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
DECEMBER 5, 2018
TABLE OF CONTENTS
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 pro-
gramming 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. Such live programming programming
systems have targeted both educational [37, 47, 36, 21] and expert [15, 4] 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 we perhaps 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 creating programs, while at the same time offering streamlined interac-
tions for building up large portions of the program. But because the program is still text, any
functionality that can’t be added to the program by direct manipulation can still be created
by regular text editing. Thus, 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 [43]
1.4 Sketch-n-Sketch

In this work, we extend the Sketch-n-Sketch output-directed programming system [9], an editor for creating programs that output SVG vector graphics. In the version of Sketch-n-Sketch upon which we build, once a programmer has written a program, they may directly manipulate the output to move, resize, and recolor shapes. Appropriate constants in the program are updated accordingly.

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).

We also make several UI choices to enable 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 to generate non-trivial designs.
CHAPTER 2
RELATED WORK

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 created may be parameterized based on existing 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 [35] is notable for offering a PBD workflow 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. [49, 19]). Of these systems, Apparatus [38], Recursive Drawing [39], and Geometer’s SketchPad [18] 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. Although we expect our techniques might be useful for the actual creation of parametric drawings, we primarily view the visual domain as a convenient initial substrate for discovering how one might build programs by output manipulation. Thus, we have focused our development of the tool on exploring more of the program transformation space 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 [45], explicitly view building a constrained system as a programming task [5, 17, 13]. 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. Instead of 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)

Several early PBD approaches [10] used shape drawing as a domain for exploring non-textual programming techniques. These systems usually rely on a visual representation of the program rather than a textual one (e.g. [23, 25]), or show actions step-by-step [28]. 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 [26] 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 slightly more practical bent. These systems address a wide variety of domains, such as data visualization [46], mobile applications for collaboration [11], web scraping [2, 7], and API discovery [50]. Each of these systems focus on solving a particular domain task and either do not expose the program as plain text in an ordinary language, or do not offer PBD 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”), often with some built-in inference used to determine what the arguments or triggers for the action was. In contrast to this imperative style, our system builds functional, effect-free programs. Instead of one user action generating one or two statements, each user action instead specifies a program transformation that might add expressions to the program, or it might just modify multiple existing expressions. Techniques for transforming side-effect-free expressions are more likely to generalize to non-visual domains than a domain-specific stateful computation model.

2.5 Output-Directed Programming

A number of recent systems offer a regular text-based programming experience augmented with abilities to directly manipulate output to enact code changes. The transformations available may be “small”, e.g. changing constants [9, 24, 29], strings [48, 41, 24, 29], or list literals [29], but, like our improved SKETCH-N-SKETCH, several systems enable “larger” program changes via output manipulation. We discuss two below.

Transmorphic [40] re-implements the Morphic UI framework [27] using static, functional (i.e. stateless) views. Transmorphic retains Morphic’s directly manipulatable morphs by affecting changes to the view’s (text-based) code rather than changing live object state. Adding and removing morphs, or changing a morph’s primitive properties, are supported. 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.

Like the work presented here, APX \cite{30, 31} 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 entirely with direct manipulation edits on the canvas. The “Koch Snowflake” example highlights point widgets, list widgets, and recursive function construction via focused editing; “Tree Branch” demonstrates offset widgets and repetition over point lists; and “Target” briefly walks through a programming-by-demonstration workflow for other repetition scenarios.

Below, tool names 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. 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.

3.1.1 One-Third Point Function

Tools Highlighted: Draw Point, Relate, Abstract, Rename in Output, Delete
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 almost entirely blank, defining only an empty list of SVG shapes:

\[
\text{svg (concat [ ])}
\]

We will build this first helper function by demonstration; to do so we need example points to work with. We select “Point or Offset” 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:

\[
[x, y] \text{ as point} = [87, 206]
\]

\[
\text{svg (concat [ ])}
\]

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 \((87, 206)\) as shown on the right. Even though this point isn’t part of our program’s output, whenever the Sketch-n-Sketch evaluator encounters a number-number pair during execu-
Figure 3.2: The Output Tools floating panel appears when some item on the canvas is selected. Various program transformations are offered.

tion of our program, a point widget corresponding to that coordinate is emitted as a benign side effect during execution.

We add two more points in similar fashion, and then drag the points with the “Cursor” tool (utilizing the pre-existing live synchronization [9] of Sketch-n-Sketch) 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. We would now like to tell Sketch-n-Sketch to relate the $\frac{1}{3}$ point 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, which triggers an “Output Tools” panel that offers possible actions to transform the program based on our selection [Figure 3.2]. We hover the mouse over the Relate tool. On hover, Sketch-n-Sketch attempts to discover an arithmetic expression that, were it substituted into the program, would define one of points in terms of the others and leave our points in roughly the same place. After about 20 seconds of guess-and-check work, three possible results are shown in the Relate tool submenu[1].

---

1. The freeze annotations on 1.5! and 3! indicate that these numbers should not be changed during live synchronization [9].
The Relate tool found 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 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.

\[
\begin{align*}
\text{[x3, y3] as point3} &= \begin{bmatrix} 264, 131 \end{bmatrix} \\
\text{[x, y] as point} &= \begin{bmatrix} 87, 206 \end{bmatrix} \\
\text{[x2, y2] as point2} &= \begin{bmatrix} x / 1.5! + x3 / 3!, y / 1.5! + y3 / 3! \end{bmatrix} \\
\text{svg (concat [ } & \\
\text{ ])}
\end{align*}
\]

The point2 definition has been rewritten using the discovered 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. Abstract 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 \( \begin{bmatrix} \frac{1}{3} \end{bmatrix} \) point. We choose the last result, resulting in the following code:

\[
\begin{align*}
\text{[x3, y3] as point3} &= \begin{bmatrix} 264, 131 \end{bmatrix} \\
\text{[x, y] as point} &= \begin{bmatrix} 87, 206 \end{bmatrix} \\
\text{point2Func [x3, y3] [x, y] =} & \\
& \begin{bmatrix} x / 1.5! + x3 / 3!, y / 1.5! + y3 / 3! \end{bmatrix} \\
\text{[x2, y2] as point2 = point2Func point3 point} & \\
& \ldots
\end{align*}
\]

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, indicating that the point was produced by a call to `point2Func` in our program.

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, 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 line was removed because both of the other points were also involved in calculating `point2`—DELETE removed something about all three 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:** Distance Features, Make Equal

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. However, 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, as shown to the right, 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 [16]) 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.

```plaintext
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, Snap-Drawing, Call Focusing, Focused (Recursive) Drawing, Add to Output, Reorder in List

The principal component of our design is the fractal motif, which we will then 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:

```plaintext...
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 so the existing endpoints are reused instead of adding new points—the endpoints are pulled out into variables and used for both calls.

```plaintext
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 newly 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 (shown after the next step 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.

Our system provides 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 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. As a result, SKETCH-N-SKETCH inserts an if-then-else recursive skeleton and the recursive call.

```plaintext
... 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 has 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 machKochPts between the remaining three pairs of points; the calls are inserted in the recursive branch.

```plaintext
... 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 only the blue point is being output from our
function. 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.

We want the leftmost four points to all be in the output of the base case, rather than just the protruding point. The fifth point will be provided by the neighboring call to the base case.

We click-select the three additional points we would like to be in the output and then 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.

```plaintext
... 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 we re-focus the recursive case.

2. 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!
The recursive case currently only returns \([\text{equiTriPt2}]\); 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 with a dashed gray border that we may select. We select all four returned lists from the calls to the base case and invoke \textsc{Add to Output}.

\[
\text{...}
\]

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

\textsc{Sketch-n-Sketch} realizes we are trying to combine lists together and inserts a \texttt{concat} (flatten) call so the function produces a list of points, rather than a list of lists of points. \textsc{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. The \([\text{equiTriPt}]\) singleton list from the original return value of the function is extraneous; we find and \textsc{Delete} its list widget, leaving a return expression of:

\[
\text{concat } [\text{makeKochPts2}, \text{ makeKochPts3}, \text{ makeKochPts4}, \text{ makeKochPts5}]
\]

All points are now in the output of \texttt{makeKochPts} (and therefore displayed in blue). One last item remains: we must choose a termination condition. The focused call widget for \texttt{makeKochPts} displays the conditional for the recursive case as \texttt{not <| ??terminationCondition} which we can click to \textbf{Choose Termination Condition}. Currently, \textsc{Sketch-n-Sketch} offers only one kind of automatically generated termination, fixed depth, which we choose. A depth argument is added to our \texttt{makeKochPts} function which is decremented on the recursive calls. These changes are visible in the final code \textbf{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 \textsc{Sketch-n-Sketch} draws a slider for modifying depth in the output \[9\].

\[
\text{equiTriPt2} = \text{makeKochPts } 2\{1-5\} \text{ point point2}
\]
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* all our lists together. We choose this latter option. 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!

3.2 Example 2: Tree Branch

*Tools Highlighted: Draw Offset, Repeat Over List*

To highlight offset widgets and the repetition tools, we construct the tree branch design
shown in Figure 3.3. 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 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. $x_{	ext{offset}} = x + 102$. 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:

\[
\text{...}
\begin{align*}
\text{num} & = 102 \\
\text{xOffset} & = x + \text{num} \\
\text{xOffset2} & = x - \text{num}
\end{align*}
\]

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. Several functions in the standard toolbox that returns a list of points. Among these, pointsBetweenSepBy 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.

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

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 in their spacial positions, but not over any other attributes such as size or color. To support other repetitions scenarios where the varying attributes could be calculated from an index (i.e. 0,1,2,...), we offer a programming-by-demonstration workflow which we now briefly illustrate through the construction of a target.

To start, 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).

```plaintext
... circles =
  map (\i ->
    circle \n      ??(1 => 0, 2 => 466, 3 => 0) \n    point \n      ??(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 a syntactic merger of our original three circle definitions. The differences between the three original circle expressions—radius and color—have been turned in 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 [42] 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 below.

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*sep` expression, to calculate the radii of our circles. Similar to the Koch snowflake example, the inserted code `zeroTo 3{0-15}` contains a slider annotation allowing us to manipulate the number of circles in the output. We display five circles for the final design (Figure 3.4).
Figure 3.4: 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 available in 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. To avoid being constrained by the syntax of the existing program, Sketch-n-Sketch attempts to add the new shape variable to some list literal in the program and succeeds when the size of the output increases by the expected amount. 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 Prelude 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

There are three ways to focus an expression or definition. Call widgets on the canvas may be clicked to focus. Or, if a selected item can be interpreted as a coming from an expression, a Focus Expression tool is displayed in the output tools. Finally, an enterprising programmer can text-edit the comments into their code themselves.

If a function or group is focused, the drawing actions attempt add to that function or group rather than the main shape list. As shown in the first overview 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, which is 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 select items are always numerically equal. The numeric traces of the selected numbers 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), 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. Like the MAKE EQUAL tool, RELATE works 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.

4.3.2 List Widgets

Enables the selection, reordering, and renaming of lists, thereby also enabling the selection, reordering, 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.

4.3.3 Abstract

Abstracts some selected expression over its free variables, after gathering in all bindings that are only used in that expression—bindings with no free variables are not gathered, to ensure there are free variables left to abstract over.

4.3.4 Merge

Performs a syntactic merge of the selected expressions. Differences between the expressions become arguments to the function created.

4.3.5 Repeat over Function Call

Builds an abstraction \( f \) over some point in the selected shape to repeat that shape along several points, inserting \( \text{map } f (\text{pointFunc} \ldots) \) into the program where \( \text{pointFunc} \) is a Prelude or program function that returns a list of points.
4.3.6 Repeat over Existing List

Like Repeat over Function Call, but repeats along an existing point list in the program instead over a new function call.

4.3.7 Repeat by Indexed Merge

Merges the selected expressions, with differences replaced by PBE holes. A skeleton map (\i \rightarrow merged) (zeroTo n\{0-3\ast n\}) is inserted into the program, with an appropriate literal for n. An extra result is also provided to optionally reverse the indices.

4.3.8 Fill PBE hole

Offers replacements for any PBE holes found in the program. Described below in 5.3.

4.4 REFACTOR

4.4.1 Rename in Output

Widgets throughout the UI are labeled with associated program expressions. 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 display and may be clicked 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 with buttons next to each to remove or reorder the argument. Selecting a feature while in a focused call displays an option in the Output Tools Panel to add the feature as an argument to the function.

4.4.3 Reorder List

Finds list literals the selected value participates in and offers four options to move the relevant item within each list: to the list head, to the list end, one space headwards, or one space endwards. May be used to affect the “Move to front” etc. 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

Attempt 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 determine where the new variable usage should be placed.
4.4.5 Select Termination Condition

Fill the conditional hole produced by drawing a function inside itself, as demonstrated in the [Koch snowflake example](#). Only one option is currently supported: fixed depth termination.
CHAPTER 5
IMPLEMENTATION

5.1 Introduction

Sketch-n-Sketch is an in-browser app written in the functional language Elm [12] with exceptions and mutation added via a custom library.

Below, we discuss several of the key technical mechanisms that facilitate the operation of the 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.

In a functional setting, such as the Elm-like language exposed to the programmer in Sketch-n-Sketch, values are immutable. “Changing” a value actually means creating a new, slightly different value. For output-directed programming in such a setting, our key insight is that provenance can be loose. Although we generally find that expressions later in the computation are more relevant, we tag each value such that it could be traced all the way back to the primitive constant numbers that effected it. The programmer is always editing within some context (e.g. the program, not the standard library) and is therefore always looking for some expression(s) within the context of their attention. A loose approach to provenance ensures that if there is some expression in their attention that can be associated with the selected output value, it will be found.

Previous versions of Sketch-n-Sketch tagged numeric values only with control-flow ignoring traces of the mathematical operations performed on them, enabling live synchronization [9]. These numeric traces are still for Make Equal, Relate, and when a drawing snap cannot be resolved by using or introducing a variable.

A goal in this work was to extend provenance so all kinds of values, not just numbers, could be traced back to relevant program expressions. Those expressions might be anywhere in the program (not just at the top level), and this provenance should operate without requiring the programmer to call special functions.

We deploy two complementary techniques: (a) “Based On” provenance to trace the origin of values and (b) “Parents” provenance to find larger values that contain the value of interest.
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, SKETCH-N-SKETCH first needs to trace back the value the programmer has selected to one or more program expressions, and then affect the transformation.

In the current implementation, the context within which we interpret values is defined to be the user-visible program itself, and not the provided Prelude of built-in code which is implicitly imported into every program.

Below we discuss the extra tags we add to values during evaluation to record provenance, and the algorithm for interpreting a value’s provenance as 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 the standard big step relation \( \Gamma \vdash e \Downarrow v \) we employ \( \Gamma \vdash e \Downarrow v^{e,\{v_1,\ldots,v_n\}} \), where \( \{v_1,\ldots,v_n\} \) is the set of tagged values upon which \( v \) is based. We call this “Based On” provenance.

In “Based On” provenance, control flow is implicit. 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. We are most concerned with values “becoming” other related values, the how is implicit in where these values appear in the code. But if needed, looking at the expression tagged to the return value will reveal an expression inside the appropriate function and thereby the function itself. As well, the application is not based on any of the arguments to the function call—transitively following the “Based On” provenance of the return value will eventually discover any arguments used.

A selection of these evaluation rules are displayed in §5.2.1.

5.2.2 Interpreting “Based On” provenance

“Based On” provenance answers the question, “For a particular value, what other values at other execution steps were used to produce it?” As we now detail, we use the answer to that question to answer our main query: “For a particular selected value, what expression(s) in the program does it most likely refer to?”

Notice that transitively following the “Based On” provenance of a value yields a tree of tagged values. The values near the root of the tree are newer, while the values at the leaves of the tree were produced earliest in the execution. To this tree we apply the following algorithm to interpret the value as a set of program expressions:

1. Philosophically, the context should be narrowed even further if the programmer has focused a particular definition. Our examples have not required this yet.
\[
\begin{align*}
\Gamma \vdash e \downarrow n^e, \Gamma^e,
\end{align*}
\]

\[
\begin{align*}
e &= x \quad x : v_1^{e_1, \{v_2, \ldots, v_n\}} \in \Gamma \\
\Gamma \vdash e \downarrow v^e, \{v_1^{e_1, \{v_2, \ldots, v_n\}}\}
\end{align*}
\]

\[
\begin{align*}
e &= \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
e_1 \downarrow \text{true}^{e_1, \{\ldots\}} & \quad e_2 \downarrow v_2^{e_2, \{\ldots\}} \\
\Gamma \vdash e \downarrow v^e, \{v_2^{e_2, \{\ldots\}}\}
\end{align*}
\]

\[
\begin{align*}
e &= \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
e_1 \downarrow \text{false}^{e_1, \{\ldots\}} & \quad e_3 \downarrow v_3^{e_3, \{\ldots\}} \\
\Gamma \vdash e \downarrow v^e, \{v_3^{e_3, \{\ldots\}}\}
\end{align*}
\]

\[
\begin{align*}
e &= e_1 + e_2 \\
e_1 \downarrow n_1^{e_1, \{\ldots\}} & \quad e_2 \downarrow n_2^{e_2, \{\ldots\}} \\
v &= n_1 + n_2 \\
\Gamma \vdash e \downarrow v^e, \{n_1^{e_1, \{\ldots\}}, n_2^{e_2, \{\ldots\}}\}
\end{align*}
\]

\[
\begin{align*}
e &= e_1 e_2 \\
e_1 \downarrow (\lambda x. e_f, \Gamma')^{e_1, \{\ldots\}} & \quad e_2 \downarrow v_2^{e_2, \{\ldots\}} \\
x : v_2^{e_2, \{\ldots\}}, \Gamma' \vdash e_f \downarrow v_3^{e_3, \{\ldots\}} \\
\Gamma \vdash e \downarrow v^e, \{v_3^{e_3, \{\ldots\}}\}
\end{align*}
\]

Figure 5.1: Selected 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. Where trivial, \(\Gamma \vdash\) is omitted for brevity.

1. We discard all subtrees whose values were all produced by expressions outside the context of interest (i.e. we discard all subtrees occurring entirely in the Prelude).

2. We consider all possible prunings of the remaining tree (i.e. all possible ways to discard some of the subtrees).

3. We keep all prunings whose leaves were all produced by expressions inside the context of interest (i.e. we discard a pruning if any of its leaves were produced by a Prelude expression).

4. For each remaining pruning, we take the values in the pruning’s leaves, and make a set out of the expressions tagged to those values. We call this expression set an interpretation. Since there may be multiple possible prunings, we may end with a number of different possible interpretations.
For example, the value produced by evaluating the expression $10 + 20$ has two possible expression interpretations: the set of the single expression \{10 + 20\}, or the set of two expressions \{10, 20\}.

In general, we find the expressions furthest along in the execution to be the most relevant, i.e. the interpretation from the most aggressive pruning is the best interpretation. And in practice, step (2) above may produce a combinatorial blow up. To avoid this blow-up, in most cases our actual implementation employs a specialized version of the algorithm that finds this best interpretation only. Some tools may impose additional constraints while searching for interpretations. For example, the DUPLICATE tool searches for an interpretation that consists only of a single expression, and that expression must not be a variable usage.

“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

If a value (e.g. the number 10) is packed inside a list (e.g. \([10, 20]\)), the “Based On” provenance of the value 10 will not give any hint that it was ever included in a list—a value is only tagged with “Based On” provenance when it is produced by an evaluation step. The production of the list \([10, 20]\) produces only a list that points to the previous number 10, rather than reproducing it, so 10’s “Based On” provenance is unchanged when the list \([10, 20]\) is produced. Thus, if we have the number 10 and want to use it somewhere in the program, there is no clear why to know that we might be able to unpack \([10, 20]\) to get our desired number: 10’s “Based On” provenance does not point at \([10, 20]\). We need another kind of provenance that points the other way.

Thus, in order to be able to find values that contain a value of interest, we tag all contained values with what we call “Parents” provenance. “Parents” provenance operates as follows: if a step of evaluation results in a a value that contains other values (i.e. a list—in this version of SKETCH-N-SKETCH, lists are the only kind of container), 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.

In SKETCH-N-SKETCH’s SVG setting, the most important use of “Parents” provenance is to, given two numbers, recover any points where those two numbers were used as the $x$ and $y$ coordinates. The SKETCH-N-SKETCH UI does not, technically, allow selection of points—only coordinates—as “point” is not a primitive concept in SVG (e.g. an SVG rectangle is parameterized on $x,y,width,height$, not on $topLeft,width,height$). In the prior discussions in this work when we mention selecting "points", in practice we were selecting precisely the $x$ and $y$ coordinates of the point, from which the point pair itself might later be recovered.

“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 snap resolution may make the needed value available by inserting a binding that pattern matches the needed value out of the container variable.

5.2.4 Approaches to Provenance in Other Work

The live programming environment Posma, Jan Paul [36] similarly keeps track of execution steps and operates such that any mouse point on the canvas can be mapped to an expression to highlight; however, no output-directed manipulations are supported.

Output to location in code Burckhardt et al. [6], Posma, Jan Paul [36]

5.3 Value Holes

To affect snap-drawing and grouping, Sketch-n-Sketch relies on value holes, a mechanism to simplify inserting new template code that must honor output-directed constraints. A value hole is a temporary, internal expression that contains a value. Values inside holes are placed in program where an expression should equal an existing output value, such as at the $x$ and $y$ coordinates of a line endpoint being snapped to an existing point. Before showing the code to the programmer, the provenance of the value in each hole is inspected and the hole is replaced by an expression that enforces the snap.

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 the 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 drag-drawing a 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 offer the 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 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.

In this work, PBE holes are only generated by the Repeat by Indexed Merge tool. However, we imagine PBE holes might also be useful to facilitate a future 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 PBE programmer synthesizer such as Myth [33].

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

When the programmer defines a custom function, we want to the function to be drawable with the mouse without requiring the programmer to annotate which of the function arguments are e.g. the width and height. During type inference, we tag the types at code locations with additional side information explaining the expression’s role. The standard library drawing functions are already annotated with roles—using these functions causes role information to flow back into program. A similar scheme called brands is used in APX [30]. Perhaps unlike brands, we also apply certain usage rules—e.g. a number added to an X value must be a HorizontalDistance—to infer roles in more cases.

These rules allow programmer-defined functions to be drawable without inserting special annotations—the horizontal/vertical distance roles for width/height bubble up to the custom
function’s arguments from the Prelude drawing functions that are called.

To draw programmer-defined functions on the canvas, SKETCH-N-SKETCH needs to figure out which arguments are points, widths, or heights, and also needs to provide reasonable defaults for the remaining arguments. Points are number-number pairs and 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. Note that Prelude functions are already annotated with roles (via type aliases), so the use of such functions propagates role information into the program. 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, 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 tries to add centimeters to inches, but roles are side information that is not used for type checking.
CHAPTER 6
EVALUATION

To evaluate our approach, we have implemented a total of sixteen programs without text-based editing. The output of each are shown in Figure 6.1, along with their program sizes (measured in source lines of code and math operations). These 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 [3, 8, 9].

3. We additionally implement 5 novel tasks.

Below, we discuss the features required to complete the remaining WWID: PBD parametric drawing benchmarks.

We base our evaluation primarily on the benchmark test suite proposed in What What I Do: Programming by Demonstration [1].

Of the 32 benchmark tasks spanning visual, UI automation, and textual domains, 15 can be reasonably interpreted as the construction of a parametric drawing. The version of Sketch-n-Sketch on which we build was able to complete one task and partially complete two others. In this work, we are able to complete four of the examples and partially complete an additional two. One of those complete examples was the Koch snowflake presented in the overview. Below, we briefly present the other five examples as well as examples drawn from the prior version of Sketch-n-Sketch, examples adapted from related work, and a few new examples. Including the overview examples, overall we present a total of 16 examples in this work. To close this section, we discuss what features might be required to address the remaining nine parametric drawings in the WWID PBD benchmark suite.

6.1 Precision Floor Plan (WWID: PBD p. 566 Alan Turransky)

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 'table' rectangle, equalize the heights, and then invoke RELATE on the widths. (xxx LOC including xxx math operations)

6.2 Mondrian Arch (WWID: PBD p. 554 Henry Lieberman)

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 [25] programming
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). \( \lambda = \) Final design is abstracted. \underline{} = Task from WWID: PDB test suite (dashed = only partially completed). \( \star \) = need to remove the stars (\( \star \star \) = need to remove the stars). Other sources: Battery is from Bernstein and Li [3], Ladder from Cheema et al. [8], Logo (via Three Tris), N Boxes, and Ferris Wheel from Chugh et al. [9].

by demonstration workflow.

In this work, to end with the ideal parameterization of \([\text{left}, \text{top}] \text{ width height stoneWidth}\), 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 serve as the overall height—however we consider overlapping shapes to be inelegant in this case as it does not match the spatial reality of e.g. Stonehenge and thus forego this method. The remainder of the design is constructed by repeatedly utilizing \texttt{MAKE EQUAL} once all three rectangles have been drawn. We take advantage of the multiple resolution options provided by each \texttt{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 lintel 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 it with the offset amount we began the design with. MAKE EQUAL offers several options to enforce the constraint, we choose the option that preserves \texttt{stoneWidth} as a parameter and computes \texttt{pillarHeight} as \texttt{height - stoneWidth}. The offset may then be removed as the \texttt{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 \texttt{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. \texttt{stoneWidth} is before \texttt{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 xxx LOC and xxx math operations.

6.3 Balance Scale (WWID: PBD p. 568 David Kurlander)

In this example, the goal is constrain the drawing so the hanging trays always stay attached to the arms as they rotate. We want each arm to rotate independently but stay equal in length to each other. This is accomplished by drawing 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 nice parameterization for the parts 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 xxx LOC with xxx math operations.

6.4 Box Volume (WWID: PBD p. 546 Finzer and Jackiw)

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 of 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. \( \text{width}_{\text{total}} - 2 \cdot \text{width}_{\text{cut}} \)) 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 does not snap during drawing (only the amount), but the end point can be snapped after the fact with \text{MAKE EQUAL}, thus causing the amount to be derived as \( \text{width}_{\text{total}} - 2 \cdot \text{width}_{\text{cut}} \).

Once in place, we can construct the perspective view. The back plate of the box can be laid out with offsets, snapping their amount to the existing \( \text{width}_{\text{cut}} \) and just-created offset amount of \( \text{width}_{\text{total}} - 2 \cdot \text{width}_{\text{cut}} \). To construct the diagonal, we utilize the \text{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 an instance of the \text{onLine} function from the back corner of the box which places a point on a diagonal. To make that point’s distance from the corner equal to the box base length, we use equalize the distance features such that the ratio is calculated from the appropriate math to stay at a constrained distance from the corner.

The rest of the design skeleton can be constructed with offsets, with a single polygon four-sided polygon attached to each face. Thus drawn, resizing any of the \( \text{width}_{\text{cut}} \) 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 xxx LOC with xxx math operations.

To help the student know when the volume is maximized, a complete design should also display 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. Each arm is based on a separate call to the \text{nPointsSepBy} library function which takes a base point, a horizontal separation, a vertical separation, and a number of points. 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. Since the calculations in the standard library are not visible in the code box, any widgets produced by these calculations might confuse the programmer and clutter the canvas. In the "View" menu, the "Show Offset Widgets from Prelude" displays these offsets on the canvas, 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 \( \text{width}^{\square} \) or \(-\text{width}^{\square}\). For the those that should be \( \text{width}^{\square} \), we can just invoke \text{MAKE EQUAL}. For the \(-\text{width}^{\square}\) separations, we must lay out the offset at roughly the right distance, invoke \text{RELATE}, and choose the \(0 - \text{width}^{\square}\) option (our language currently lacks a negation operator). Unlike \text{MAKE EQUAL}, \text{RELATE} must be used separately on each separation offset with a \( \text{width}^{\square} \) select, as its semantics are to "relate one selected item in terms of all others". With the repetitions laid out, \text{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 \text{concat}ed with the arm rectangle).

From here we would like to invoke the Abstract tool, however the Group and Abstract tools deliberately do not gather helper functions into the new definitions they create, which in this design leaves several instances of \( \text{width}^{\square} \) 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. (xxx LOC with xxx math operations).

### 6.6 Tackling the Remaining WWID: PBD Tasks

Two of the WWID: PBD tasks we attempt can only be partially completed in this work (Figure 6.1g and Figure 6.1h). To complete these tasks, the “Box Volume” example would require an interaction to compute and display the numeric volume of the folded box. The “Xs” task would require more precise control over what definitions are pulled into the abstraction. (Not all uses of a \( \text{squareWidth} \) parameter are pulled in to the abstraction so the design breaks if we try to add an X with a different square size.)

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. \text{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 \text{SKETCH-N-SKETCH} to create \text{if-then-else} branches outside of a recursive or hole-filling setting.
CHAPTER 7
DISCUSSION

We presented new techniques for the output-directed 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 to make Sketch-n-Sketch 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 shown in ??, the core technical implementation in Sketch-n-Sketch here—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 displayed. Intermediate execution products, and some indication of how they were computed, must be displayed. The display of intermediates is also within vision of Live programming: Kasibatla and Warth [21] is a good recent example of displaying intermediate code products line-by-line by the code. Our proposal in this work is to make these intermediate results manipulable, which we affect for
the SVG domain 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 space, and 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! But 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 are critical to human program comprehension. Thus, the ability in this work to Rename in Output is boon for helping the programmer create 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 specification of recursive functions, by drawing the function in itself (as in [39]) and by shifting editing focus between the base or recursive case, depending on 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 interaction, we also speculate that focusing the editing context might offer cognitive benefits to the programmer by decluttering the canvas. However, the examples we have explored so far are still too small to gain any intuition as to whether this hypothesis is true.
7.1.4 Programming by Demonstration or by Direct Manipulation?

The programming by demonstration 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 programming by demonstration paradigm. However, other than these two tools, 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. However, this distinction between direct manipulation and PBD in Sketch-n-Sketch may not actually be so clear. PBD’s inference step is characteristically ambiguous. Often a PBD system must ask the programmer for clarification. So too must almost all of sns’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 in a thoroughness that cannot be economically represented on the canvas. For example, imagine 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 (a) show the output of only one branch at a time, or (b) 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.

Quite a few graphics-specific features could be added to **Sketch-n-Sketch** to support more designs. 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 Test Suite. During the course of this work, we added a path tool to **Sketch-n-Sketch** but later disabled it. 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 and partially evaluate those events into templated which returns 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 are only available during drawing, not during mouse manipulations after a shape is drawn. Similarly, snaps are only allowed to point widgets or offset amounts. Snaps are a more convenient interface than manually selecting features and invoking **Make Equal**, so 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**. A more flexible implementation might instead look in the Prelude and in the program for e.g. functions of type `Rect → Point`, apply all such functions to the selected rectangle, 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 isn’t conveyed by the existence of a function `rectCLPoint : Rect → Point`. 41
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 one.

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 options from a list. However, it is not immediately clear how to offer such an interface for elements on a canvas that already have a spacial position.

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

In our example programs we discovered that it is quite common for a point widget 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. Imagining a general purpose program. In the limit, if a processor is running at 3GHz with a 4-wide dispatch, it could produce \(12,000,000,000\) intermediate numbers per second. These cannot all be displayed. 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: (a) we display widgets in a context sensitive manner, i.e. on hover of a relevant element, and hide them otherwise, and (b) 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. The automatic termination condition synthesis as well could benefit from offering 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 below.

First, the language in this work is an Elm-inspired [12] re-skinning of the more heavily parenthesized little language of the original Sketch-n-Sketch. [9] 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 an egregious abuse that 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 have certain limitations that may warrant their replacement. The “Based On” provenance is useful for answering the question, “Given a context, what expressions in this context can together be considered to ‘constitute’ the value of interest?” However, the “Based On” provenance does not store enough information to precisely recreate how a value was computed. This missing information is useful if, e.g., the programmer invokes Make Equal on an intermediate value computed inside a particular function, but the programmer is asking to use the value outside the function, outside of bound expressions the value may depend on. Smarter provenance is required to recreate the bound value outside of its scope. The functional program slicing work of Perera et al. [34] proposes tracing paired with an unevaluation relation that might be adaptable for this purpose.

Additionally, the “Parents” provenance answers “what container(s) have carried this value?” but some transforms (e.g. Dupe and Delete) are using the “Parents” provenance to answer “what container(s) carried this value to the canvas?” For the examples we have explored so far, the answer to both questions is usually the same. But the answers need not be the same, which could result in e.g. extraneous deletions. A different mechanism of identifying containment needs to be devised, perhaps by explicitly recording the unpacking which occurs in pattern matches.

Lastly, in this work Sketch-n-Sketch employs no less than four kinds of provenance. In addition to the “Based On” provenance and “Parents” provenance discussed above, the original numeric traces of Chugh et al. [2] are still in use for live synchronization, Make
EQUAL, and RELATE. During evaluation, we also additionally notice numbers used in points (i.e. in number-number pairs) and tag those numbers with their sister coordinate to support offsets. These four provenance methods should be unified.

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 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 [36]. However, in the case that a value is interpreted into multiple expressions that are not all within one page of each other, some reasonable 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 build into most language, those could prove a starting point. Alternately, various ways to visual programs for pedagogical or comprehension purposes have been explored in e.g. [44, 20, 32] which might provide fruitful ideas. How to usefully display and focus on the relevant portions of large data structures will likely be a challenge.
REFERENCES


