THE UNIVERSITY OF CHICAGO

OUTPUT-DIRECTED SVG PROGRAMMING

IN CANDIDACY FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
BRIAN HEMPEL

CHICAGO, ILLINOIS
APRIL 26, 2019

TABLE OF CONTENTS

ABSTRACT iv

1 INTRODUCTION 1
1.1 Text 1
1.2 Live Programming 1
1.3 Output-Directed Programming 1
1.4 Sketch-n-Sketch 2
1.5 Contributions 4
1.6 Outline 5

2 RELATED WORK 6
2.1 Parametric Computer-Aided Design (CAD) 6
2.2 Drawing With Constraints 6
2.3 Constraint-Oriented Programming (COP) 6
2.4 Programming by Demonstration (PBD) 7
2.5 Output-Directed Programming 8

3 OVERVIEW EXAMPLES 9
3.1 Example 1: Koch Snowflake 9
3.1.1 One-Third Point Function 10
3.1.2 Equi-Tri Point Function 14
3.1.3 Recursive Koch Curve Points Function 15
3.1.4 Koch Snowflake Polygon 19
3.2 Example 2: Tree Branch 21
3.3 Example 3: Target 23

4 TOOLS 25
4.1 DRAW 25
4.1.1 Drawing Shapes 25
4.1.2 Custom Drawing Functions 25
4.1.3 Focused Drawing 25
4.1.4 Draw Point 26
4.1.5 Draw Offset 26
4.1.6 Dupe 26
4.1.7 Delete 26
4.2 RELATE 26
4.2.1 Make Equal 26
4.2.2 Relate 27
4.2.3 Distance Features 27
4.3 GROUP, ABSTRACT, REPEAT 27
4.3.1 Group 27
4.3.2 List Widgets 27
4.3.3 Abstract 28
4.3.4 Merge 28

4.3.5 Repeat over Function Call . . . 28
4.3.6 Repeat over Existing List . . . 28
4.3.7 Repeat by Indexed Merge . . . 28
4.3.8 Fill PBE hole . . . 29
4.4 REFACTOR . . . 29
4.4.1 Rename in Output . . . 29
4.4.2 Add / Remove / Reorder Argument . . . 29
4.4.3 Reorder in List . . . 29
4.4.4 Add to Output . . . 29
4.4.5 Select Termination Condition . . . 30

5 IMPLEMENTATION . . . 31
5.1 Introduction . . . 31
5.2 Provenance . . . 31
5.2.1 “Based On” provenance . . . 32
5.2.2 Interpreting “Based On” provenance . . . 34
5.2.3 “Parents” provenance . . . 37
5.2.4 Expression IDs . . . 38
5.2.5 Approaches to Provenance in Other Work . . . 39
5.3 Value Holes . . . 41
5.4 Programming by Example Holes . . . 41
5.5 Roles . . . 42

6 EVALUATION . . . 44
6.1 Precision Floor Plan (Figure 6.1d) . . . 44
6.2 Mondrian Arch (Figure 6.1e) . . . 44
6.3 Balance Scales (Figure 6.1f) . . . 46
6.4 Box Volume (Figure 6.1g) . . . 46
6.5 Xs (WWID: PBD p. 591 David Maulsby) . . . 47
6.6 Tackling the Remaining WWID: PBD Tasks . . . 48

7 DISCUSSION . . . 49
7.1 Lessons Learned . . . 49
7.1.1 Is it working? . . . 49
7.1.2 Intermediates . . . 49
7.1.3 Focused Contexts . . . 50
7.1.4 Programing by Demonstration or by Direct Manipulation? . . . 51
7.1.5 Could you hide the code? . . . 51
7.2 Limitations and Future Work . . . 52
7.2.1 SVG-Specific Improvements . . . 52
7.2.2 SVG and General Improvements . . . 53
7.2.3 General Improvements . . . 54
7.2.4 Graphical Widgets for Non-Visual Domains . . . 55

8 CONCLUSION . . . 56

REFERENCES . . . 57

ABSTRACT

We propose *output-directed programming*, a paradigm whereby users directly manipulate the output they wish to create while the system automatically builds a text-based program that produces the output.

Specifically, we present output-directed programming techniques for programs that generate Scalable Vector Graphics (SVG) designs. Like a traditional graphics editor, the system presents a direct manipulation interface for drawing and manipulating shapes on a canvas. Unlike a traditional editor, the drawing is represented as a text-based program in an ordinary functional programming language. Direct manipulation actions to *draw*, *relate*, *group*, and *repeat* shapes are affected by transforming the program. Although the program remains text-editable at any time, we show how the output-directed tools enable a variety of complex, readable programs to be constructed without any text-based program editing.

We evaluate the techniques by implementing over a dozen parametric designs. While some of the output-directed programming interactions are—by necessity—domain-specific, we describe how others are implemented in a generalized way, to support their application to other output-directed programming domains.

CHAPTER 1
INTRODUCTION

1.1 Text

Most programs are created by writing text. Although flexible and familiar, text-based programming presents a steep learning curve to novices as they approach programming, and experts, though adept at manipulating text, are still “playing computer in their heads”[1] as they construct their program. Given this observation, could development environments better help programmers understand and construct programs?

1.2 Live Programming

In order to relieve the programmer of the need to simulate the operation of the computer in their head, a line of work dubbed *live programming* has sought to quicken the feedback loop while a programmer writes code. Generally, live programming systems continuously display the output of the user’s program while it is being constructed. Updates to the program’s code are immediately reflected in the live output display, without requiring the user to manually invoke a compile-and-run action. Live programming systems have targeted both educational [49, 60, 48, 28] and expert [19, 5] domains.

1.3 Output-Directed Programming

Could live programming be taken further? Programs are often incorrect while under construction—instead of merely seeing the program’s incorrect output and having to hunt in the code for the lines to modify, could the programmer instead *directly manipulate the output itself* to automatically affect a program repair?

Such output-directed interactions could *augment* rather than *replace* the traditional text-based paradigm for writing code, while at the same time offering streamlined interactions for building up large portions of the program. But because the program is still text, any functionality that cannot be added to the program by direct manipulation can still be created by regular text editing. By building on top of text editing rather than supplanting it, the power of traditional text-based programming is a *lower bound* for the power of such a system.

1. Chris Granger, quoted in [56]

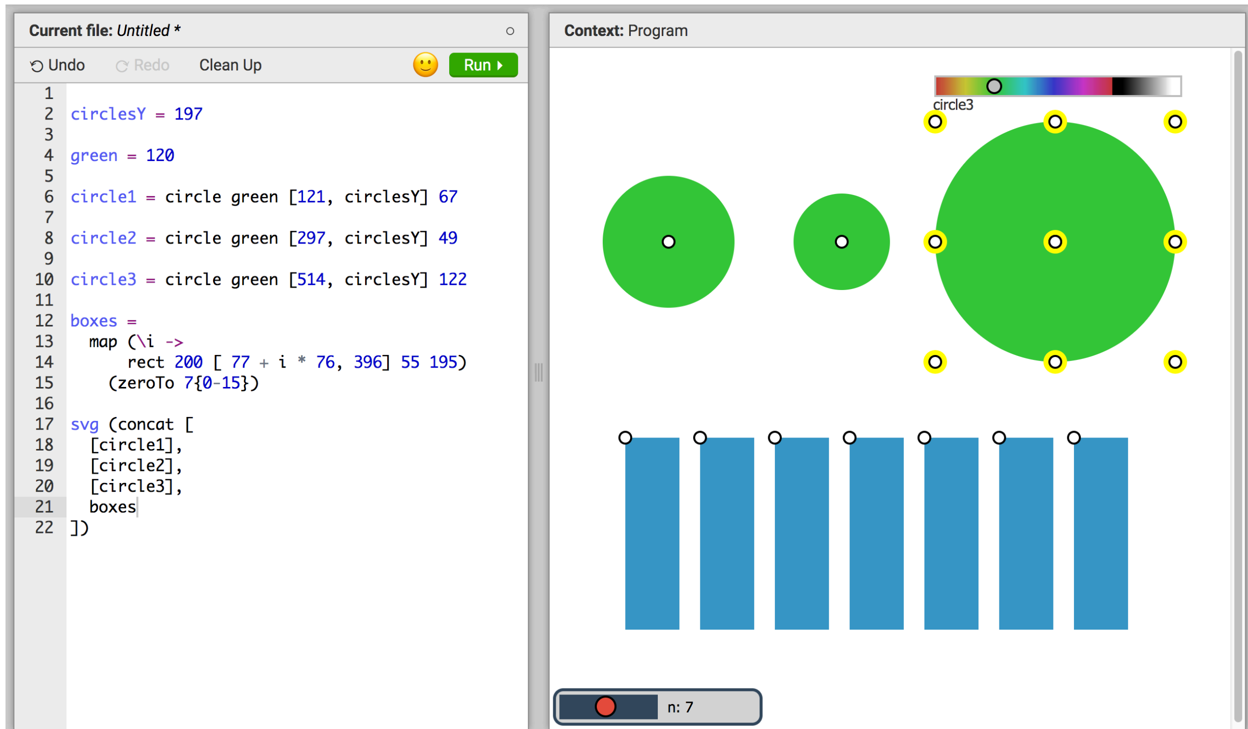


Figure 1.1: The two-pane interface for the Sketch-n-Sketch vector graphics programming environment. The code box on the left contains a simple program whose SVG output is displayed on the canvas to the right. Programs may be modified either by text edits in code box or by direct manipulation of the output on the canvas. In this instance, the third circle is selected on the canvas, revealing a color slider to manipulate the circle's fill color; a second slider at the bottom of the canvas controls the number of repeated blue boxes. Shapes may also be moved and resized as in a traditional vector graphics editor.

1.4 Sketch-n-Sketch

To explore and extend the capabilities of output-directed programming, this work builds upon the Sketch-n-Sketch programming environment introduced by Chugh et al. [12]. Sketch-n-Sketch is an interface for creating programs that output scalable vector graphics (SVG) images. Unlike a traditional source code text editor, the Sketch-n-Sketch interface of Chugh et al. [12] includes a basic set of output-directed programming functionality: once a programmer has written a program that outputs SVG, they may directly manipulate the output to move, resize, and recolor shapes. Appropriate constants in the program are updated accordingly. We now introduce Sketch-n-Sketch in more detail and describe its initial output-directed programming mechanisms. In the next section (§ 1.5), we will summarize how the present work extends these mechanisms.

Figure 1.1 displays the two panes central to the Sketch-n-Sketch interface. The left pane—the *code box*—is an ordinary source code text editor. The right pane—the *canvas*—displays the program's SVG output along with UI elements for manipulating that output.

The code box allows the user to create their program using traditional text editing. The language in Sketch-n-Sketch is a simple, functional programming language—an extended lambda calculus with syntax inspired by Elm [15].[2] Figure 1.1 shows a simple Sketch-n-Sketch program. The variables `circle1`, `circle2`, and `circle3` each contain a circle value produced by a call to the `circle` library function. These shape values are immutable once created (akin to the *picture* type of Henderson [22, 23]). The circles share the same color (the variable `green`) and the same y position (the variable `circlesY`), but differ in their x positions and radii. Designs may also be produced through higher-order programming. The anonymous function on lines 13-14 takes an index `i` and produces a single rectangle whose x position depends on that index. Mapping this function over the the indices `0` through `6` (*i.e.*, `zeroTo 7`) creates a list of identical boxes, each differing only in their x position. This list is stored in the `boxes` variable. Finally, the last statement of the program is the program's output: the circles and the boxes list are flattened (via `concat`) into a single shape list which is given to the `svg` library function to produce a final SVG value out of the shapes. This final SVG value is rendered graphically on the canvas.

Unlike a traditional programming environment, Sketch-n-Sketch's output canvas is not static. Similar to a vector graphics editor, Sketch-n-Sketch allows the user to directly manipulate the shapes on the canvas: the user may move, resize, or recolor shapes. Modifications to the shape properties are realized by changing appropriate numeric constants in the program. For example, dragging the third circle horizontally will change its x value on line 10 (the literal `514`). Spatial constraints between shapes are encoded in the program structure: because the three circles share the `circlesY` variable for their y position, dragging any circle vertically will modify the value of that variable (the literal `197` on line 2) and all three circles will take on the new y position, maintaining their alignment. Non-spatial shape properties such as color may also be manipulated: in Figure 1.1, `circle3` is selected and its fill color slider is displayed above the shape—dragging this slider will change the value of the `green` variable on line 4 and thus the colors of all three circles. Finally, numeric literals not associated with a shape property can still be exposed for direct manipulation by annotating the literal with a *range*, *e.g.*, `{0-15}`. A range annotation instructs Sketch-n-Sketch to draw a slider on the canvas to manipulate the number. For example, on line 15, the numeric literal `7` specifies the number of boxes to draw; that literal `7` is annotated with the range `{0-15}`, which exposes a slider on the bottom of that canvas to modify the literal between the numbers 0 and 15. Dragging the slider will change the number of boxes.

All of the dragging operations above are designed to operate fluidly. Sketch-n-Sketch updates the program, reruns it, and renders the output *continuously* while the user moves their mouse, providing multiple frames per second of feedback. Chugh et al. [12] thus call this scheme *live synchronization.*

2. Unlike Elm, the Sketch-n-Sketch language offers heterogeneous lists as its only data structure. Tuples and lists are thus conflated. The purpose of this footnote is to advise readers against adopting this scheme in their own work.

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 [12]. For example, consider the simple program `let x = 5 in x + 10`. Sketch-n-Sketch will evaluate the program to the tagged number $\texttt{15}^{\ell_1+\ell_2}$ where $\ell_1 + \ell_2$ is a *numeric trace* explaining the origin of the value `15`; within the trace, ℓ_1 and ℓ_2 refer to the program locations of the literals `5` and `10` respectively. When, on the canvas, the user manipulates the number `15`, Sketch-n-Sketch chooses a location to change (using heuristics [12]), 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 when substituted into the program results 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.

1.5 Contributions

As described above, the Sketch-n-Sketch of Chugh et al. [12] is an *output-directed programming* environment, allowing program modification either by performing text edits to the source code *or* by directly manipulating the output. These output-directed interactions, however, only change numeric literals. To construct the program in the first place, the programmer must type out the entire program using the keyboard. Could output-directed programming in Sketch-n-Sketch be extended to help with program construction as well?

In this work, we do just that. We extend Sketch-n-Sketch with the ability to construct whole designs solely via interactions on the canvas.

Specifically, we extend Sketch-n-Sketch with new direct manipulation tools for *drawing* new shapes (by adding new definitions to the program), for *relating* the attributes of shapes (by introducing new shared variables and arithmetic expressions), for *grouping*, *abstracting*, and *repeating* shapes (by transforming individual definitions into reusable functions abstracted over design parameters), and for *refactoring* the program based on selections in the output (by associating output selections with program expressions and invoking standard refactorings on those expressions). To facilitate these new features, we develop techniques for tracking value *provenance* to associate output selections with relevant program expressions.

We also make several UI choices to further enhance the process of program construction. We expose selected *intermediate execution products* on the canvas for manipulation, so the programmer is not limited to manipulating their final output, and we offer *focused editing* to allow the programmer to insert nested definitions and create recursive functions.

These output-directed programming tools are used to construct several high-level, readable programs that generate non-trivial designs.

1.6 Outline

The user interface and program transformation techniques are best described by example. Therefore, after discussing related work (Chapter 2), a substantial portion of the presentation is a detailed description of how three example programs can be built in Sketch-n-Sketch solely using output-directed program transformations (Chapter 3). Having introduced most of the tooling in context, we then briefly enumerate each tool's operation (Chapter 4) before exploring shared underlying technical mechanisms that facilitate that operation, such as our methods for provenance tracking (Chapter 5). Afterwards, we evaluate the system by constructing another 13 example programs (for a total of 16 examples), focusing our discussion on tasks drawn from the benchmark suite proposed in *Watch What I Do: Programming by Demonstration* [1] (Chapter 6). Finally, we discuss lessons learned, limitations, and future work (Chapter 7), and conclude (Chapter 8).

CHAPTER 2
RELATED WORK

In this work, we present an *output-directed programming* system that allows the programmer to either text-edit their program *or* to directly manipulate the output of their program to affect a program repair. We specifically focus on programs that create vector graphics designs. Borrowing familiar UI interactions from shape-drawing applications, we adapt those actions to create and transform shape-drawing programs written in a general-purpose functional programming language. Several related approaches also offer interfaces to create parametric designs or to transform programs via direct manipulation.

2.1 Parametric Computer-Aided Design (CAD)

Feature-based parametric CAD systems 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. Among CAD systems, EBP [47] is notable for allowing the user to perform a step-by-step demonstration to create loops and conditionals.

2.2 Drawing With Constraints

Several systems integrate constraint specification into the visual design process, but are targeted at drawing creation and thereby do not seek to expose a traditional, text-based programming language (*e.g.*, [65, 26]). Of these systems, Apparatus [51], Recursive Drawing [52], and Geometer's SketchPad [25] are notable for supporting recursion.

In contrast to these systems, we are more interested in programming qua programming rather than in creating visual (or geometric) design, hence we explicitly expose the text-based program for editing. Although we expect our techniques might be useful for the actual creation of parametric drawings, we primarily view parametric drawing as a convenient initial domain for discovering how one might build programs by output manipulation. Thus, we have focused the development of our tool on exploring novel program transformations rather than on optimizing the usability for practical design tasks.

2.3 Constraint-Oriented Programming (COP)

Other constraint-oriented systems, following in the footsteps of Sketchpad [59], explicitly view building a constrained system as a programming task [6, 24, 16]. While offering varying degrees of visibility into the code, these systems are distinguished by running constraint

solvers alongside the program, querying those solvers *during runtime* to affect the execution of the program.

Our system instead follows a standard execution model. Rather than running a constraint solver while the program executes, any "constraints" are expressed as ordinary math in the program (*e.g.*, `x2 = x1 + w/2`) and are executed normally. New constraints may be added by changing the program—*i.e.*, adding math to the program—when the program is not running. That new math may be automatically generated by a solver, but the solver is invoked as part of program transformation, after and separate from program execution.

2.4 Programming by Demonstration (PBD)

To offer end-users some of the benefits of programming, a class of interactions dubbed *programming by demonstration (PBD)* [13] 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.*, [31, 34]), or show actions step-by-step [38]. We also use shape drawing as a concrete application domain, but we do not hide the program text.

Although not as visual as peer PBD systems, Tinker [35] is notable for supporting recursion by demonstration—indeed, any Lisp expression may be created. Unlike our work, manipulations are performed on a symbolic representation of the example not far removed from the underlying Lisp. But, like our work, the underlying code is set in a traditional programming language and that code is featured in the UI.

More recently, PBD techniques have been developed with a more practical bent. These systems address a wide variety of domains, such as data visualization [61], mobile applications for collaboration [14], web scraping [3, 9], and API discovery [67]. Each of these systems focus on solving a particular domain task but, unlike the system presented here, either do not expose the program as plain text in an ordinary language, or do not offer demonstration-based editing after the initial program is generated.

A further philosophical difference should be noted. Traditional PBD systems are generally organized around a user demonstrating an imperative sequence of actions on a stateful system. Each user action generates one or more (effectful) statements in the language (*e.g.*, "Add a circle at such-and-such position"). Although the user actions in our system are also given step-by-step, our system instead builds functional, effect-free programs. Rather than each user action generating one or two imperative statements, each user action instead specifies a *program transformation* that may not add any expressions to the program, instead modifying one or more existing expressions. The underlying artifact always remains a pure functional program. We embrace this approach because we believe such techniques for transforming side-effect-free expressions are more likely to generalize to non-visual domains.

2.5 Output-Directed Programming

A number of recent systems offer a regular text-based programming experience augmented with the ability to directly manipulate output to enact code changes. The changes that can be affected by output-directed manipulation may be "small," *e.g.*, changing constants [12, 32, 39], strings [63, 54, 32, 39], or list literals [39], but, like our improved Sketch-n-Sketch, several systems enable "larger" program changes via output manipulation. We discuss two such systems below.

Transmorphic [53] re-implements the Morphic UI framework [36] 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 changing live object state. The programmer may use direct manipulation to add morphs, remove morphs, or change a morph's primitive properties. Transmorphic's code transformations are based on associating each visual morph with a syntactic location in the program via a static analysis pass over the program. We instead use runtime tracing built in to the evaluator to map output values to program expressions.

Like the work presented here, APX [40, 41] is a two-pane (code box and output canvas) environment for creating programs that draw pictures. APX additionally supports creation of realtime 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.

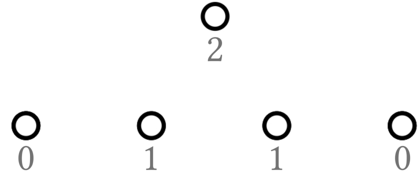
CHAPTER 3
OVERVIEW EXAMPLES

We introduce the improved SKETCH-N-SKETCH system via three examples, explaining the workflow to construct each example. Our main example ("Koch Snowflake") introduces the workflow and a number of SKETCH-N-SKETCH's tools and UI features. Two shorter examples follow ("Tree Branch" and "Target") to showcase features not exercised by the main example.

Although we are able to perform regular text-edits to our program at any point during its construction—and we may do so without losing access to the output-directed tooling for later edits—we choose to limit ourselves to performing *only* output-directed edits, thereby highlighting the expressive power of SKETCH-N-SKETCH's new tooling.

In the text below, the names of our new tools are written in small caps (*e.g.*, ADD TO OUTPUT), with boldface the first time the tool is introduced (*e.g.*, **ADD TO OUTPUT**). Code examples shown are as produced by SKETCH-N-SKETCH—newlines inserted for formatting purposes in this paper are escaped by a "\" backslash.

3.1 Example 1: Koch Snowflake



For the first overview example, we construct a program that draws a von Koch fractal snowflake [62]. Figure 3.1 shows the final design, created by recursively repeating the adjacent motif. 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 will recursively define the fractal.

This motif requires two helper functions not provided in the standard library, so our work proceeds in four phases:

(a) We construct a helper function that, given two points, computes a point $\frac{1}{3}$ of the way between them (Figure 3.1a). This function, reused backwards, will produce the $\frac{2}{3}$ point.

(b) We will create a function that, given two points, computes a third point that completes the equilateral triangle with its two input points (Figure 3.1b).

(c) We use these two helpers to build a function that creates the motif *and* recursively repeats the motif within itself. This forms the points of a Koch curve, *i.e.*, one side of the final snowflake (Figure 3.1c).

(d) We complete the snowflake by laying out three instances of Koch curve points along the sides of an equilateral triangle and then attaching a polygon to the points.

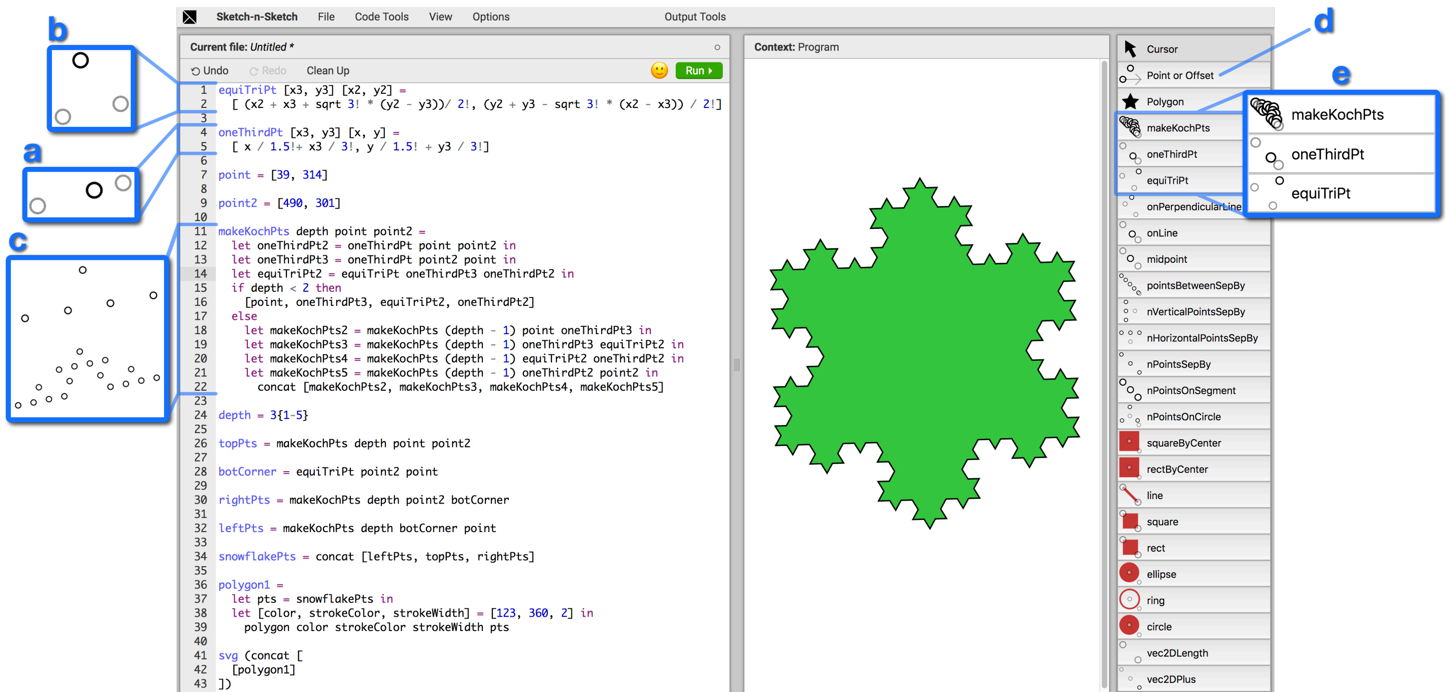


Figure 3.1: A Koch snowflake fractal constructed entirely via output-directed programming.

3.1.1 *One-Third Point Function*

Tools Highlighted: Draw Point, Relate, Abstract, Rename in Output, Delete
UI Features Highlighted: *Toolbox, Point Widgets, Call Widgets, Output Tools Panel*

First, we need to construct a helper function which takes two points and returns a point $\frac{1}{3}$ of the way between them. The initial program template provided by Sketch-n-Sketch is almost entirely blank, defining only an empty list of SVG shapes:

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

We will build this first helper function by demonstration; to do so we, need example points to work with. In this work, we extended Sketch-n-Sketch with drawing tools for inserting new definitions into the program. These tools are visible in the *toolbox* on the right side of Figure 3.1; a number of these tools will be demonstrated throughout this and the following examples. In this case, we select the "Point or Offset" tool from the toolbox (Figure 3.1d). When we click on the canvas to **Draw Point**, Sketch-n-Sketch inserts a new point definition at the top of the program:

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

svg (concat [
])
```

Although the new `x`, `y`, and `point` variables are not yet used—Sketch-n-Sketch has not changed the empty shape list at the end of our program—a point appears on our canvas at $(87, 206)$. We modified Sketch-n-Sketch's evaluator to draw widgets on the canvas when certain types of values are encountered during execution of the program. In this case, even though the point is not part of our program's output, whenever the evaluator encounters a number-number pair during execution a *point widget* corresponding to that coordinate is emitted as a benign side effect. Point widgets allow points in the program to be selected or manipulated on the canvas, even if a point is only an intermediate product during execution and does not occur in the program's final output. We will see examples of widgets for other intermediate execution products later.

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. The constant numbers in our program for the points' coordinates are updated accordingly via Sketch-n-Sketch's pre-existing live synchronization mechanism [12], as discussed in § 1.4.

```
[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. Like traditional GUIs, Sketch-n-Sketch supports selecting items on the canvas, either by clicking the items or by dragging a selection box around the items to select. We drag-select the three points on the canvas. Making a selection causes a floating "Output Tools" panel to appear, offering possible actions to transform the program based on our selection. This Output Tools panel is depicted in Figure 3.2. (It is worth noting that all the tools shown in Figure 3.2 are novel to this work; Chapter 4 provides tool descriptions, though most tools will be demonstrated throughout this and the following examples.)

In our case, we hover the mouse over the **Relate** tool. On hover, 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; specifically, the math inserted in place of a numeric constant must evaluate to within 20% of the original constant's value *and* the distances to the the other selected values must

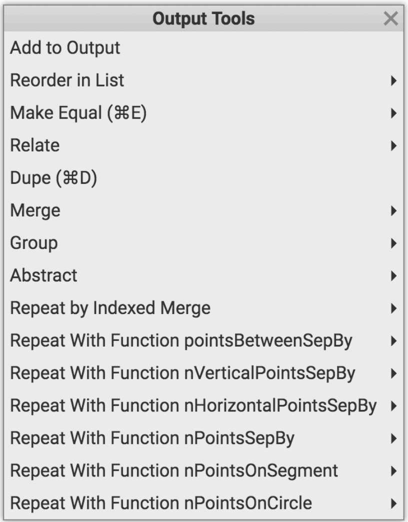
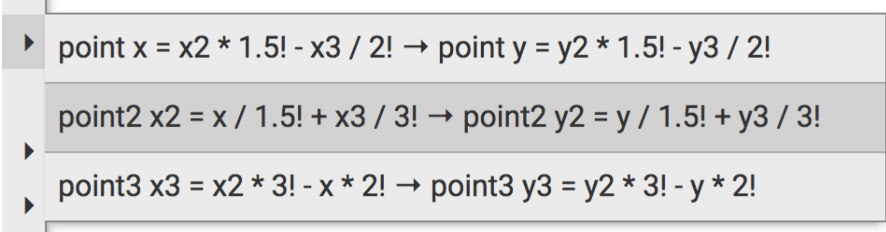


Figure 3.2: The Output Tools floating panel appears when some item on the canvas is selected. Various program transformations are offered.

not change more than 20%.[1] After about 20 seconds of guess-and-check work, three possible results are shown in the RELATE tool submenu.[2]



The RELATE tool synthesized three results—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 first hover over the result—which previews the new program in the code box and the new output on the canvas—and then click to choose that result.

```
[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!]
...
```

1. Note that x coordinates and y coordinates are related separately, so the 20% constraint is 1D distance along each single coordinate. The only constraint between the x and y coordinates is that the arithmetic term must be identical for both directions, modulo variable names.

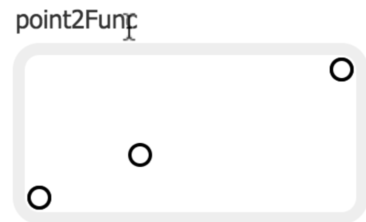
2. The *freeze annotations* on `1.5!` and `3!` indicate that these numbers should not be changed during live synchronization [12].

The `point2` definition has been rewritten using the synthesized arithmetic terms. The `point` definition, which previously was the last definition, has been moved upward so its `x` and `y` variables are in scope for `point2`.

To turn this concrete expression into a reusable function, we select all the points again and invoke **Abstract** from the Output Tools panel. The Abstract tool attempts to abstract *some* expression within our selection over that expression's free variables. In our case, there are three possible results, which abstract 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. If we hover the point emitted by this function on the canvas, a gray outline appears labeled with the text `point2Func`. The gray box is a *call widget*—emitted by the evaluator whenever a user-defined function is called in the program—indicating that the point was produced by a call to `point2Func`.



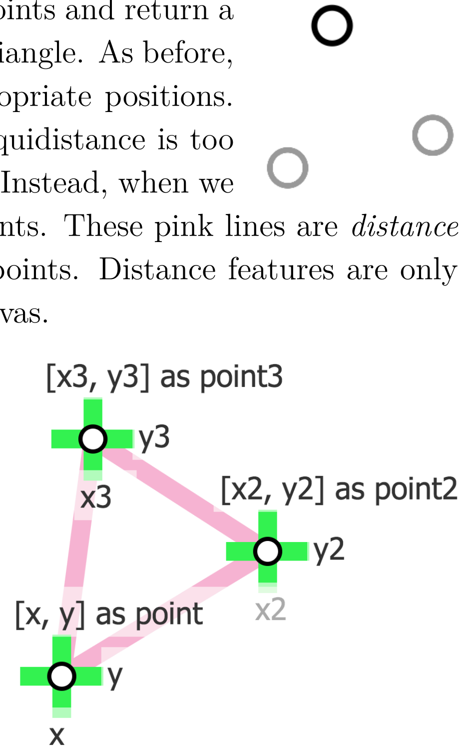
A name label on any widget may be clicked to **Rename in Output**. We thus click `point2Func`, type `oneThirdPt` into the text box that appears (not shown in the above screenshot), and hit Return. The function definition and its one use are appropriately refactored to be named `oneThirdPt`.

When we created this function, it also became a new drawing tool in our toolbox (Figure 3.1e). Using type inference, Sketch-n-Sketch realized that our function takes two points and could therefore be drawn with the mouse. We will use this new tool later.

Our first helper function is now complete, so we no longer need the example points used to build the function. We select all three points and press the Delete key to **Delete**. Perhaps surprisingly, Delete only removes the definition for `point2` 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 Chapter 5. 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.

3.1.2 Equi-Tri Point Function

Tools Highlighted: Make Equal
UI Features Highlighted: *Distance Features*



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, when we select the three points, pink lines appear between the points. These pink lines are *distance features* that, when clicked, select the distance between points. Distance features are only shown between selected points to avoid cluttering the canvas.

We click the three pink lines to select the distances and invoke the **Make Equal** tool, which queries an external solver (REDUCE [21]) 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, about which we defer discussion until our second 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 resuable function, and Rename in Output on the call widget label to name our function `equiTriPt`. Again, based on its inferred type signature our new function appears in the toolbox (Figure 3.1e). 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!]
...
```

3.1.3 Recursive Koch Curve Points Function

Tools Highlighted: DRAW FUNCTION, ADD TO OUTPUT, REORDER IN LIST
UI Features Highlighted: *Snap-Drawing, Call Focusing, Focused Drawing, List Widgets*

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. This **DRAW FUNCTION** action inserts the following definition into our program, calling our helper with two new points:

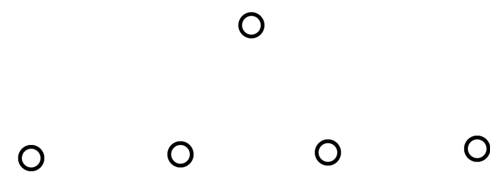
```
oneThirdPt2 = oneThirdPt [65, 199] [235, 141]
```

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* [18] so the existing endpoints are reused instead of adding new points—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). Using RENAME IN OUTPUT, we name the function `makeKochPts`.

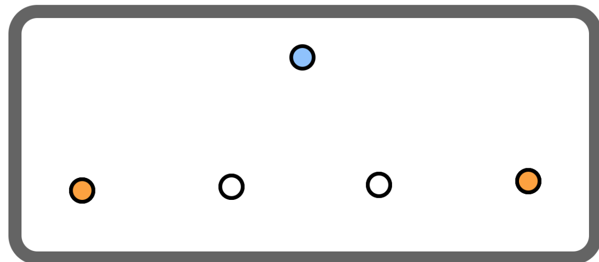
Now, we want to repeat the motif inside itself. If we select `makeKochPts` from the toolbox and draw it on the canvas, SKETCH-N-SKETCH will just insert another call at the top level of the program. Instead, we need the function to call itself *recursively*.

In this work, we added the ability to *focus* on a particular definition. Setting focus on a definition has two effects: (a) the canvas only displays the output of the focused definition, and (b) drawing operations add code to the focused definition rather than to the top level of the program.

We focus `makeKochPts` by clicking on the gray call widget border. The focused function, and the example call which provide example arguments for its execution and display, are then specified by special comments automatically inserted in 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
...
```

On the canvas, the focused function displays its arguments, with buttons for removing or reordering each. The inputs of our function are colored orange, while the outputs are colored blue. With the function focused, we can draw it 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.



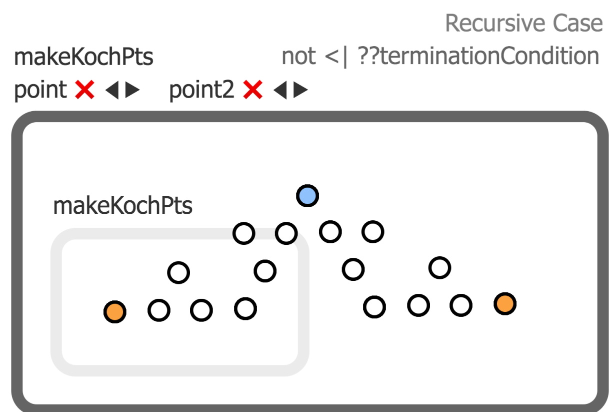

```
...
makeKochPts point point2 =
  let oneThirdPt2 = oneThirdPt point point2 in
  let oneThirdPt3 = oneThirdPt point2 point in
  let equiTriPt2 = equiTriPt oneThirdPt3 oneThirdPt2 in
  if ??terminationCondition then
    equiTriPt2
  else
    let makeKochPts2 = makeKochPts point oneThirdPt3 in
    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

To avoid infinite recursion, the `if-then-else` skeleton branches on a specially named hole, `??terminationCondition`. During evaluation, `??terminationCondition` returns `False`

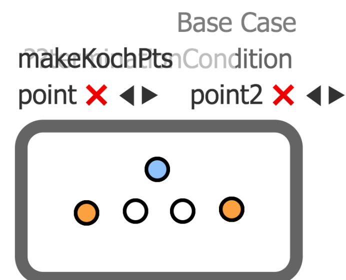
the first time the function is encountered in the call stack, and `True` if the function appears earlier in the call stack, affecting termination at a fixed depth of two. This allows us to run the program and manipulate its output even before we replace the hole expression later.

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.

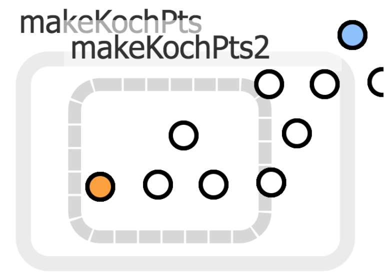
We click-select the three additional points we would like to be in the output,[3] and then choose **Add to Output** from the Output Tools menu. In order for the function to output multiple points, the function must now return a *list* of points. The return expressions of both branches of our function are therefore wrapped in lists, and the three selected points are added to the list in the base case.

3. Click-select instead of drag-select because, often, multiple point widgets are created exactly on top of each other. Drag-select will silently also select these covered points!

```
...
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.



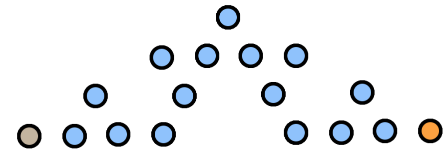
The recursive case currently only returns `[equiTriPt2]`; instead, we need it to combine all the point lists returned from the recursive calls. To select the *lists* returned from the base cases, we hover one of the points of those lists, which reveals a *list widget*—a list widget is emitted by the evaluator whenever a program expression evaluates to a list of graphical elements. The list widget has a dashed gray border that we may select. 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]
```

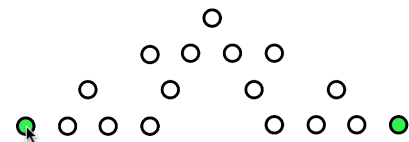


All the points are now in the output of `makeKochPts` (and therefore displayed in blue). One last item remains: we must choose a termination condition. The focused call widget for `makeKochPts` displays the conditional for the recursive case as `not <| ??terminationCondition` which we can click to **CHOOSE 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. These changes are visible in the final code Figure 3.1. 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 depth [12].

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

3.1.4 Koch Snowflake Polygon

Tools Highlighted: GROUP, ATTACH POLYGON TO POINTS



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* our 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 and RENAME the resulting group to `snowflakePts`. To finish the design, we choose the "Polygon" tool from the toolbox and click the `snowflakePts` list widget, which affects **ATTACH POLYGON TO 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, and then perform a bit of renaming to result in code shown in Figure 3.1. We grab the single `depth` slider, set the `depth` to three, and find the menu option to hide all the widgets. We're done!

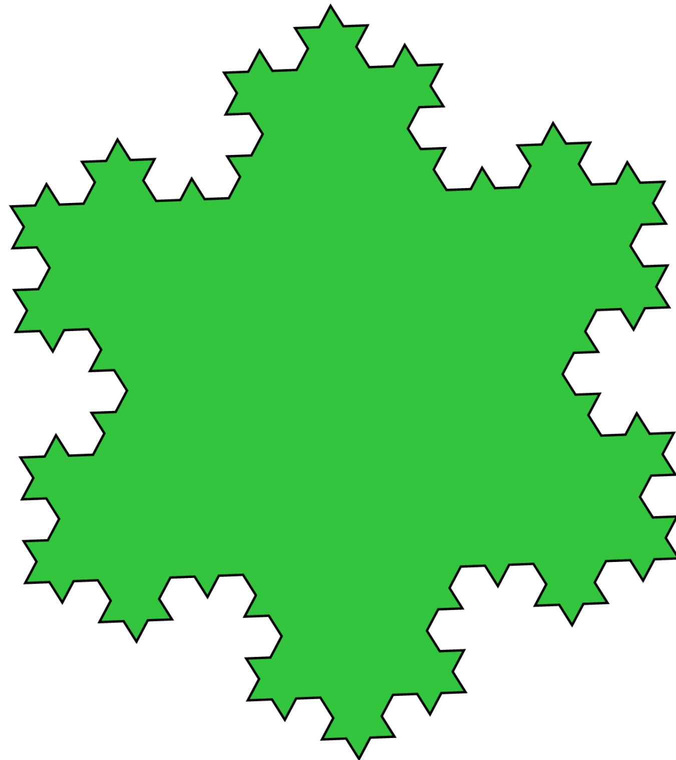
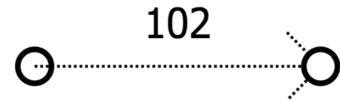


Figure 3.3: The completed Koch snowflake fractal.

3.2 Example 2: Tree Branch

Tools Highlighted: Draw Offset, Repeat Over List
UI Features Highlighted: *Offset Widgets*

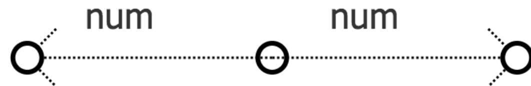
To highlight the new *offset widgets* and the *repetition tools*, we construct the tree branch design shown in Figure 3.4 on the next page. The construction involves a rhombus abstracted over its center point, which is then repeated over a list of points we draw in the program. For this example and the following, we omit mentioning uses of Rename in Output.



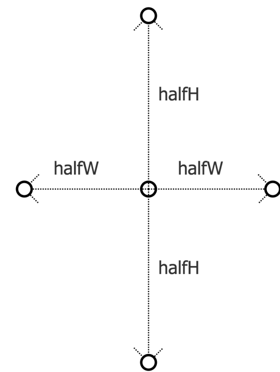
We construct the rhombus around a central point using *offsets*, which are simple additions to or subtractions from an x or y coordinate. The "Point or Offset" tool (Figure 3.1d) affects **Draw Offset** when *dragged* on the canvas, inserting an addition or subtraction operation, *e.g.*, `xOffset = x + 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:

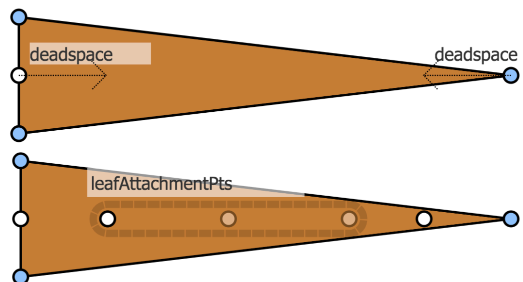
```
...
num = 102

xOffset = x + num

xOffset2 = x - num
...
```



We leverage this amount snapping to quickly create the skeleton of the leaf rhombus, as shown at right. We then draw a polygon, snapping to each offset endpoint, and Abstract the resulting shape into a function parameterized over `[x, y]`, `halfW`, and `halfH`. We will attach instances of this function over the branch.

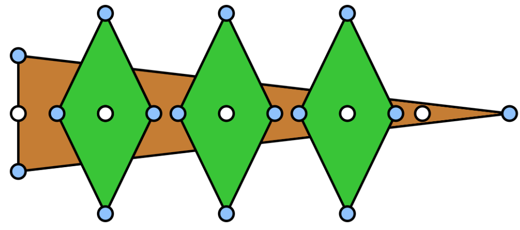


The branch is also constructed with offsets, so that it forms an axis-aligned isosceles triangle. We place two additional "deadspace" offsets inward from the ends of the branch to form the start and end of the attachment points for the leaves.



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.

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

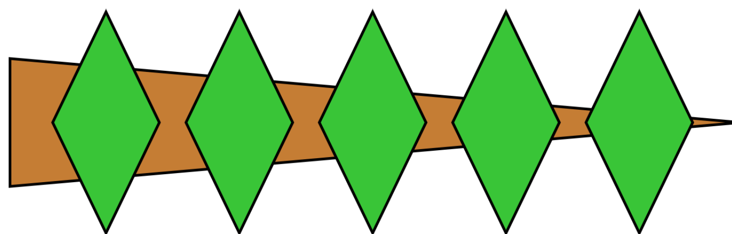


Figure 3.4: The completed tree branch design, utilizing *offset widgets* and *repetition* in its construction. Shown with widgets hidden.

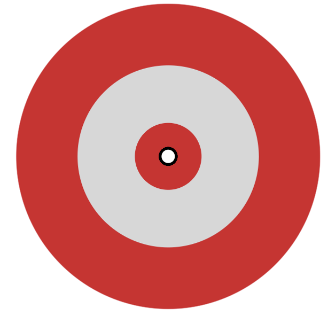
3.3 Example 3: Target

Tools Highlighted: Repeat by Indexed Merge, Fill PBE Hole

Repeating over a point list allows copies of shapes to vary only in their spacial positions. To support other repetitions scenarios where the varying attributes could be calculated from an index (*i.e.*, $0, 1, 2, \ldots$), we offer 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` always for the smallest circle).

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



The program maps an anonymous function that takes an index `(\i -> ...)` over the list `[2,1,0]`. Each index is thus transformed into one of our circles. The anonymous function contains an expression form by merging our original three circle definitions. The syntactic differences between the three original circle expressions—radius and color—have been turned into what we call *programming-by-example (PBE) holes*, represented by `??(...)`. 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 [55] 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.

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

For the first hole, we choose the `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, `zeroTo 3{0-15}`, contains a range annotation which exposes a slider on the canvas. 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 3.5: The final target design, constructed by filling PBE holes produced with Repeat by Indexed Merge. Shown with widgets hidden.

CHAPTER 4
TOOLS

In this chapter, we catalog the output-directed programming tools and associated UI elements we added to SKETCH-N-SKETCH. Below, we divide the tools into four categories, tools to *draw* new items into the program, tools to *relate* features of existing items, tools to *group, abstract,* and *repeat* shapes, and tools to *refactor* the program from the output.

4.1 DRAW

4.1.1 Drawing Shapes

Shape drawing tools insert a new definition into the program and insert a usage of the new variable in a location such that the shape appears in the output. 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 the shape list. 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).

4.1.2 Custom Drawing Functions

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 3.1 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. In § 5.5, we discuss the role inference used to determine if a numeric argument is a distance.

4.1.3 Focused Drawing

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, as in the Koch example (§ 3.1.3). Or, if a selected item can be interpreted as a coming from the right hand side of a `let`-binding, then a FOCUS DEFINITION tool is displayed in the output tools. Finally, an enterprising programmer may text-edit special comments into their code directly.

If a function or a shape group is focused, drawing actions will attempt to add shapes to that function or group rather than the main shape list. As shown in the Koch example, focused editing enables recursive drawing.

4.1.4 Draw Point

In response to a single click on the canvas, the DRAW POINT OR OFFSET tool inserts a `[x, y] as point = [123, 456]` binding at the top of the drawing context. To expose these points for manipulation, a point widget is produced during evaluation whenever a numeric pair is encountered in the program.

4.1.5 Draw Offset

In response to a mouse drag on the canvas, the DRAW POINT OR OFFSET tool inserts a `offset = existingCoordinate + 123` binding into drawing context. The tool also inserts a new point if the offset was not drawn from an existing point. Offset widgets are produced during evaluation when a number tagged as an x or y coordinate is added to some number. These tags originate from the evaluation of pair (`[e1, e2]`) expressions—if both elements are numbers, each is tagged as an x or y coordinate and is also tagged with the sister coordinate to allow any offset widgets later produced to be drawn in the appropriate place on the canvas.

4.1.6 Dupe

The DUPLICATE tool examines the selected value's provenance to find an expression in the program (see § 5.2.2) which is then duplicated into a new binding and (potentially) added to the shape list.

4.1.7 Delete

Pressing the "Delete" key interprets the selected values into expression(s) (see § 5.2.2) and attempts to remove them. If the last variable usage of a binding is removed, the binding is also removed.

4.2 RELATE

4.2.1 Make Equal

The MAKE EQUAL tool replaces constant(s) in the program with an expression such that the selected items are always numerically equal. The numeric traces of the selected numbers [12] are rearranged into a system of algebraic equations over the constants in the program. Based on the solution(s) to the equations (as determined by an external solver [21]), one of the constants is removed and replaced with a mathematical expression using the other constants. In the process, some of the constants may be pulled out into variables in a higher scope in the program so all the constants are visible at the replaced expression.

4.2.2 Relate

The RELATE tool finds an arithmetic expression (by guess-and-check) that defines one of the selected constants in terms of the others. The redefined value must be similar to the original literal value for the result to be shown to the programmer. Specifically, the inserted math must evaluate to within 20% of the replaced constant's original value and the distances to the the other selected values must not change more than 20%. If both x and y coordinates were selected, the x coordinates and y coordinates are each related separately. Additionally, except for variable names, the x and y dimensions must use an identical arithmetic expression.

Like the MAKE EQUAL tool, RELATE operates in terms of the constants in the program.

4.2.3 Distance Features

Distances between selected points are exposed for selection, appearing as pink lines. A selected distance is interpreted as $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where x_1, x_2, y_1, y_2 are each numeric traces of the features.

4.3 GROUP, ABSTRACT, REPEAT

4.3.1 Group

The GROUP tool means simply "gather into list," and so 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 processes uses value holes—discussed in § 5.3—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 item in the new list that is a variable usage, in a guess-and-check fashion 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.

4.3.2 List Widgets

List widgets, drawn as dotted borders on the canvas as shown near the end of § 3.1.3, enable the selection and renaming of lists, and thereby also enable the selection and renaming of groups of shapes (*i.e.*, shape lists). List widgets are generated on every evaluation step that (a) produces a list and (b) is an expression in the program. To avoid cluttering the canvas, each list widget is hidden until the programmer mouses over a value or a sub-value of a value in the list.

4.3.3 Abstract

The ABSTRACT tool builds a function that returns the selected expression. 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 in order 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).

Although the Juno [42] and Juno-2 [24] constraint-oriented programing systems rely on non-standard execution semantics as discussed in § 2.3, these two systems 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.

4.3.4 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, ant the original selected expressions are each replaced by a call to the new function with appropriate arguments.

4.3.5 Repeat over Function Call

The REPEAT OVER FUNCTION CALL tool builds an abstraction `f` over some point in the selected shape such that, given a point, `f` returns copy of the shape. To repeat that shape across several points, a call `map f (pointsFunc ...)` is inserted into the program, where `pointsFunc` is a standard library function or program function that returns a list of points.

4.3.6 Repeat over Existing List

The REPEAT OVER EXISTING LIST tool operates the same as REPEAT OVER FUNCTION CALL, but repeats the shape across an existing point list in the program instead of over a new function call.

4.3.7 Repeat by Indexed Merge

The REPEAT BY INDEXED MERGE tool, demonstrated by the Target example (§3.3), syntactically merges the selected expressions with the differences replaced by PBE holes, detailed in § 5.4. A skeleton `map (\i -> merged) (zeroTo` *n*`{0-`*3n*`})` is inserted into the program,

where *n* 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.

4.3.8 Fill PBE hole

When a program contains PBE holes produced by Repeat by Indexed Merge, the Output Tools panel offers replacements for each PBE hole in the program, using a simple sketch-based synthesis [55] technique described below in § 5.4.

4.4 REFACTOR

4.4.1 Rename in Output

To aid program comprehension, most widgets on the canvas are labeled with an associated program expression. If the associated expression is a variable usage or the left hand side of a binding, the pattern for the binding (the right side of the binding) is displayed and may be clicked to expose a text box to rename the variable(s) in the pattern.

4.4.2 Add / Remove / Reorder Argument

When a function call is focused, its arguments are shown with buttons next to each to remove or reorder the argument. 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.

4.4.3 Reorder in List

The Reorder in List tool finds the list literals in which the selected value participates and offers four options to move the selected item within each list: the selected item may be moved to the list head, to the list end, one space headwards, or one space endwards. Reorder in List may be used to affect the "Move to front," "Move to back," etc. shape reordering actions of a traditional graphics editor (albeit backwards—the front of a shape list is the shape drawn at the bottom).

4.4.4 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 logic for determining where the new variable usage should be placed.

4.4.5 Select Termination Condition

Drawing a function inside itself introduces a recursive skeleton which includes a hole, displayed as `??terminationCondition`, for the as-yet-undefined termination condition for the recursion, as demonstrated in the Koch snowflake example (§ 3.1.3). The Select Termination Condition tool offers fillings for the conditional hole. Currently, only one filling is supported: fixed depth termination. When chosen, a `depth` argument is added to the function, the `??terminationCondition` hole is replaced by `depth < 2`, and the `depth` is decremented for any recursive calls.

CHAPTER 5
IMPLEMENTATION

5.1 Introduction

SKETCH-N-SKETCH is an in-browser app written in the functional language Elm [15], with exceptions and mutation added via a custom library.

Below, we discuss several of the key technical mechanisms that facilitate the operation of the new tools: value *provenance* for deducing which program expressions to change, *value holes* for snap-drawing, *PBE holes* for the REPEAT BY INDEXED MERGE workflow, and *roles* for determining how to draw programmer-defined functions.

5.2 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.

Previous versions of SKETCH-N-SKETCH tagged numeric values with control-flow ignoring traces of the mathematical operations performed on them, enabling live synchronization [12] of numeric literals in the program as the user moved shapes on the canvas (as discussed in §1.4). We inherit these numeric traces and use them for MAKE EQUAL, RELATE, and when a drawing snap cannot be resolved by using or introducing a variable (as a last resort we directly insert the math recorded by the trace). Numeric traces only record the provenance of numeric values, however.

In this work, we extend SKETCH-N-SKETCH's provenance tracking to meet two goals. First, we would like to be able to trace all kinds of values—not just numbers—back to relevant program expressions. Second, we would like to retain enough information about execution that we can trace values back to relevant program expressions *within a particular editing context*. The naive interpretation of a value may not be an expression within the portion of the code the programmer is editing, so we would like to be able to find multiple interpretations of a value in order to discover expressions within the portion of the code the programmer cares about.

To meet these goals, we record the origin of 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.

$$
\begin{array}{rclll}
\textbf{Expressions} & e & ::= & c \mid x \mid e_1 \oplus e_2 \mid \lambda x.\, e \mid e_1\, e_2 \\
 & & \mid & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
 & & \mid & (e_1, e_2) \mid \texttt{fst}(e) \mid \texttt{snd}(e) \\
\textbf{Contants} & c & ::= & n \mid b \\
\textbf{Detagged Values} & v & ::= & c \mid \langle E, \lambda x.\, e \rangle \mid (\widehat{v}_1, \widehat{v}_2) \\
\textbf{Tagged Values} & \widehat{v} & ::= & v^t \\
\textbf{Tags} & t & ::= & \langle e, \{\widehat{v}_1, \ldots, \widehat{v}_n\} \rangle \\
\textbf{Environments} & E & ::= & - \mid E, x \mapsto \widehat{v}
\end{array}
$$

Figure 5.1: 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 5.2.

5.2.1 "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 he 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 gross 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.[1] 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, instead of standard big-step semantics, we employ a tagging evaluation relation $E \vdash e \Downarrow v^t$, where t is a tag of the form $\langle e, \{v_1^{t1}, \ldots, v_n^{tn}\} \rangle$ which records the expression that produced the value as well as immediate values used in that production. We call this "Based On" provenance.

1. Philosophically, 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.

$$\frac{E \vdash e \Downarrow v, \{\widehat{v}_1, \ldots, \widehat{v}_n\}}{E \vdash e \Downarrow\!\!\Downarrow v^{\langle e, \{\widehat{v}_1, \ldots, \widehat{v}_n\}\rangle}}$$

$$\frac{}{E \vdash c \Downarrow c, \{\}} \qquad \frac{E(x) = \widehat{v}}{E \vdash x \Downarrow v, \{\widehat{v}\}}$$

$$\frac{E \vdash e_1 \Downarrow\!\!\Downarrow \widehat{v}_1 \quad E \vdash e_2 \Downarrow\!\!\Downarrow \widehat{v}_2 \quad v = v_1 \oplus v_2}{E \vdash e_1 \oplus e_2 \Downarrow v, \{\widehat{v}_1, \widehat{v}_2\}}$$

$$\frac{}{E \vdash \lambda x.\, e_f \Downarrow \langle E, \lambda x.\, e_f \rangle, \{\}} \qquad \frac{E \vdash e_1 \Downarrow\!\!\Downarrow \langle E', \lambda x.\, e_f \rangle^{t_1} \quad E \vdash e_2 \Downarrow\!\!\Downarrow \widehat{v}_2 \quad E', x \mapsto \widehat{v}_2 \vdash e_f \Downarrow\!\!\Downarrow \widehat{v}_3}{E \vdash e_1\, e_2 \Downarrow v_3, \{\widehat{v}_3\}}$$

$$\frac{E \vdash e_1 \Downarrow\!\!\Downarrow \mathit{true}^{t_1} \quad e_2 \Downarrow\!\!\Downarrow \widehat{v}_2}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_2, \{\widehat{v}_2\}} \qquad \frac{E \vdash e_1 \Downarrow\!\!\Downarrow \mathit{false}^{t_1} \quad e_3 \Downarrow\!\!\Downarrow \widehat{v}_3}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_3, \{\widehat{v}_3\}}$$

$$\frac{E \vdash e_1 \Downarrow\!\!\Downarrow \widehat{v}_1 \quad E \vdash e_2 \Downarrow\!\!\Downarrow \widehat{v}_2}{E \vdash (e_1, e_2) \Downarrow (\widehat{v}_1, \widehat{v}_2), \{\widehat{v}_1, \widehat{v}_2\}} \qquad \frac{E \vdash e \Downarrow\!\!\Downarrow (\widehat{v}_1, \widehat{v}_2)^t}{E \vdash \texttt{fst}(e) \Downarrow v_1, \{\widehat{v}_1\}} \qquad \frac{E \vdash e \Downarrow\!\!\Downarrow (\widehat{v}_1, \widehat{v}_2)^t}{E \vdash \texttt{snd}(e) \Downarrow v_2, \{\widehat{v}_2\}}$$

Figure 5.2: 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 $\widehat{v} = v^t$).

Figure 5.2 presents "Based On" tagging for a core language (Figure 5.1), with $\widehat{v} = v^t$. For brevity, the presentation splits evaluation into two mutually recursive relations: An augmented big-step semantics $E \vdash e \Downarrow v, \{\widehat{v}_1, \ldots, \widehat{v}_n\}$ which produces a detagged value v as well as the set of values $\{\widehat{v}_1, \ldots, \widehat{v}_n\}$ that v is immediately "Based On," and the relation $E \vdash e \Downarrow\!\!\Downarrow \widehat{v}$ that completes the tagging by gathering the immediate expression and the "Based On" values into a tag $\langle e, \{\widehat{v}_1, \ldots, \widehat{v}_n\}\rangle$ that is placed onto the value.

The $E \vdash e \Downarrow v, \{\widehat{v}_1, \ldots, \widehat{v}_n\}$ relation in Figure 5.2 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 other branch nor 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 container (via `fst`() or `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 5.2).

Since the "Based On" provenance of a value may be followed transitively, it can reveal a large number of program expressions involved in the production of the final value. The programmer is likely only interested in one or two of these expressions. In the next section, we discuss how we take a selected value and narrow down its "Based On" provenance to only a small set of expressions in the program.

5.2.2 Interpreting "Based On" provenance

"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 the tools are interested in expressions rather than values, so will we now detail how we use the "Based On" provenance to answer our main query: "For a particular value, what *expression(s)* in the program does it most likely refer to?"

After the user selects a value on the canvas, the simplest approach to selecting a relevant expression is to look at the tag on the selected value and take the single expression e in the tag—the expression that immediately produced that value—and consider that single expression e to be the *interpretation* of the canvas selection.

In practice, this naive approach usually finds the expression the programmer intended. Nevertheless, Sketch-n-Sketch occasionally needs to discover expressions earlier in execution, if, *e.g.*, the immediate expression is actually in the standard library and we instead need an expression in the program. Algorithm 1 shows the general schema for discovering interpretations further back in the execution history. Informally, a value may be interpreted as either (a) the single expression e that immediately produce the value, or (b) any concatenation of an interpretation each for all "Based On" values $\widehat{v}_1, \ldots, \widehat{v}_n$. For example, consider the following program:

```
let pt = (10, 20) in
pt
```

Assuming `let` desugars to function application, the final value produced is `(10, 20)`. The final value `(10, 20)` has three possible interpretations according to Algorithm 1: the final variable usage {`pt`}, the pair introduction {`(10, 20)`}, or the two constants {`10`, `20`}. Note that each interpretation is a *set* of expressions. The expressions `10` and `20` together constitute all the parts of the pair value `(10, 20)`, but simply `10` or `20` alone would not.

While Algorithm 1 is the basic mechanism for finding expressions based on a selected value, Sketch-n-Sketch does not follow the algorithm exactly. First, the programmer

Algorithm 1 Ways to interpret a value as expressions using "Based On" provenance. Returns multiple interpretations. Each interpretation is a set of expressions that together can be considered to "constitute" the value of interest. At a high level, the algorithm defines that a value may be interpreted as *either* the immediate expression e that produced the value *or* as a conglomerate of interpretations of the values $\widehat{v}_1, \ldots, \widehat{v}_n$ the value was immediately based on. $\prod$ below is set cartesian product.

function INTERPRETATIONS($v^{\langle e, \{\widehat{v}_1, \ldots, \widehat{v}_n\}\rangle}$) ▷ Returns sets of expressions.

$$\widehat{v}sInterpCombos \leftarrow \prod_{i=1}^{n} \text{INTERPRETATIONS}(\widehat{v}_i)$$

$$earlierInterps \leftarrow \left\{ \bigcup_{interp \in combo} interp \;\middle|\; combo \in \widehat{v}sInterpCombos \right\}$$

return $\Big\{\{e\}\Big\} \cup earlierInterps$

end function

is always working in some context and therefore expects their selections to only be interpreted inside that current context. In practice, this context is their visible program and the provided library is considered out-of-context. Consequently, Sketch-n-Sketch limits interpretations to expressions in the program. This in-context interpretation is accomplished by Algorithm 2, which especially curious readers may study (simply filtering the results of Algorithm 1 turns out to be undesirable). Second, discovering all possible interpretations is subject to combinatorial blow up. To avoid exponential blow up, we assume that expressions occurring later in execution are more likely to be the referent of the user's selection because later expressions are closer in form to the selected value. Consequently, in practice, the various tools often use a specialized variant of Algorithm 2 that returns at most one interpretation, the interpretation latest in execution. Finally, alongside these two considerations, the vagaries of particular tools call for other constraints on the interpretation. Below, we describe three such constraints and examples of where each constraint is used.

Single Expressions. Although a selected value may be interpreted as a set of expressions, many tools can only operate on a single expression. Consequently, Sketch-n-Sketch often employs a variant of Algorithm 2 specialized to find interpretations that consist only of a single expression. For example, the Reorder in List tool searches for an interpretation that is a single expression, more specifically a single expression that occurs inside a list literal. Similarly, to find an expression to duplicate, the Dupe tool searches for a single expression that also is not a variable usage.

Algorithm 2 Extension of Algorithm 1 to interpret a value inside an expression context specified by a predicate $\textsc{inContext?}(e)$. In practice, the context is usually the user-specified program, with the standard library considered out-of-context. Simple post-processing of Algorithm 1 would not produce desirable results: in cases where execution leaves and re-enters the context, an interpretation from Algorithm 1 might contain out-of-context expressions that should not be discarded but instead replaced by in-context expressions from further back in execution. The algorithm below handles that scenario as desired. Additionally, if any "Based On" $\widehat{v}_i$ cannot be interpreted in context—*i.e.*, was produced entirely by out-of-context expressions—that $\widehat{v}_i$ is discard. Otherwise, the cartesian product would collapse to the empty set. For example, consider the simple user-specified program `f 2` calling an out-of-context standard library function `f x = x + 1`. Without removing empty sets from $\widehat{v}sInterps$ below, the program's final value `3` can only be interpreted as {`f 2`}. Instead, the algorithm will ignore that `1` cannot be interpreted in context and will follow `x` back into the program to offer the interpretation {`2`} in addition to the interpretation {`f 2`}.

function $\textsc{contextInterps}(v^{\langle e, \{\widehat{v}_1, \ldots, \widehat{v}_n\}\rangle}, \textsc{inContext?})$

$$\widehat{v}sInterps \leftarrow \left\{ \textsc{contextInterps}(\widehat{v}_i, \textsc{inContext?}) \;\middle|\; \widehat{v}_i \in \{\widehat{v}_1, \ldots, \widehat{v}_n\} \right\}$$

$$\widehat{v}sInterps' \leftarrow \left\{ interps \;\middle|\; interps \in \widehat{v}sInterps,\ interps \neq \{\} \right\}$$

$$\widehat{v}sInterpCombos \leftarrow \prod_{i=1}^{n} \widehat{v}sInterps'$$

$$earlierInterps \leftarrow \left\{ \bigcup_{interp \in combo} interp \;\middle|\; combo \in \widehat{v}sInterpCombos \right\}$$

if $\textsc{inContext?}$(e) **then**

return $\left\{\{e\}\right\} \cup earlierInterps$

else

return $earlierInterps$

end if

end function

Unique, Late Expressions. If the programmer selects a shape and hits Delete, they are asking to delete the selected shape but, implicitly, all non-selected shapes should remain intact. Since a single 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 non-selected 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 non-selected shapes. Additionally, as later execution products are closer in form to the selected value, the tool uses only the unique interpretation occurring latest in execution. For each expression in this interpretation, Delete successively attempts to remove that expression from the program.

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 within that reachable expression set. Similarly, the Add Argument tool—available when a function is focused for editing and some item on the canvas has been selected—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. 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.

"Based On" provenance is Sketch-n-Sketch's primary means to interpret output selections, but is supplemented by "Parents" provenance, as described below.

5.2.3 "Parents" provenance

Recall from Figure 5.2 that containers are "Based On" their constituent values, but not vis 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 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 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.

5.2.4 Expression IDs

Within the core language of Figure 5.1 it is not possible to distinguish between structurally identical expressions that occur in different code locations—if a variable `x` occurs in the standard library, and the programmer also uses a variable named `x` in a different scope, if either of those `x`'s are executed, the e in the resulting value's "Based On" tag will simply be a bare `x` and it is impossible to know which `x` in the program or standard library is intended. 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. For example, the INCONTEXT? predicate given to Algorithm 2 examines the expression ID to determine if the expression is within the editing scope.

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. A single invocation from the programmer may internally run a series of 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. Many of these sorts of compositions require certain 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 convenience 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.

5.2.5 Approaches to Provenance in Other Work

Live programming environments attempt to continuously inform the programmer about their program's behavior even while the programmer makes edits, tightening the feedback loop. Though few live programming systems also attempt to make the output manipulable as we do here, the live programming system may attempt to aid program comprehension by allowing the programmer to ask where a particular piece of the output came from. We mention two recent examples. The live game programming environment JSDARE [48] keeps track of execution steps and creates a map from pixels on the canvas back to expressions in the program such that hovering over items on the canvas will highlight a relevant block of code. Similarly, the TouchDevelop live programming environment for creating UIs [8] allows the programmer to touch a UI box in the output to focus the code expression that emitted the UI box. Neither system allows direct manipulation of the output, nor do they present provenance tracking as part of the core semantics of the language.

Although provenance tracking has a long history in the domain of data management, several works have explored provenance tracking in the context of general-purpose programming languages. Cheney et al. [11] propose using entire operational derivation trees to record provenance, as this complete record of the computation can be viewed as the "most precise" [11] form of provenance possible. Using derivation trees for a non-higher-order imperative core language they demonstrate extracting where-provenance (that is, finding values copied from input to output [7]) and performing incremental re-computation.

Acar et al. [2] propose a generic annotation framework that allows various forms of provenance to be projected from rich traces recorded during the runtime of a core language, Transparent ML (TML). Common provenance schemes can be deduced by post-processing

the execution trace. TML traces, also utilized in [46], are nearly identical in structure to core language expressions but additionally record control flow, namely, the function body on function application and the branch taken on case statements. Compared to storing the entire derivation trees, TML traces discard intermediate environments and intermediate values. If provided with a final value, however, the traces are complete enough to allow computation to be rewound backward to recover intermediate values and partial environments. This *unevaluation* technique is used in Perera et al. [46] to build program slices and a variant is used in Acar et al. [2] for determining disclosure and obfuscation security properties.

The goal of our provenance techniques is to identify a program expression or small set of expressions most relevant to some selected sub-value of the output. The problem of *program slicing* is similar: given some criterion, produce a subset (a "slice") of the original program that preserves the criterion. Program slicing has a long history, with most approaches applied to imperative or object-oriented languages [66]. Originally envisioned as a static analysis technique [64], Korel and Laski [30] later proposed slicing a program with respect to a single input to produce smaller, more informative slices—a technique called *dynamic program slicing*. For functional languages, dynamic slicing with respect to some sub-value of the output is elegantly handled by the approach of Perera et al. [46] (later extended to also handle imperative constructs [50]). Leveraging a semantics that uses holes to represent ignored portions of the program and output, Perera et al. [46] are able to produce executable least slices for a given output value—that is, a smallest program that may still be executed and produce the desired sub-value of the output. Executable program slices must preserve significant portions of the program to remain runnable and thus are too large for our purposes—in most cases, we need to isolate a single expression before invoking a transform. To focus in on the most relevant portions of a slice, Perera et al. [46] also propose *differential slices*. By producing two program slices, one ignoring and one including the sub-value of interest, the difference between the two program slices may be examined to find the statements most relevant to sub-value of interest. Even with the small differential slice, the differential slice is still a set that does not given any indication of which expression in the set is most important. For this reason, the tracing mechanisms used to produce the slices (TML in the case of Perera et al. [46]) are more related to our purposes than programs slices themselves, as the traces retain execution order information which indicates the most relevant expression(s) (those latest in execution).

Similar to TML, redex trails [58] trace computation by storing pointers to prior reductions. Unlike TML, redex trails do not explicitly model pattern matching and thus add special indirection nodes into the trace graph in order to trace how values are taken out of containers. Instead, we find containers using a mutable tagging scheme ("Parents" provenance described above in § 5.2.3). The biggest difference between our provenance tracking and both redux trails and TML, however, is that at each execution step we consider the immediately prior values to be a set rather than recording each value's role in each reduction. This limits the amount of information we can glean about the execution from the

"Based On" traces, and for this reason future versions of Sketch-n-Sketch may switch to a strictly more informative tracing scheme like TML.

5.3 Value Holes

Drawing a new shape requires inserting a function call into the program. But, if any snaps were used in the drawing interaction, the inserted function will need to rely on existing expressions and may require reorganizing the program somewhat to bring needed values into scope. To elegantly separate the concern of where to place the function call from the concern of transforming the program to enforce snaps, we employ special expressions dubbed *value holes*. A value hole is a temporary expression that contains a value. To enforce a snap, the values to snap to are initially inserted into the program AST as value holes. All value holes are then resolved to traditional expressions before the new program is displayed to the programmer.

For example, if the programmer draws a rectangle with its top-left corner snapped to an existing point, the rectangle is temporarily inserted to the internal AST as…

```
rect1 = rect 0 [??vx, ??vy] 150 150
```

…where `??vx` is a value hole containing the x value to snap to, and `??vy` is a value hole containing the y value to snap to. A hole resolution phase examines the provenance of the x and y values in the holes to find a variable to use instead, or to bring a needed program expression into scope as a variable.

If a value hole is encountered during evaluation, the hole simply evaluates to its contained value. This evaluation rule is not just a gimmick: while the programmer is snap-drawing a new shape, the rule allows us to continuously redraw a preview of the function call's result on the canvas without having to perform the (potentially expensive) hole resolution.

Value holes are an elegant mechanism for combining template code with output-directed constraints. Thus, for example, the Group tool also employs value holes to trivially create a list containing the selected values.

5.4 Programming by Example Holes

To affect shape repetition through a programming by demonstration workflow, the Repeat by Indexed Merge tool utilizes another type of expression hole we dub *programming-by-example (PBE) holes*. A PBE hole, written `??(1 => e1, 2 => e2, ..., n => en)`, 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 of 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 hole fillings are currently generated based on a short list of built-in simple expression sketches [55] (*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 replaces the various terms in the sketch based on the examples in the PBE hole, perhaps dispatching the external mathematical solver [21], 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 repetitious 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 program synthesizer such as Myth [44].

Unlike value holes, PBE holes may appear in the programmer-visible code. But, like the termination condition hole, PBE holes are expected to eventually be filled.

5.5 Roles

To draw programmer-defined functions on the canvas, Sketch-n-Sketch needs to know which arguments are points, widths, or heights, and also needs to provide reasonable defaults for the remaining arguments. All number-number pairs are considered points and so points can be readily identified by type inference, but width, height, stroke width, etc. are all just numbers. To infer widths and heights we augment the type system to infer *roles* in addition to primitive types. Each type in Sketch-n-Sketch may be tagged with a set of named roles, which are propagated during unification. Roles are introduced in two ways: by the use of named type aliases in type annotations, or by various hard-coded structural rules. The standard library functions are already annotated with roles (via type aliases), so the use of such functions propagates role information into the program. The structural rules include our definition of point—a number-number pair is tagged with the `Point` role, its left number with the `X` role, its right with the `Y` role—as well as special propagation rules helpful when using offsets—*e.g.*, if an untagged number is added to an `X` coordinate, tag that number as a `HorizontalDistance`.

Although similar to dimension types for notating units of measures on numbers [29], roles are different in two key ways: roles need not apply only to numbers, and, as implemented,

roles play no part in checking program correctness. Dimension types will reject a program that attempts to add centimeters to inches, but roles are side information not used for type checking.

A scheme similar to roles, called *brands*, is used in APX [40] to offer semantically meaningful UIs for manipulating program expressions. Since APX also targets shape drawing programs, many of APX's brands are the same as Sketch-n-Sketch's roles: `X` and `Y` for coordinates, `Color` for colors, and so on. APX brands are realized using a multiple inheritance scheme with inference based on duck typing. We instead implement roles as side information on each type instead of as object oriented mixins. Also unlike brands, we apply usage rules—*e.g.*, a number added to an `X` value must be a `HorizontalDistance`—to infer roles in more cases.

CHAPTER 6
EVALUATION

To evaluate our approach, we have implemented a total of sixteen programs without text-based editing. Although a distinctive feature of output-directed programming is the ability to edit the text-based code at any time during the output-based construction, the flexibility of text-based programming is already well established. We therefore limit ourselves to constructing programs using only canvas-based interactions in order to to emphasize the expressive power of the novel set of output-directed tooling we present. Our main argument for the value of our contribution is an existence proof: SKETCH-N-SKETCH can create these programs without text edits.

The output of each of the sixteen examples is shown in Figure 6.1. We also present program sizes (measured in source lines of code and math operations) to provide a rough indication of the scope of programs our system is able to handle.

The example designs are taken from three main sources:

1. Six of our examples (underlined in Figure 6.1) are from the PBD test suite proposed in *Watch What I Do: Programming by Demonstration* [1]. The *WWID: PBD* suite is diverse: 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.

2. We take 5 tasks from other drawing literature [4, 10, 12].

3. We additionally implement 5 novel tasks.

The Koch fractal presented previously (§ 3.1) was a task in the *Watch What I Do: Programming by Demonstration* benchmark suite [1]. For the other five examples taken from the *WWID: PBD* benchmark suite, we give a brief discussion of their construction below.

6.1 Precision Floor Plan (Figure 6.1d)

The goal of this example is to draw a “table” rectangle exactly over the left third of a “floor” rectangle. We draw the floor rectangle, snap the top corner when we draw the “tabl’e’ rectangle, equalize the heights, and then invoke RELATE on the widths. (9 LOC including 3 math operations.)

6.2 Mondrian Arch (Figure 6.1e)

The goal in this task is to create an abstraction that draws an arch consisting of three rectangles. This arch task was used to present the Mondrian graphical editor’s [34] programming by demonstration workflow.

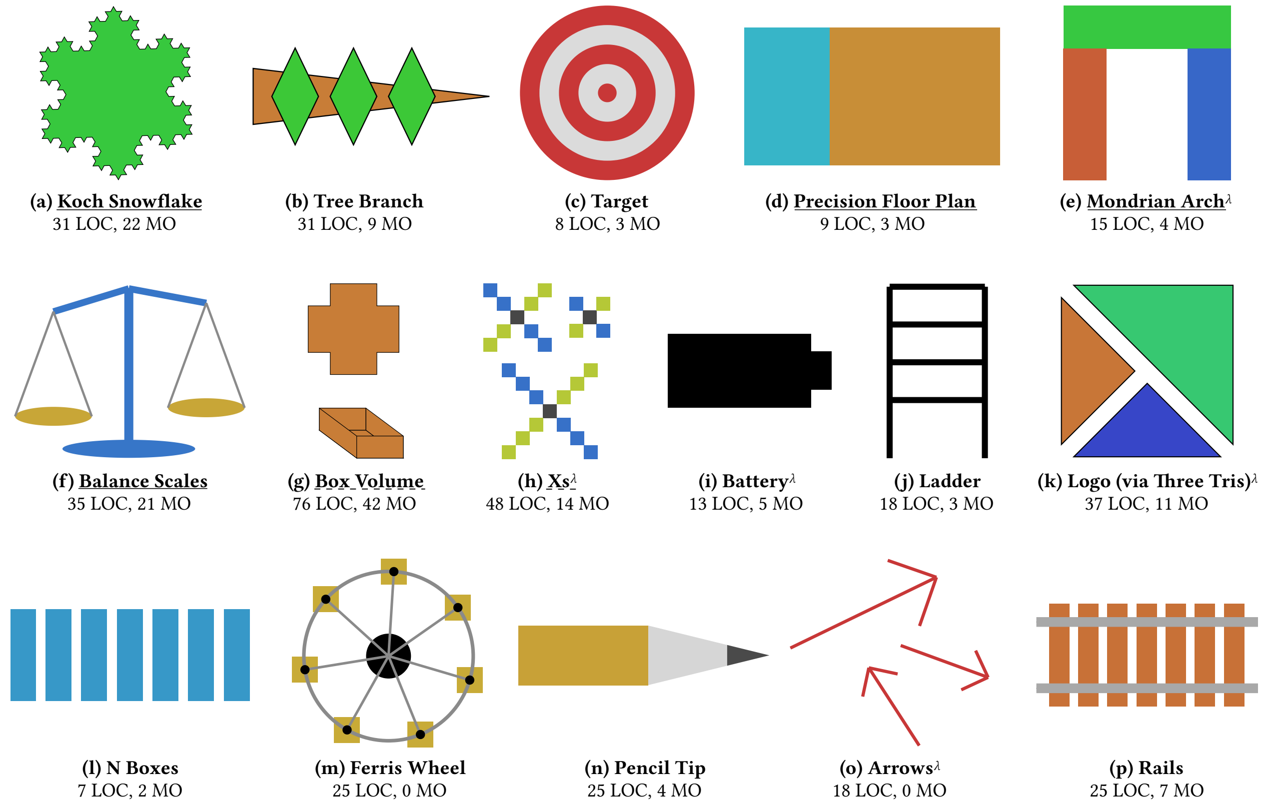


Figure 6.1: Examples programs created using only output-directed manipulations in Sketch-n-Sketch, with source lines of code (LOC) and number of math operations in the code (MO). λ = Final design is abstracted (*i.e.*, available as a drawable function). underline = Task from WWID: PDB test suite (dashed = only partially completed) [1]. Other sources: *Battery* is from Bernstein and Li [4], *Ladder* from Cheema et al. [10], *Logo (via Three Tris)*, *N Boxes*, and *Ferris Wheel* from Chugh et al. [12].

In this work, to end with the ideal parameterization of `[left, top] width height stoneWidth`, we start by drawing a point and offset downward to serve as the top left point and the overall height. If we permit ourselves to overlap the rectangles (as in the original presentation in Mondrian), the offset can be omitted and instead the height of a pillar rectangle serves as the overall height—but we consider overlapping shapes to be inelegant in this case as it does not match the physical reality of, *e.g.*, Stonehenge and thus forego this method. The remainder of the design is constructed by repeatedly utilizing Make Equal once all three rectangles have been drawn. We take advantage of the multiple resolution options provided by each Make Equal invocation to, for example, ensure that the right pillar is snapped to the right edge by deriving the right pillar's x position—rather than its width—in terms of other pre-existing parameters. Just before the Group and Abstract operations to finish the design, to make the overall height an argument for the abstraction we select the distance between the top left point of the top stone and the bottom left

point of the left pillar (recall that distance features are accessed by selecting the two points first and then the distance) and equalize that vertical distance with the offset amount we began the design with. MAKE EQUAL offers several options to enforce the constraint, we choose the option that preserves `stoneWidth` as a parameter and computes `pillarHeight` as `height - stoneWidth`. The offset may then be removed as the `top + height` operation that generated the offset is not needed as a value anywhere in the program—the operation was only transiently necessary to expose a widget to select `height`. A "Clean Up" button removes this dead code. From there the design is finished by grouping the three rectangles and abstracting the resulting group. If the arguments are in the wrong order so that drawing new instances of the function are awkward (e.g. the mouse's x movement controls `stoneWidth` instead of `width` because `stoneWidth` is before `width` in the argument list) then the arguments may be reordered by focusing the function call and utilizing the REORDER ARGUMENT buttons. The final program consists of 15 LOC and 4 math operations.

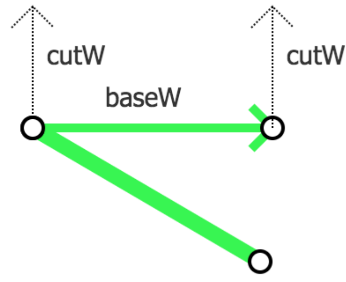
6.3 Balance Scales (Figure 6.1f)

In this example, the goal is to constrain the drawing so the hanging trays always stay attached to the arms as the arms rotate on the fulcrum. We want each arm to rotate independently but stay equal in length to each other. This is accomplished by placing points for the fulcrum and two arm ends and then use the distance features to equalize the distance of the two arm ends from the fulcrum. In the rest of the design, the key to ending with a nice parameterization is to first start by drawing a point and offset downward and then to build a tray and its two wires around this point and offset. The start point of the offset is where the wires will attach to the balance arm, and the offset end point is the center of the tray. GROUP (with dependencies gathered), DUPLICATE, and MERGE of the tray+wires creates a function abstracted over the attachment point of the tray (to optionally make the function drawable, the offset amount, that is, the tray hanging distance, must then be added as an additional argument). The rest of the design proceeds apace from there, for a final program of 35 LOC with 21 math operations.

6.4 Box Volume (Figure 6.1g)

Imagining a scenario where a student wants to explore how to maximize the volume of a box constructed by cutting out the corners from a flat square and folding up the sides, this design displays a top-down view of the cutout template synchronized with a pseudo-3D view of the resulting folded box. The main parameter is the side length of the square cut from each corner, *i.e.*, the height of the resulting folded box. The top-down view may be constructed by first laying out offsets for the overall width and height of the material and then drawing a number of offsets around the corners to lay out a skeleton of points to which a polygon will be attached. Amount snapping of the offsets while drawing makes equalizing

the appropriate lengths trivial. In this construction, the width of the base of the folded box (*i.e.*, $width_{total} - 2 \cdot width_{cut}$) is not immediately available on the drawing but will be needed to make the perspective view. To expose this distance, we draw an offset in the appropriate location on the top down view, and another in an equivalent location to introduce a variable for the distance. The end points of the offsets do not snap during drawing (only the amount), but the end point can be snapped after the fact with MAKE EQUAL, thus causing the amount to be derived as $width_{total} - 2 \cdot width_{cut}$.



Once in place, we can construct the perspective view. The back plate of the box can be laid out with offsets, snapping their amounts to the existing $width_{cut}$ and just-created offset amount of $width_{total} - 2 \cdot width_{cut}$. To construct the diagonal, we utilize the `onLine` library function which, given two points and a ratio, produces a point on the segment between the points with the ratio controlling its relative distance from the input points (*e.g.*, a ratio of 0.5 produces the midpoint). We draw `onLine` from the back corner of the box to make a diagonal with a point on it. To make that point's distance from the corner equal to the box base length, we invoke MAKE EQUAL on the appropriate distances (shown selected in green in the inset). MAKE EQUAL will cause the ratio to be calculated from the appropriate math to so that the point on the diagonal is always $width_{total} - 2 \cdot width_{cut}$ (that is, $width_{base}$) from the corner.

The rest of the design skeleton can be constructed with offsets, with a single four-sided polygon attached to each face. Thus drawn, resizing any of the $width_{cut}$ offsets changes both the top down template view and the shape of the folded box. This is as far as we can proceeding, resulting in a program of 76 LOC with 42 math operations.

To help the student know when the volume is maximized, a complete design should also provide textual display of the box volume and cutout length. We have not focused on any textual output and thus are not yet able to accommodate this part of the task.

6.5 Xs (WWID: PBD p. 591 David Maulsby)

Originally presented as a color change operation, here we interpret the Xs design as a parameterized abstraction that should draw an X with a specified number of squares on the arms. In our realization of the design, each arm is based on a separate call to the `nPointsSepBy` library function which takes a base point, a horizontal separation between points, a vertical separation between points, a number of points, and returns a list of points according to the parameters. To equalize the separation amounts to the square widths, we must expose offset widgets for math operations conducted outside the written program in the standard library. By default, since the calculations in the standard library are not visible in the code box, execution steps in the standard library do not produce widgets lest these widgets confuse the

programmer and clutter the canvas. In the "View" menu, the "Show Offset Widgets from Prelude" enables these offsets, which, in our case, allows us to select the amount of horizontal and vertical separation between the repeated points. Depending on the direction of the arm, we want each of the separations to be either $width_{square}$ or $-width_{square}$. For the those that should be $width_{square}$, we can just invoke MAKE EQUAL. For the $-width_{square}$ separations, we must lay out the offset at roughly the right distance, invoke RELATE, and choose the $0 - width_{square}$ option (our language currently lacks a negation operator). Unlike MAKE EQUAL, RELATE must be used separately on each separation offset with a $width_{square}$, as its semantics are to "relate one selected item in terms of all others." With the repetitions laid out, `rectByCenter` instances may be drawn and then attached to the points. From here we may GROUP all the parts of the X together into a single definition (this requires first grouping the center rectangle with just itself to produce a singleton list, so it can then be `concat`ed with the arm lists).

We now invoke the ABSTRACT tool to make a function that will draw an X. The GROUP and ABSTRACT tools deliberately do not gather helper functions into the new definitions they create, which in this design, unfortunately, leaves several instances of $width_{square}$ outside the control of the abstraction. One solution to this problem might be to optionally offer transform results that do gather helper functions; but as of this writing such a fix remains unimplemented and we must be content with only partially implementing the task. (48 LOC with 14 math operations).

6.6 Tackling the Remaining WWID: PBD Tasks

The remaining nine tasks in the *WWID: PBD* test suite are diverse. No single feature would help with any more than two or three of the tasks. Our most prominent missing feature is the ability to draw text boxes, with other elements placed relative to the text size. Beyond this, several examples require a wide variety of list operations. SKETCH-N-SKETCH would also need the ability to reason about intersections of lines with shape edges, to specify overlapping and containment-based constraints, and to solve for different kinds of such constraints simultaneously. Finally, one example would require SKETCH-N-SKETCH to create `if-then-else` branches outside of a recursive or hole-filling setting.

CHAPTER 7
DISCUSSION

We presented new techniques for output-directed programming in the SVG programming environment SKETCH-N-SKETCH, and we used the new system to construct a variety of non-trivial programs entirely through direct manipulation. Our system lacks many features—such as rotation attributes and curved path tools—that would be required in order for SKETCH-N-SKETCH to be a practical tool for creating parameterized drawings. We expect that many of these features would work in much the same way as the techniques presented.

Our goal has not, however, been to create the ideal system for parameterized drawings. Instead, we consider this particular application domain as a laboratory for developing output-directed programming capabilities for a variety of future programming settings. In this chapter, we reflect on lessons learned so far, and offer avenues for further research.

7.1 Lessons Learned

As just mentioned, the version of SKETCH-N-SKETCH we present is not intended to be a serious drawing tool. Instead, we view SKETCH-N-SKETCH as a workbench for exploring output-directed programming. What has this work revealed?

7.1.1 Is it working?

We start our discussion by addressing an obvious question—does the output-directed programming system described here *work*? If SKETCH-N-SKETCH isn't a serious drawing tool, what is its valuable research contribution?

This work is an existence proof that meaningful, human-readable programs can be produced entirely by output-directed interactions, at least for a single domain. This work only explores a single domain, but as mentioned at the beginning of §5.2, the core technical mechanisms in SKETCH-N-SKETCH—tagging values with provenance, interpreting the provenance for selected values, and emitting widgets during execution to expose intermediate values—are general-purpose and apply to any domain.

7.1.2 Intermediates

We believe that in order for output-directed programming to scale up to larger programs, more than just the final result of execution must be exposed for manipulation. Intermediate execution products, and some indication of how they were computed, must be displayed. The display of intermediates falls within the vision of live programming. For example, Seymour [28] shows intermediate code products to the side of each line of code. Our proposal in this work is to make these intermediate results *manipulable*. For the SVG domain, we

make intermediate results manipulable by exposing various automatically generated widgets on the canvas—point, offset, list, and call widgets. Point and offset widgets are graphics-specific, but, other than the computation of where they should be displayed on the canvas, list and call widgets are general-purpose.

Should an output-directed programming system attempt to visualize the entire execution of the program? The intermediates we have chosen to display still only represent a fraction of the program's execution. If the goal were to visualize the whole program, SKETCH-N-SKETCH could represent every code expression on the canvas, a la a visual programming language, or even display an entire execution trace to expose even more detail about the running of the program. More detailed display, however, requires more visual space, and thereby comes at the cost of increased visual noise. In the limit, display of all intermediates is infeasible: a 3.0 Ghz four-wide superscalar processor can produce 12,000,000,000 intermediates per second—they cannot all be displayed! Where to fall on this trade-off between more thorough computation visualization versus more economical use of canvas space is an open question.

Our final observation concerning intermediates regards our goal of producing human-readable programs. Names aid human program comprehension [33]. Thus, we believe the ability to RENAME IN OUTPUT is a boon for creating understandable programs.

7.1.3 Focused Contexts

When working on larger programs, programmers consider only small pieces of the whole program at a time. In light of this observation, we explored the ability to focus editing on a specific definition (usually a function), limiting display on the canvas to the output of the focused definition and the widgets produced therein.

In this work, focusing the editing context enables three interactions. First, it enables contextual drawing, that is, the ability to add a shape or widget inside an existing definition rather than at the top level of the program. Second, it enables two interactions that facilitate specification of recursive functions: the ability to draw a function in itself (as in [52]) and the ability to selectively edit the base or the recursive case by shifting editing focus between them (by selecting which call in the program is chosen as the contextual example for the focused function). Third, function arguments are hidden until a function is focused, allowing their display to be omitted until needed.

Beyond these interactions, we speculate that focusing the editing context might offer cognitive benefits to the programmer by decluttering the canvas. But the examples we have explored so far are too small to gain any intuition as to whether this hypothesis is true.

7.1.4 Programing by Demonstration or by Direct Manipulation?

A programming by demonstration (PBD) workflow is characterized by the programmer *showing* the editor what the programmer wishes to happen, and then the editor attempts to *infer* the programmer's intent. SKETCH-N-SKETCH's workflow for the RELATE tool and REPEAT BY INDEXED MERGE clearly fall into this PBD paradigm because these tools attempt to infer the programmer's intent based on where the programmer has placed the shapes on the canvas. Other than these two tools, however, we have largely opted for a more direct *select* and *tell*, rather than *show* and *infer*, workflow. The *select* and *tell* interaction is more similar to traditional direct manipulation tools, but this distinction between direct manipulation and programming by demonstration may not be clearcut in SKETCH-N-SKETCH. PBD's inference step is characteristically ambiguous. Often a PBD system must ask the programmer for clarification. So too must almost all of SKETCH-N-SKETCH's transforms—we ask the programmer to choose one of many transform results. For this reason, although we have attempted to create a set of tools that operate in small, direct, mostly unambiguous steps—with the expectation that small transformation steps will generalize to other domains—it may be appropriate to classify SKETCH-N-SKETCH's approach as both direct manipulation *and* programming by demonstration.

7.1.5 Could you hide the code?

Although in this work 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 during program construction. 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. On the canvas, however, we can 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 are not eager to hide the 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.

7.2 Limitations and Future Work

This work presented tools and techniques that enabled a number of designs to be programmed entirely by manipulations on the output canvas. Nevertheless, there are many opportunities for improvement on this work.

Below we discuss improvements specific to this SVG realization of output-directed programming, improvements that concern both SVG and general programming simultaneously, and general improvements not specific to SVG. SVG-specific concerns tend to involve the UI, and general concerns involve the core implementation.

7.2.1 SVG-Specific Improvements

Sketch-n-Sketch might benefit from a number of SVG-specific improvements. The end of Chapter 6 detailed features that Sketch-n-Sketch might need to create the remainder of the *Watch What I Do: Programming by Demonstration* benchmark suite [1], including support for text boxes as well as a comprehensive offering of list operations. A couple of other missing but desirable graphics-specific features are worth mentioning.

Sketch-n-Sketch currently lacks a path tool, but a path tool is needed for many real world designs. The mouse interaction to specify paths, however, is quite unique. To generalize that interaction, we might imagine drawing tools that, instead of taking, *e.g.*, two points, or a point and a pair of distances, take instead a sequence of mouse events on the canvas. Those mouse events could be partially evaluated into a programmer-defined template, generating an expression to be inserted in the program. At present, we leave the specifics of this generalized drawing interface to future work.

Beyond missing tools, even within the SVG tooling presented in this work many reasonable use cases remain to be implemented. Snaps (*i.e.*, automatic alignments while dragging, as in § 3.1.3) are only available when drawing new shapes, not during mouse manipulations after a shape has been drawn. Similarly, the programmer may only snap to point widgets or offset amounts. Snaps are a more convenient interface than manually selecting features and invoking Make Equal, thus it would be helpful if the programmer could also snap to (a) derived features, *e.g.*, the bottom right corner of a rectangle or the midpoint of a line, and (b) just x or just y coordinates, to perform vertical or horizontal alignment as in common shape editors such as PowerPoint.

Additionally, these derived shape features do appear on the canvas when a shape is hovered, but the math to determine their positions is still hard-coded inside Sketch-n-Sketch. Akin to type-directed API discovery (*e.g.*, [37, 45, 20, 17]), a more flexible implementation might instead look in the standard library and in the program for, *e.g.*, functions of type `Rect` → `Point`, apply all such functions to the selected shape, and show each resulting point as a derived feature on the shape. The main reason these derived features are still hard-coded is because there is extra semantic information associated with these features that

cannot be captured in their primitive numeric values. For example, if Sketch-n-Sketch discovered a function `width : Rect → Num`, that tells Sketch-n-Sketch that the function can convert a rectangle to some numeric feature of the rectangle, but it doesn't tell Sketch-n-Sketch *where* to display the line that the programmer might click to select that feature. Similarly, direct manipulation of the center left point on a rectangle's edge should change the rectangle's x position *and* width, but that information is not conveyed by the existence of a function `rectCLPoint : Rect → Point`.

This work demonstrates designs with various kinds of one-dimensional repetition, but Sketch-n-Sketch doesn't yet have any pleasant way to affect designs with 2D repetition, such as checkerboards. One possible solution might be to offer a version of Repeat by Indexed Merge that generates a template function over two variables, i and j, rather than over just one variable i.

The interaction for Reorder in List could be improved. First, there is as of yet no indication on the canvas of how the list elements are ordered. Simple numbering, displayed when a list widget is hovered, might help. Second, a drag-and-drop interface for reordering elements is preferable for reordering a list rather than choosing reordering possibilities from a results menu. Nevertheless, it is not immediately clear how to offer such a drag-and-drop reordering interface for elements that have already been positioned on the canvas by their x and y coordinates. Although Sketch-n-Sketch does offer the tool to Reorder in List, until the interaction is improved, list reordering remains an operation that is easier to perform via text edits.

List widgets cannot yet be dragged to move all of the grouped elements simultaneously. This interaction should be implemented.

In our example programs, we discovered that it is quite common for a point widget shown on the canvas to actually be several point widgets stacked exactly on top of each other, often because the same point is used multiple times in the program. If the programmer drags to make a selection, they may inadvertently select multiple overlapping points. This can cause transforms to fail or to produce unexpected results. How to resolve this UI trouble is a bit of a conundrum, because sometimes the programmer may actually want to select all overlapping points to, *e.g.*, equalize their positions if they were previously only incidentally aligned. At a minimum, the UI should be improved so that it is clear when multiple overlapping points have been selected, at least by visually declaring, *e.g.*, "6 points selected."

7.2.2 SVG and General Improvements

A number of possible improvements concern the current SVG setting but are likely to be applicable to other domains as well. Here, we discuss three.

First, if we want to display intermediate execution products, there might be a lot of them. Looking forward to general purpose programs, in the limit a program produces an intermediate result for each processor instruction dispatched—that is quite a few! Even in

our SVG domain with a limited number of widgets, clutter is still a problem. To date we have attempted to reduce clutter in two ways: we hide widgets until necessary to display them, *i.e.*, when the mouse hovers over a relevant element, and we reposition the bounding boxes of list and call widgets to minimize overlap. Even with these techniques, there is still sometimes quite a bit of visual noise on the canvas. Reducing that visual noise may prove to be an tradeoff with allowing functionality to be discoverable, and there may be no clear optimum.

Second, the recursive drawing feature presented in this work is still fairly preliminary and needs more development before it can work for examples besides the Koch snowflake. As well, the automatic termination condition synthesis might offer more options than just fixed depth. A synthesis algorithm that examines the execution environment, similar to that for PBE holes, might prove useful for generating reasonable termination conditions.

Finally, we have not yet given great thought to optimizing performance. On larger examples, re-running the code can be rather sluggish. To generalize this work to other domains, more attention will need to be paid to efficient computation.

7.2.3 General Improvements

Within the features of Sketch-n-Sketch likely to be broadly applicable to many output-directed programming domains, a number of improvements could be made. We discuss four.

First, the language in this work is an Elm-inspired [15] re-skinning of the more heavily parenthesized `little` language of the original Sketch-n-Sketch [12]. The Elm-like syntax improves readability considerably, but this work still lacks traditional ML features such as algebraic data types or mutual recursion. This work's optimistic assumption that any number-number pair is an (x, y) point is inelegant and could be avoided with ADTs or record types. Richer structural types are also more broadly useful to allow better representation of arbitrary data structures.

Second, the different types of provenance in this work were sufficient to build the functionality shown here, but, as discussed at the end of § 5.2, each has certain limitations that may warrant their replacement. In particular, Sketch-n-Sketch's various provenance mechanisms should be unified. The current provenance mechanisms also cannot unwind computation to regenerate a value outside of its scope if the value was based on bound variables. This ability that would facilitate better Make Equal results in certain scenarios. The functional program slicing work of Perera et al. [46] proposes tracing paired with an *unevaluation* relation that might be adaptable for this purpose.

Third, all the program transformations presented in this work were hand-coded one-by-one. In addition to being time consuming, the one-by-one hand coding has resulted in varying levels of quality between the different transforms. We would like to see a domain-specific language developed for the specification of output-directed code transformations to (a) ease the implementation burden of creating a new transform, (b) increase confidence in

transform correctness, (c) consistently generalize the handling of non-deterministic choices during transformations—*e.g.*, the shape drawing tools, Add to Output, and Dupe do not offer multiple choices, but perhaps they should and they would if defined via a DSL—and (d) enable a single generalized handling of interpreting transforms within an execution environment—a number of transforms (notably Make Equal and Relate) only operate on program constants and cannot take advantage of the unique execution environments within a function call, that is, we would like to be able to say, "Make this equal to that, but *within this function call* rather than always."

Finally, 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 no longer be sufficient: the expression may be offscreen. Scrolling the code to the relevant expression may be reasonable, as is done in, *e.g.*, Posma, Jan Paul [48]. 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.

7.2.4 Graphical Widgets for Non-Visual Domains

As we look towards taking output-directed programming beyond the graphics domain to more general-purpose programs, a major question is how to visual the execution products of non-visual domains. Textual pretty-printers for values are built into most languages, these could be a starting point. Alternately, various ways to visual programs for pedagogical or comprehension purposes have been explored in, *e.g.*, [57, 27, 43] which might provide fruitful ideas. How to usefully display and focus on the relevant portions of large data structures will likely be a challenge.

CHAPTER 8
CONCLUSION

Text-based programming is ubiquitous for its flexibility and power, but the text-based representation of a program is abstract compared to the concrete values on which computation is performed. To help the programmer understand the concrete operation of their program, live programming efforts seek to visualize the concrete execution values in order to tighten the feedback loop during the programming process. We take the live programming concept a step further and make these visualized execution products not just visible, but *manipulable*, a paradigm we call *output-directed programming*.

In this work, we presented an output-directed programming system that enables the programmer create picture-drawing programs in a general-purpose (functional) programming language with few or no text edits.

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.

We demonstrated the expressive power of these output-directed programming tools by creating programs for 16 non-trivial designs using only the output-directed transformations.

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. This work is an early step on that journey.

REFERENCES

[1] A Test Suite for Programming by Demonstration. In Richard Potter and David Maulsby, editors, *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[2] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A Core Calculus for Provenance. *Journal of Computer Security*, 2013.

[3] Shaon Barman, Sarah Chasins, Rastislav Bodík, and Sumit Gulwani. Ringer: Web Automation by Demonstration. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2016.

[4] Gilbert Louis Bernstein and Wilmot Li. Lillicon: Using Transient Widgets to Create Scale Variations of Icons. *Transactions on Graphics (TOG)*, 2015.

[5] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. *Pharo by Example*. Square Bracket Associates, 2010.

[6] Alan Borning. The Programming Language Aspects of ThingLab. *Transactions on Programming Languages and Systems (TOPLAS)*, October 1981.

[7] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory (ICDT)*, 2001.

[8] 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.

[9] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. Rousillon: Scraping Distributed Hierarchical Web Data. In *Symposium on User Interface Software and Technology (UIST)*, 2018.

[10] Salman Cheema, Sumit Gulwani, and Joseph LaViola. QuickDraw: Improving Drawing Experience for Geometric Diagrams. In *Conference on Human Factors in Computing Systems (CHI)*, 2012.

[11] 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.

[12] 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.

[13] 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.

[14] Jonathan Edwards, Jodie Chen, and Alessandro Warth. Live End-User Programming. In *LIVE Workshop*, 2016.

[15] Evan Czaplicki. Elm. `http://elm-lang.org`.

[16] 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.

[17] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based Synthesis for Complex APIs. In *Symposium on Principles of Programming Languages (POPL)*, 2017.

[18] Michael Gleicher and Andrew Witkin. Drawing with Constraints. *The Visual Computer: International Journal of Computer Graphics*, 1994.

[19] Chris Granger. Light Table—A New IDE Concept. 2012. `http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/`.

[20] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete Completion using Types and Weights. In *Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[21] Anthony C. Hearn. REDUCE: A User-Oriented Interactive System for Algebraic Simplification. In *Interactive Systems for Experimental Applied Mathematics*. Academic Press, 1968.

[22] Peter Henderson. Functional Geometry. In *Symposium on LISP and Functional Programming*, 1982.

[23] Peter Henderson. Functional Geometry. *Higher-Order and Symbolic Computation*, 15 (4):349–365, Dec 2002.

[24] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor. In *Technical Report 131a, Digital Systems Research, Digital Equipment Corporation*, 1994.

[25] 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.

[26] Jennifer Jacobs, Sumit Gogia, Radomír Mech, and Joel R. Brandt. Supporting Expressive Procedural Art Creation Through Direct Manipulation. In *Conference on Human Factors in Computing Systems (CHI)*, 2017.

[27] 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.

[28] Saketh Kasibatla and Alex Warth. Seymour: Live Programming for the Classroom. In *LIVE Workshop*, 2017.

[29] 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`.

[30] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[31] David Kurlander. *Graphical Editing by Example*. PhD thesis, Columbia University, 1993.

[32] Kevin Kwok and Guillermo Webster. Carbide Alpha. `https://alpha.trycarbide.com/`, 2016.

[33] 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.

[34] Henry Lieberman. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[35] Henry Lieberman. Tinker: A Programming By Demonstration System for Beginning Programmers. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[36] 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.

[37] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping To Navigate the API Jungle. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[38] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying Graphical Procedures by Example. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1989.

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

[40] McDirmid, Sean. A Live Programming Experience. Future Programming Workshop, StrangeLoop 2015. `https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8` `https://www.youtube.com/watch?v=YLrdhFEAiqo`, .

[41] McDirmid, Sean. The Future of Programming will be Live. Curry On 2016. `https://www.youtube.com/watch?v=bnqkglrSqrg`, .

[42] Greg Nelson. Juno, A Constraint-Based Graphics System. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1985.

[43] 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.

[44] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[45] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed Completion of Partial Expressions. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[46] 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.

[47] 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.

[48] Posma, Jan Paul. jsdare: A new approach to learning programming. Master's thesis, University of Oxford, St Hugh's College, 2012. `http://jsdares.com/`.

[49] 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.

[50] Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. Imperative Functional Programs That Explain Their Work. In *International Conference on Functional Programming (ICFP)*, 2017.

[51] Schachman, Toby. Apparatus. `http://aprt.us/`.

[52] Schachman, Toby. Recursive Drawing. Master's thesis, New York University Interactive Telecommunications Program, 2012. `http://recursivedrawing.com/`.

[53] Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*. 2016.

[54] Christopher Schuster and Cormac Flanagan. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*, 2016.

[55] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[56] James Somers. The Coming Software Apocalypse. *The Atlantic*, September 2017.

[57] Juha Sorva. *Visual Program Simulation In Introductory Programming Education*. PhD thesis, Aalto University, 2012.

[58] Jan Sparud and Colin Runciman. Complete and Partial Redex Trails of Functional Computations. In *International Workshop on Implementation of Functional Languages (IFL)*, 1997.

[59] Ivan Sutherland. *Sketchpad, A Man-Machine Graphical Communication System*. PhD thesis, MIT, 1963.

[60] Victor, Bret. Learnable Programming. `http://worrydream.com/LearnableProgramming/`.

[61] Victor, Bret. Drawing Dynamic Visualizations. `http://worrydream.com/#!/DrawingDynamicVisualizationsTalk`, 2013.

[62] 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.

[63] 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.

[64] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, 1979.

[65] Haijun Xia, Bruno Araújo, Tovi Grossman, and Daniel J. Wigdor. Object-Oriented Drawing. In *Conference on Human Factors in Computing Systems (CHI)*, 2016.

[66] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A Brief Survey of Program Slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[67] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. DemoMatch: API Discovery From Demonstrations. In *Conference on Programming Language Design and Implementation (PLDI)*, 2017.