

Our user study results suggest that while a traditional refactoring environment may be better for learning which refactorings exist and how to use them, DEUCE may be faster to use once it is learned. Moreover, in their actual usage on the open-ended tasks, participants demonstrated a strong preference for DEUCE mode. These results accord with DEUCE's aim to provide human-friendly structural interactions on top of familiar text-based editing.