Output-Directed Programming
Programming with Direct Manipulation of Output

Brian Hempel
Ph.D. Thesis Proposal
May 8, 2020

Maniposynth!
data BTree a
= Leaf a
| Node (BTree a) (BTree a)
fold ?«a» ?«b» ?«c» =

In:

tree
Node
Leaf
1
Node
Leaf
2
Out:

[]
c = tree
(::)
Leaf
3
[1,2,3]
ba
fold
Node
Leaf
1
Node
Leaf
2
Leaf
3

1 Introduction

Direct manipulation is everywhere. Direct manipulation is the workflow characterized by “visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex command language syntax by direct manipulation of the object of interest” [69]. Because of these intuitive affordances, direct manipulation has become the expected way that most people interact with computers.

While the intuitive point-click-operate workflow of direct manipulation is the standard mode of interaction for most computer applications, for over half a century one important application has remained a primarily text-based activity: that application is programming. Can the intuitive workflow of direct manipulation be applied to the practice of programming? If so, direct manipulation might allow novices to learn programming more easily. And even for experts, program text can be opaque and clumsy—direct manipulation might also let them more naturally complete their tasks.

Because direct manipulation promises benefits to both novices and experts, a number of systems have explored different ways to add direct manipulation to programming. Most of these systems provide direct interactions on the program’s abstract syntax tree (AST) but not on the products of the program—its output. While a few systems do allow direct manipulation of program output to change the program’s code, these manipulations are limited to specific scenarios.

Overall, it is unknown how expressive direct manipulation of output can be for creating programs. Therefore, I propose to vastly expand the answer to the question:

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

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

Figure 1: The systems comprising the proposed thesis.
Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-directed programming interactions.

I will justify the above thesis by building and demonstrating two different output-directed programming systems, along with one supporting technical mechanism in between. The first system—called SKETCH-N-SKETCH—is a development environment for creating programs that output SVG vector graphics (completed work [27, 30]; Section 4). Looking forward to manipulation of non-visual data structures, I introduce a mechanism—called TINY STRUCTURE EDITORS (TSE)—for manipulating program values by interacting with their toString representation (completed work [28]; Section 5). The final system—called MANIPSYNTH—is an environment for output-directed programming of ordinary functional data structures (remaining work; Section 6).

Before introducing these systems, I will survey related programming techniques (Section 2) and argue that output-directed programming avoids common pitfalls of prior systems (Section 3). At the end, I offer possible avenues for future work (Section 7) and conclude (Section 8).

2 Related Work

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

2.1 Live Programming

Programmers often want to see values their program produces; either values in the program’s output, or, perhaps via a debugger, values at some intermediate state of the program. To see output values after an edit, a programmer must manually trigger and wait for their program to compile and run. Or, to see intermediate values, they must set up a debug session. These repeated operations have the potential to become tedious. The goal of live programming is to tighten the programming feedback loop by automatically updating displayed values in response to program changes. As envisioned by Tanimoto [76], the greatest level of liveness is achieved when the visual display of the program state reacts continuously to all program edits as well as to any incoming streaming data.

A large number of live programming systems have been created. Python Tutor [25] is popular teaching tool for visualizing Python program state and includes a live programming mode. Victor’s Inventing on Principle presentation [80] demonstrates several live programming environments and served as inspiration for later work [39, 44]. The LightTable IDE [23] obtained multiple hundred thousand dollars of crowd funding [9]; its key selling points were immediately available documentation and a live updating display of values flowing through code. All these systems live-update the code display in response to program changes. New live programming ideas continue to be explored at the yearly LIVE Programming Workshop [1].

Is live programming effective? Fabry [18] found the existing studies of live program lacked quantitative evidence for its effectiveness (a conclusion also reached by Rein et al. [63]). Despite the lack of quantitative evidence, participants often reported qualitative preferences for live programming [18]. Some qualitative insights about live programming system design have emerged. Via pilot studies, Kang and Guo [38] observed that two seconds of delay was a good tradeoff between maintaining liveness and reducing the distraction of UI updates. In contrast, Lerner [44] found that immediately live information was not necessarily distracting, although user customization to filter the displayed information is necessary to achieve the best effect.

In the proposal below, I assume live programming is desirable. Building on a foundation of live programming, output-directed programming seeks to make live programming more live by offering direct manipulation of the execution products.

2.2 Direct Manipulation Programming

Visual dataflow programming. In his 1966 Ph.D. thesis [75], William Sutherland introduced a direct manipulation computer system to perform what is now known as visual dataflow programming. In visual dataflow programming, the user graphically creates a computation by laying out nodes and wires on a canvas, much like a flowchart. Nodes represent operations on data, and wires carry data between operations.

Although the wires can quickly become noisy and resemble literal “spaghetti code” [85], visual dataflow programming has been practically applied to domain specific tasks. A notable example is the commercially successful...
LabVIEW [56] environment which primarily targets engineers and technicians working with electronics.

While most nodes-and-wires dataflow programming systems use nodes to represent operations and wires to represent data, the reverse is also possible. PANE [32] is a recent such example: nodes display example values and wires represent transformations between values. Because the example values are foremost in the display and may be clicked to invoke operations on them, PANE’s workflow bears resemblance to the MANIPOSYNTH system proposed below. PANE does not, however, maintain an editable text representation of the program.

The visual data processing system Luna [49] displays both the operation and output on nodes. Additionally, like the systems proposed below, Luna is a bi-modal environment offering an always-editable text representation of the program. Luna does not offer direct manipulation of the displayed output values, however.

**Blocks and other structure editors.** In block-based programming environments [3], traditional syntactic constructs such as statements, loops, and if-then-else structures are represented as graphical *blocks* that can be directly manipulated. Syntactically valid combinations of blocks snap together like puzzle pieces, allowing the programmer to build up a program with minimal keyboard input. Additionally, a toolbox of available blocks is provided so that new functionally can be discovered and immediately added to the program. Because blocks obviate the need to memorize syntax and can be used without being adept at keyboard input, block-based environments have been used in computer science education, most notably in Scratch [64] and Alice [11].

Block-based editors are a specific instance of *structure editors*, also known as *projectional editors*. In structure editors, instead of editing the program via a raw text buffer, the system offers tree transformations in order to maintain a syntactically valid program throughout its construction. Structure editors do not necessarily operate by direct manipulation. For example, the Cornell Program Synthesizer [72] operated via key commands rather than by a mouse. And, although inspired by the traditional textual rendering of the program, structure editors also differ in how closely they support ordinary raw-text editing. Several structure editors combine both direct manipulation interactions together with a more familiar raw-text editing experience—a combination the proposal below also seeks. Barista [40] and Greenfoot [6] offer drag-and-drop structural interactions while also mimicking an ordinary text buffer for ordinary editing. Similarly, Deuce [29] augments an ordinary text buffer with a structural multi-selection mode to quickly invoke refactorings. Unlike the proposal below, in structure editors the programmer directly manipulates the code rather than the output.

**Constraint-oriented programming (COP).** Following in the footsteps of Sketchpad [74], constraint-oriented programming (COP) systems explicitly view building a constrained system as a programming task [5, 31, 19]. In these systems, the programmer declares a series of constraints, either graphically (via direct manipulation) or in text. In the graphical setting, the constraints are common geometric assertions, e.g., “these points should be equidistant from this other point”; while in a non-graphical a constraint might be “$x$ should always be twice the value of $y$”. Unlike traditional programming, COP systems run a constraint solver alongside the program, querying the solver during runtime to affect the execution of the program. The systems in this proposal instead follow a standard execution model—any “constraints” are expressed as ordinary math in the program (e.g., $x2 = x1 + w/2$) and are executed normally.

Several systems targeted at visual design also follow a COP model but do not seek to expose ordinary text-based code (e.g., [36, 37]). Of these systems, Apparatus [66], Recursive Drawing [65], and Geometer’s SketchPad [36] are notable for supporting recursion.

**Programming by demonstration (PBD).** To offer end-users some of the benefits of programming, a class of interactions dubbed *programming by demonstration (PBD)* [12] allow users to, instead of typing out code, specify programs by demonstrating the desired actions to the computer. Taking the role of a learner, the computer infers the intent of the demonstrated actions and constructs a program.

Several early PBD approaches used shape drawing as a domain for exploring these non-textual programming techniques. PBD systems usually rely on a visual representation of the program rather than a textual one (e.g., [46]), or show actions step-by-step [51].

Although not as visual as peer PBD systems, Tinker [47] is notable for supporting recursion by demonstration—indeed, any Lisp expression may be created. Unlike the systems proposed here, manipulations are performed on a symbolic representation of the example not far removed from the underlying Lisp. But, like the systems proposed here, the underlying code in Tinker is set in a traditional programming language and that code is featured in the UI.

More recently, PBD techniques have been developed with a more practical bent. These systems address a wide variety of domains, such as data visualization [82], mobile applications for collaboration [15], web scraping [7], and API discovery [57]. Each of these systems focus on solving a particular domain task and, unlike the systems presented here, either do not expose the program as plain text in ordinary code, or do not offer demonstration-based editing after the initial program is generated.

3
2.3 Prior ODP Systems

Output-directed programming (ODP) is defined above as the activity of specifying always text editable code via direct manipulation of program execution products. The systems demonstrated below in this proposal are not the first ODP systems, although the goal in this proposal is to push ODP further than prior work. Most prior ODP systems only support “small” changes to the code via direct manipulation. For example, output manipulation may change numbers \([10, 42, 52, 21]\), strings \([84, 68, 42, 52]\), or list literals \([52]\) in the code. That is, direct manipulation can make minor tweaks to an existing program but does not help create the program to begin with. A handful of prior systems, however, like those systems proposed here, do enable “larger” program changes via output manipulation and can assist the programmer in creating and refactoring the program, not just modifying constant literals. These systems will be discussed later in context of the proposed systems.

3 Why Output-Directed Programming?

As discussed above, a long line of work has attempted to make programming a more direct and immediate experience than simply typing out opaque code. Programming by demonstration (PBD), in particular, has put forward many ideas over many years for how to make programming more demonstrational, and therefore more approachable, than the arcana of text-based programming \([12, 45]\). Despite all this work, PBD interfaces are rare in practice. Why? Can output-directed programming avoid the same fate?

Reflecting on her many years working on PBD systems, Tessa Lau offers five guidelines that PBD systems often neglect \([43]\). Below, I recount Lau’s principles and argue how they are satisfied by output-directed programming (ODP).

1. “Detect failure and fail gracefully.” In general, PBD systems accept a series of demonstrations from the user and attempt to infer the intended program. Because this inference may involve a complex constraint satisfaction problem, when a wrong program is produced it can be unclear why—in the limit, the constraints may simply be unsatisfiable and the system gives no hint as to why. In contrast, ODP relies less on inference: each output-directed interaction enacts a limited change to the code. Instead of constructing a large constraint satisfaction problem over several demonstrations, ODP steps are small and immediate. The effect of each step is visible and reversible. Where there is ambiguity about the meaning of an interaction, the programmer chooses the desired result before continuing.

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

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

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

5. “Consider the bottom-line value of automation.” Lau admonishes the system designer to weigh not just the user effort required to perform a single demonstration, but also the additional costs of learning to use the system and the switching cost of leaving one’s normal workflow to enter the demonstrational interface. On this point ODP has less to offer. Certain ODP interactions might go unused, despite being notionally superior to their textual counterparts. For example, in SKETCH-N-SKETCH below it is easier to draw a shape on the canvas rather than remember how to type the appropriate function call in the code. Even so, a programmer already typing text may choose to add the function call via typing. A similar phenomena has been shown in traditional integrated development environments (IDEs). IDEs have long had a large suite of labor-saving automated refactoring tools, and yet an open problem in the software engineering community is how to get folks to use those tools—one study found that even when automated tool where available, up to 90% of code refactorings were performed manually \([55]\). Unlike refactoring tools, however, if an ODP system offers enough output-directed interactions to cover most of a programmer’s workflow, then it may be possible to switch the programming workflow from a primarily text-editing activity to a primarily direct manipulation activity—as evidence of this possibility, text edits in the code editor will not be used in the presentations of the SKETCH-N-SKETCH and MANIPSYNTH systems below.

As outlined above, output-directed programming addresses many of the problems faced by PBD systems. Why hasn’t ODP been more thoroughly explored? There are at least two challenges. First, maintaining two editable representations is non-trivial—most ODP systems therefore only support small code changes in limited scenarios as noted in [Section 2]. Second, raw output from a pro-
gram may hide how that output was generated, but the how may be what the programmer wants to manipulate—e.g., a large program that outputs a single number cannot be effectively modified just by indicating a desired change to that number. Because these are not small challenges, the work presented here focuses on expanding the expressive power of ODP, i.e., demonstrating new possibilities of what kinds of code can be created with output-directed interactions. Hence the thesis:

Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-directed programming interactions.

I aim to evaluate expressivity only—what kinds of programs can be created—and do not directly assess usability. I assume users are expert programmers familiar with code and that, for the reasons outlined above, ODP interactions are desirable. Expressivity will be demonstrated by creating example programs entirely by output-directed interactions, without ordinary text edits to the code (though, of course, these edits remain possible). These systems are introduced below.
4 Sketch-n-Sketch: ODP for SVG

This section summarizes work completed and published at UIST 2016 [27] and UIST 2019 [30]. Additionally, the system was awarded a Best Demo Honorable Mention at UIST 2019.

Direct manipulation tools for visual design lack the flexibility of a code-based tools such as Processing [4]. However, if the visual artist chooses a code-centric tool, they lose the affordances of direct manipulation. Could they have both the flexibility of programming and the power of direct manipulation?

In this section I introduce Sketch-n-Sketch, a program editor for creating programs that output SVG vector graphics. Sketch-n-Sketch imitates a traditional graphics editor, with tools for drawing and manipulating shapes. I introduce these tools and, even though text-based editing remains available at any time, I show how these tools allow a number of designs to be created entirely through output-directed interactions.

4.1 Related Graphical ODP

Like Sketch-n-Sketch, two prior systems offer direct manipulation graphical editing capabilities while realizing those manipulations in plain, editable code.

Transmorphic [67] re-implements the Morphic UI framework [50] using static, functional (i.e., stateless) views. Transmorphic retains Morphic’s ability to directly manipulate morphs (i.e., UI elements), but affects the manipulations by changing the view’s text-based code rather than changing live object state. The programmer may use direct manipulation to add morphs, remove morphs, or change a morph’s primitive properties.

Like the work presented here, APX [53, 54] 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.

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

4.2 Workflow

Sketch-n-Sketch aims to imitate a a traditional vector graphics editor, but with the design realized as textual code in a programming language. The work presented here builds on a prior initial version of Sketch-n-Sketch [10] which, once a program had been created by text edits, allowed the user to directly move and resize shapes and change colors, thereby changing appropriate numbers in the code. Below, this pre-existing ability to move shapes is assumed and I focus on this work’s novel program construction and editing facilities.

Figure 2 shows the Sketch-n-Sketch interface. The code box ➊ on the left is an ordinary source code text editor. The language in Sketch-n-Sketch is a simple, functional programming language—an extended lambda calculus with syntax inspired by the Elm programming language [17]. The canvas ➋ on the right displays the program’s SVG output. The toolbox ➌ offers various drawing tools for adding items into the program. The particular program shown in Figure 2 uses a recursive function to draw a von Koch snowflake fractal [83]. This program was constructed entirely through output-directed interactions.

To introduce these interactions, however, I present a simpler example: the construction of the Sketch-n-Sketch logo, shown at right. The workflow is summarized in Figure 3. After first drawing the needed shapes, the programmer will relate properties of the shapes (e.g., constraining the endpoints of the lines to match the corners of the square), and afterwards the shapes can be gathered into a group. Because the design is a program, the programmer can abstract the shapes into a function, allowing them to
reuse their design. Finally, the programmer may refactor the program, for example, clean up variable names and choose which shape properties should be arguments to the function. The final code for this example is shown in Figure 5. I walk through these steps below.

**Draw** The initial program template provided by SKETCH-N-SKETCH is nearly blank, defining only an empty list of SVG shapes:

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

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

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

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

The programmer similarly uses the “line” tool to add the needed lines (Figure 3a).

**Relate** The programmer would like to require that the endpoints of the lines always coincide with the corners of the rectangle. To relate these positions, the programmer first selects two points to be coincident (Figure 3b, top). Whenever a selection is made upon the canvas, SKETCH-N-SKETCH displays a floating menu of output tools, offering operations on the selected items (Figure 4).

To snap the points together, the programmer chooses the MAKE EQUAL tool. A submenu offers multiple ways to introduce variables into the program so that the points are always in the same position. The results differ in which original position is preserved—should the line move to the square, or vice versa? The first result moves the line to the square, and the programmer chooses it. A `topLeft` variable is introduced and used for both the positions of the square and the line:

```javascript
topleft = [158, 127]
square1 = square 150 topLeft 100
line1 = line 0 5 topLeft [276, 222]
```

If the programmer moves the square (thereby changing the numbers of the `topLeft` point), the endpoint of line will also move because the line’s endpoint uses the same variable.

The remaining endpoints can be similarly related with MAKE EQUAL. The positions of the other corners and center of the square must be calculated—for example, the right edge of the square is `x + w`. MAKE EQUAL automatically inserts the needed math (visible on lines 11-13 in the final code in Figure 5).

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

```javascript
... squareLineLine = [square1, line1, line2] svg (concat [ squareLineLine ))
```

Lists are represented on the canvas as a dotted border encompassing the list items (Figure 3, Figure 6). To aid comprehension, variable names are also shown on the canvas next to appropriate items—double-clicking on a name allows the variable to be renamed. In Figure 3, the programmer has selected the group list and renamed its variable to logo.

**Abstract** The programmer would like to make their logo design reusable—in other words, they would like to create a function that, given size and color arguments, generates a logo appropriately. With the logo (formerly squareLineLine) group list selected, the programmer invokes ABSTRACT from the output tools menu. ABSTRACT performs an ordinary Extract Method refactoring to produce a function that returns the selected item, namely [square1, line1, line2]. ABSTRACT heuristically pulls in variable bindings used only in the construction of [square1, line1, line2], resulting in the logoFunc visible on lines 7-14 of Figure 5.

The logo design is now reusable. To insert more copies of the design, the programmer could manually write additional calls to this function into the program. Conveniently, however, SKETCH-N-SKETCH’s type inference notices that this function accepts at least an x, y coordinate and a width, and this function appears in the toolbox as a custom drawing tool (Figure 3, top). The programmer uses the tool to draw two more copies of the function (Figure 3, bottom; Figure 5, lines 18 and 20).

**Refactor** The logoFunc produced by ABSTRACT’s heuristics was only parameterized on y, x, and w (and in that order). With this parameterization, copies of the design may differ in their position and size, but all copies must share the same colors. The programmer would like to change the parameterization of the function.

Items on the canvas that result from a function call are encompassed by a solid border (e.g., Figure 6a); the border represents the function call. Clicking the border focuses the function call (Figure 3). Other shapes on the canvas disappear and the programmer may edit the function. The arguments to the function also appear, as seen in Figure 3, and may be reordered, removed, or renamed. Arguments may also be added by selecting a property of a shape (e.g., its color, not shown) and invoking ADD ARGUMENT from the output tools menu. Additionally, focusing a function has a further consequence: using a drawing tool will add the new shape to the function instead of to the top level of the program.

In this example, the programmer modifies the arguments so that the function accepts x, y, w, (line) color and (line) strokeWidth, as seen on line 7 of Figure 5. The programmer is satisfied, resulting in the final code shown in Figure 5. The programmer was able to create the program entirely through output-directed manipulations on the canvas by using tools for drawing, relating, grouping, abstracting, and refactoring their design. A handful of additional SKETCH-N-SKETCH features not exercised by this example are discussed below.

**Intermediates** The goal of output-directed programming is to create a program by manipulating the execution products of the program. However, if only the final output of a program is shown, only the final output can be manipulated. Often, what is interesting is how that output was computed. To offer some opportunities for manipulating that how, SKETCH-N-SKETCH displays certain intermediate execution products on the canvas, summarized in Figure 6. The logo example above discussed how lists and function calls are displayed on the canvas. SKETCH-N-SKETCH will also display widgets on the canvas representing the points and the offsets encountered during program evaluation. Whenever the evaluator encounters a number-number pair during execution of the program, a point is placed on the canvas (Figure 6), allowing the programmer to select or move that point.

```javascript
let logoFunc x y w color strokeWidth = let topleft = [x, y] in let square1 = square 140 topleft w in let y2 = y + w in let line1 = line color strokeWidth topleft [x, y2] in let line2 = line color strokeWidth [x, y2] \ [ (2! * x + w) / 2!, (2! * y + w) / 2! ] in [square1, line1, line2] 8
```

Figure 5: Final code for the logo example. Numbers annotated with ! will not change when shapes are moved on the canvas. 10.
Additionally, in graphics code it is common to define offsets from some base \( x \) or \( y \) values. Therefore, during evaluation, when a numeric amount is added to or subtracted from an \( x \) or \( y \) coordinate, an offset arrow is drawn on the canvas (Figure 6b). The arrow may be selected, or dragged to change the offset amount.

The “Point or Offset” drawing tool in the toolbox allows the programmer to click to place new points on the canvas, or drag to create an offset. When creating the examples shown below in Figure 7, it was common to first lay out the desired parameterization of the design using points and offsets, creating a skeleton (as in the inset, for creating a rhombus). Afterwards, shapes can be attached to the points.

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

```plaintext
let w = 100 in
let x y w @ 5

nPointsOnSegment2 = nPointsOnSegment 3(0-18) [75, 326] 430 379

concatMap logoFunc2 nPointsOnSegment2
```

The REPEAT tool creates a function (logoFunc2) that produces a copy of the logo given a single point, and maps that function over the list of points that were drawn earlier (nPointsOnSegment2).

Figure 7: Examples created in SKETCH-N-SKETCH entirely through output-directed interactions.

### 4.3 Case Studies

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

### 4.4 Limitations

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

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

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

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

4.5 Summary

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

I implemented and demonstrated direct manipulation tools for drawing, relating, grouping, abstracting, and repeating shapes, as well as tools for refactoring the constructed program. SKETCH-N-SKETCH additionally exposes various intermediate execution products on the canvas for manipulation, so the programmer is not limited to manipulating their final output.

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

In the long term, I envision that the programming process might become as immediate and visual as direct-manipulation-based creativity applications—not just for shape-drawing programs but for non-visual programs as well. SKETCH-N-SKETCH is an early step on that journey. The remainder of the proposal below describes further steps towards this vision.
5 Tiny Structure Editors

This section describes work to appear as a short paper at VL/HCC 2020 [25].

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

Imagine the programmer has defined a custom data type for ranges on the number line. They have a value representing the range with a lower bound of negative infinity and an upper bound of 10, inclusive. For the system to display this value, one option (Figure 8a) is to draw a pointer graph showing which data structures point to others in the running program (as in, e.g., Python Tutor [25] or Kanon [57]). The goal, however, is not just to display the value but to also allow manipulation. The system might let the programmer grab and move the arrows representing pointers. Another option is to render a default textual representation for the value (Figure 8b). For manipulation, the system could overlay a UI on the string allowing the programmer to change different parts of the value’s internal structure.

Ideally, the system would offer visualizations that match the way the programmer thinks about the problem in their head. Neither a pointer graph nor the default textual representation are natural ways to represent contiguous intervals. Ideally the programmer could see and manipulate a natural representation for contiguous intervals, either the standard mathematical notation (Figure 8c) or a visualization on the number line (Figure 8d).

This ideal presents a challenge: the interval data type is a custom creation of the programmer. While it is fairly straightforward for the programmer write a toString (or a toSVG) function to produce a custom visualization, it is much more difficult to make a manipulable visualization. For manipulation, the programming environment might allow the programmer to code their own interactive UI. For common types, such as colors or regular expressions [59], this effort might be worth the trouble. But interactive UI programming is not easy, and may not be worth the effort for less frequently used custom types such as the interval example here.

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

**Tiny Structure Editors (TSE)** In this work, I design a system, called TSE, that given a toString function for a custom data type, automatically generates tiny structure editors for manipulating values of that type.

To do so, TSE instruments the execution of the toString function applied to a value, and then overlays UI widgets on top of appropriate locations in the output string (Figure 10). To determine these locations, TSE employs two key technical ideas: (a) a modified string concatenation operation that preserves information about substring locations and (b) runtime dependency tracing (based on Transparent ML [2]) to relate those substrings to parts of the input value.

While some prior systems [84, 68] trace string operations so that the programmer can edit strings in the output to thereby modify strings in the program, these systems can only modify literal strings in the code. TSE instead relates the output string to any original value of interest, not just strings. Thus, TSE allows the user to modify not just strings but also numbers and custom data values.

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

5.1 Algebraic Data Types (ADTs)

Somewhat analogous to inheritance in object-oriented languages, algebraic data types (ADTs) enumerate the variants of a type and the data associated with each variant [62]. Unlike an object, an ADT value is raw data, separate from the functions that operate on it. Because ADTs succinctly describe the variants of plain data, ADTs are beginning to appear in mainstream languages: “enums” in Swift and Rust are ADTs, as are “case classes” in Scala and “discriminated unions” in Typescript.

Figure 9 shows three ADT definitions comprising a
Based on the value’s type, the appropriate constructor for the custom interval data type. The lower bound of an interval (Begin) has two variants representing whether the bound is negative infinity (NegInf()) or finite (After(Num, Bool)). If finite, the bound records the finite boundary number and a boolean indicating whether the boundary is or is not included in the interval (is or is not closed). The type describing upper boundaries (End) is similar. An interval (Interval) is a lower and upper boundary together. The first word of each variant (NegInf(), After, Before, Inf, Interval) is a constructor which acts as a function to create a value of the ADT. The last line of Figure 9 uses these constructors to create an interval value representing \((-\infty, 10]\). Data inside ADT values is extracted using “pattern matching” in case splits (i.e., switch statements) which define the handling of alternative variants, as shown in the `toString` functions in Figure 9.

### 5.2 Algorithm

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

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

\[
\text{Interval}(\text{NegInf()}, \text{Before}(10)^\{2.1\} \cdot, \text{True}^{(2.2)})(2.3)^\{2\} \cdot)^\{\}^\cdot)
\]

Based on the value’s type, the appropriate `toString` function is invoked on the value of interest and the tracing evaluator propagates the dependency tags. Additionally, in TSE, string concatenation operations (`++`) do not produce a new, flattened string. Instead, the concatenation is deferred, resulting in a binary tree of substrings when evaluation completes (Figure 10\(a_1\)). Because of the tracing evaluator, each substring and each concatenation carries a set of projection paths, relating parts of the string to parts of the value of interest (Figure 10\(a_2\)).

**Spatial Regions**  
In the final display, selection regions and UI widgets will be overlaid on top of the rendered string. To generate the selection regions, the string concatenation binary tree is translated into a binary tree of nested 2D polygons, with each polygon encompassing the spatial region of the associated substring (Figure 10\(b\)).

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

**Selection regions**  
When the user moves their cursor over the rendered string, the deepest (equivalently, smallest) region under their mouse is offered for selection/deselection. For the interval example, there are four possible selection regions, shown in Figure 10\(c\). Selection is currently inert, but the selection regions are the basis for po-
Editing base values Literal numbers or strings from the value of interest may pass through to the output unchanged, for example the number 10 in the interval example. TSE lets the user manipulate these values. The user may click a number and drag up and down to scrub the number to a different value. Both numbers and strings can be double-clicked to reveal a standard text box to text edit the value.

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

For recursive ADTs such as lists or trees, TSE additionally draws buttons to insert or remove items from the data structure, as shown in Figure 11 for a list. Remove buttons are associated with item(s) to be removed. Insert buttons are trickier to position—TSE must predict where an item not currently in the data structure will appear. This prediction is occasionally imprecise, as discussed below.

Finally, in some cases, multiple buttons would be rendered in identical locations. Such overlapping buttons are coalesced into a single button that opens a menu offering the different transformations.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Subvalues</th>
<th>%Selectable</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>80% (4/5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>100% (4/4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON (multiline)</td>
<td>33% (14/43)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>86% (6/7)</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>List (&quot;[&quot; in base case)</td>
<td>100% (7/7)</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>List (via join)</td>
<td>71% (5/7)</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>List (via different join)</td>
<td>86% (6/7)</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>Tree (S-exp)</td>
<td>53% (10/19)</td>
<td>100% (5/5)</td>
<td></td>
</tr>
<tr>
<td>Tree (indented hierarchy)</td>
<td>21% (4/19)</td>
<td>100% (5/5)</td>
<td></td>
</tr>
<tr>
<td>Pair</td>
<td>100% (3/3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>100% (7/7)</td>
<td>100% (3/3)</td>
<td></td>
</tr>
<tr>
<td>ADT (recursive)</td>
<td>100% (4/4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record</td>
<td>100% (9/9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>100% (4/4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: Case studies of hand-written and translated toString functions.

5.3 Case Studies

TSE’s goal is to provide low- to no-cost domain-specific value editors. I tested TSE on toString functions for a number of datatypes, measuring several properties of the generated editors as shown in Figure 12. Figure 12 reports the percentage of ADT subvalues that could be directly selected (i.e., were not occluded, missing, or sharing a selection region with other subvalues). For data types representing containers (e.g., lists or sets), Figure 12 reports the percentage of contained items that can be selected.

To provide evidence that TSE can operate on unmodified toString functions, I translated several toString functions from Haskell’s standard libraries to TSE’s Elm-like language, as shown in the bottom half of Figure 12. These translations were performed as literally as possible.

Manual inspection of the case studies revealed a few issues to address in subsequent versions of TSE. Most notably, zero-width regions such as those from empty strings are ignored, which for some variants of list toString caused the final Nil() to be un-selectable. Additionally, selection region sharing and occlusion are sometimes troublesome. Two subvalues sharing the same selection region is a less of an issue—depending on the application, selecting a shared region could offer to operate on any of the associated items. Occlusion, however, results in certain subvalues being un-selectable. One solution might be to expand ancestor regions by a few pixels. Finally, insert buttons were not placed in reasonable locations for tree-like data structures, but, as discussed next, how best to handle actions is a domain-specific consideration.

Figure 11: Traditional list ADT definition and a generated GUI for the list Cons(1, Cons(2, Cons(3, Cons(4, Nil()))))).
5.4 Discussion

Although TSE is intended to support future output-directed programming systems (such as MANIPOSYNTH below), the TSE prototype here is a standalone demonstration system. While this independent implementation highlights TSE’s key techniques, applying TSE to a particular application will require a number of further design decisions, particularly surrounding the handling of actions. For example, consider the set data structure in Figure 12. The reference implementation [13] is based on a tree and maintains a number of invariants such as balancing, ordering, and non-duplication. None of these invariants are expressible in a standard ADT definition alone, and the internal tree structure is not exposed in the toString output (“fromList [2,3,5,7]”). Therefore, only some of TSE’s selection regions are relevant—namely, the terminal items, as reported in Figure 12—and the structural transformations generated by TSE are not meaningful because they do not enforce the set invariants. TSE does not yet provide an interface for specifying custom insert and remove functions, instead I imagine such an interface would be part of a larger, future IDE.

As implemented here, the TSE prototype has another minor limitation. Systems that rely on string tracing [84, 68] provide custom implementations of string manipulation functions that correctly propagate dependencies. TSE currently only support string concatenation and string length—supplementing the language with additional string functions remains future work.
6 MANIPOSYNTH: ODP for Functional Data Structures

This section describes proposed work.

Sketch-N-Sketch demonstrated that output-directed programming (ODP) can be used to produce non-trivial vector graphics programs without the need to resort to text-based editing. Could similar interactions be applied to non-visual, general-purpose programming? Moreover, there have been a number of recent advances in programming by examples (PBE) for synthesizing code in functional languages [61, 20, 48], but these works focus on the technical synthesis mechanism and not the user interface—could direct manipulation form the basis for a practical PBE interface? To explore the combination of direct manipulation and PBE synthesis, I propose an ODP system called (The Magnificent) MANIPOSYNTH for creating programs that perform common functional data structure manipulations.

Because output-directed programming for non-visual domains remains largely unexplored, there are a number of open questions. What kinds of manipulations are possible? What kinds of program changes are conveniently specified via output-directed programming interactions? Which types of algorithms can be created?

Below I describe MANIPOSYNTH and the interactions it affords before discussing prior ODP systems for general-purpose programming and then speculating on possible challenges to the creation of MANIPOSYNTH.

6.1 Design

The goal of MANIPOSYNTH is to explore what kinds of functional data structure manipulation programs can be created by output-directed interactions. A couple principles inform the specific design of MANIPOSYNTH.

First, like Sketch-N-Sketch, one goal is to embrace ordinary programming concepts. In Sketch-N-Sketch, applying this principle had some surprising results. For example, Sketch-N-Sketch does not have a graphics-specific concept of a “group” as in most graphics editors—ordinary lists serve the purpose well. Similarly, the Abstract tool does not need to perform domain-specific rewrites—an ordinary Extract Method refactoring is sufficient. By similarly sticking to the basics of core functional programming, MANIPOSYNTH may discover similar elegant gains.

A second design principle is to embrace non-linear editing. The structure of text-based code encourages a top-to-bottom order on program construction, whereas the actual creative process of the human brain is likely more non-linear. The ideal system will let the programmer “dump the parts bucket onto the floor” [81] and then arrange the parts how they see fit.

Non-linear editing complements functional programming. Because functional values are stateless, the dependency structure is significant but the order of execution is less so. In contrast, in a stateful, imperative environment the order of operations matters and must be carefully tracked (like the forward/backward stepper buttons in ALVIS Live [35]). The compatibility of non-linear editing and functional programming was elegantly explored in the visual programming system Eros by Elliott [16]. In Eros, values are displayed on a canvas (Figure 13). These values are usually partially applied functions. Example values are used for the arguments to these functions, and example output of each functions is shown. Functions may be graphically composed by direct manipulation: so long as the types match, the output of one function may be used as the argument for another. The result of the composition is a new value—again, usually a function, whose arguments are the unapplied arguments from its constituents. Elliot calls these composable visualizations tangible values (TVs).

MANIPOSYNTH similarly envisions a non-linear canvas of values and thus adopts the nomenclature of tangible values (TVs) for its representation of program products. Figure 14 shows the proposed MANIPOSYNTH interface. As in Sketch-N-Sketch, ordinary, text-editable code appears on the left, and program products on the right. The programmer is working with a binary tree data type 1. They have used the visual editor to construct a example tree 2 named t, which they are currently using as they work on their current goal: specifying a height function for the binary tree data type 3. In this context, below I discuss the various interactions MANIPOSYNTH will offer.

Figure 13: Composing tangible values (TVs) in Eros. The output of the middle TV (a function) is to be used as the second input to the left TV (also a function), resulting in the TV on the right. This resulting TV is also a function, with several remaining unapplied arguments. (Reproduction of Figure 14 from Elliott [16].)
Maniposynth!

1. data BTree<a>
   = Leaf(a)
   | Node(BTree<a>, BTree <a>)

   height : ?«a» → ?

   height(tree) =
   let
     t2 = ?«Synthesizing…»(Node(…))
     in
     ?
   height(t2)

   In:
   t =
   Out:
   height =
   Node
   Leaf  2
   Leaf  3
   height(t2)

   Drag to extract; Delete to remove. The tree t is being used as the example input to the height function: as indicated next to “In:”, the function argument tree is taking

Multiple examples on a visualization. Visualizations in MANIPOSYNTH will be composable, with customizable layering. The example value t is being displayed as a top-down tree. Layers may be added to the visualization: programmer has elected to show the result of applying the height function to each type-compatible item in the tree, i.e., each subtree. The upper left corner of the subtrees shows these results. Because the height function is unfinished, the height function evaluates to ? for each subtree.

One aim of MANIPOSYNTH is to provide programming by examples (PBE) interactions. For two of the subtrees, the programmer has asserted what height should evaluate to (shown in bold over pink): Leaf 2 should have height 1 and the right child of the root should have height 2. These assertions will serve as input-output examples for the PBE synthesizer inside MANIPOSYNTH and will eventually help guide the completion of the height function (although there is not enough information yet to complete the function). This interaction is noteworthy: existing PBE synthesizers [61, 20, 48] require a new example value for each input-output example which is tedious and hard-to-read when dealing with larger structures like trees. As demonstrated here, MANIPOSYNTH allows multiple assertions within a single visualization, which is more convenient and easier to read.

Drag to extract; Delete to remove. The tree t is being used as the example input to the height function: as indicated next to “In:”, the function argument tree is taking

on the value t. The programmer is working on specifying the height of the root of the tree. The height of the root is based on the heights of its immediate children. To obtain the height of the right child, the drags the height of the right child from the visualization into the function body (red arrow), producing a new tangible value. The right subtree has been automatically named t2 and the tangible value is thus the expression height(t2). The result of the expression (2, in this case) is displayed below the expression. The right subtree, however, is not in scope yet—only the root is input to the function—so the right subtree t2 must also be extracted. A new TV for this synthesis subgoal is automatically created.

Tangible values in MANIPOSYNTH can take different forms, as indicated by the location of the pink background (Figure 15). In TV, the expression takes priority with its value displayed below. In TV, the output value is asserted and the goal is to synthesize an appropriate expression to produce that value.

Figure 14: MANIPOSYNTH interface.

Figure 15: Tangible value (TV) forms in MANIPOSYNTH. (a) An optionally named expression displaying its output value, or (b) an assertion (desired value) guiding automatic synthesis of an expression.
Each TV on the canvas is associated with one definition in the code. Names are automatically generated as needed, but can be changed manually. Hole expressions, rendered ?, may be associated with an assumed result expression surround by « ». In Figure 14 the synthesizer is still searching for a candidate expression. Any expression (not just holes) may be associated with an asserted result value, surrounded by /uni27EA /uni27EB. (In the code, each assertion also stores the appropriate execution environment, which is hidden for brevity.)

In this scenario, the programmer dragged a derived value (the height of the right subtree) into the function body. MANIPSYNTH will also support dragging out a subvalue to indicate that subvalue should be extracted. Similarly, selecting a subvalue and pressing “Delete” will indicate that the a new value should be produced, albeit with the subvalue removed.

Click to type. The programmer also needs the height of the left subtree. They may drag out the appropriate value from the visualization as before. MANIPSYNTH will also support creating TVs by typing arbitrary expressions. The programmer may double click on the canvas and type arbitrary code to create a new TV for the expression. This simple interaction has a few benefits. The programmer need not choose a name for the expression, nor do they need to choose a location in the code, and because a new TV is created and its value displayed, the programmer can type any expression they would like to see the result of, even if they don’t plan to use the expression later. Additionally, the PBE synthesizer will preferentially use TVs that appear on the canvas—typing out a TV is a way to prove a hint to the synthesizer.

Auto-complete to existing value. While typing an expression, an autocomplete menu offers values and subvalues on the canvas. As shown in Figure 16 this allows the programmer to reference the left subtree (Leaf 1) even though there is not yet any variable in scope containing that value. As before, this interaction will automatically create a synthesis goal TV to extract that left subtree.

Construct desired output. Above, the programmer typed an expression on the canvas to produce a TV where the provided expression had priority (as in Figure 15a). Although not demonstrated in this example workflow, typing on the canvas may alternatively indicate a desired value, producing instead an assertion TV and thereby triggering a search for an appropriate expression (as in Figure 15b).

Select and operate. Once the programmer has acquired the heights of the left and right subtrees, they want to take the max. Any value on the canvas may be selected. When one or more values is selected, a menu appears offering possible operations on the selected items. There will usually be many such operations, so the menu offers a search box. In Figure 17 the programmer filters the operations down to those involving “max” in order to create a new TV that is the max of the two heights.

Case splits. The finished height function will need to operate differently depending on whether the root of the input tree is an internal node or a leaf. That is, a case split is required. In particular, if the input tree is a leaf, then it does not have a left and right child—the TVs to extract those children will be invalid. MANIPSYNTH takes the view that worrying about case splits is part of the top-down linear nature of textual code. To support non-linear thinking, MANIPSYNTH lets the programmer defer the introduction of case splits. The synthesizer may decide a case split is necessary and insert a case split into the code. On the canvas, however, the programmer may use subvalues as if they are already there, even if those subvalues have not yet been extracted via a case split. If a different example input takes a different branch, any TVs thereby inactive for the example are simply grayed out.

6.2 Evaluation

The goal of MANIPSYNTH is to demonstrate the feasibility of output-directed programming for data structure manipulation in functional programming. The main ques-
tion is qualitative: what kinds of data structure manipulations are well served by output-directed interactions?

As in SKETCH-N-SKETCH and TSE, this question will be answered by case studies. Like SKETCH-N-SKETCH, the goal will be to create the desired programs without resorting to ordinary text editing in the code box—although text editing will be available at any time. That said, this constraint is a bit looser for MANIPOSYNTH because the TVs on the canvas support ordinary text editing of their expressions. Even so, the larger argument is that ODP and PBE interactions provide a compelling workflow.

MANIPOSYNTH will be applied to enough diverse examples so that arguments can be made about the approach’s strengths and weaknesses. Possible functional algorithms to re-implement are not in short supply. Examples can be drawn from Okasaki’s classic text [58] and other sources.

6.3 Related Value-Centric Programming

A handful of prior systems espouse a workflow of program construction by manipulating a display of the nonvisual, primitive values within the program. These systems may be classified as programming by demonstration (PBD) systems that do not represent the manipulations in always-editable text, and output-directed programming (ODP) systems that do.

Value-Centric PBD David Canfield Smith’s 1975 Ph.D. thesis on Pygmalion [70] not only introduced the now-ubiquitous UI concept of “icons”, but used those icons to offer a general-purpose programming by demonstration (PBD) environment—indeed, Pygmalion was also the first PBD environment [71]. In Pygmalion, the user demonstrated a desired algorithm step-by-step by manipulating icons of the live values in scope. When all arguments to a function call were supplied, the icon for the function call was replaced with a display of its result value. To use that result value, the user dragged the value to where they wanted to use it. Recursion was supported.

Pygmalion envisioned that the recorded demonstration was sufficient and did not offer a corresponding always-editable text representation.

Like Pygmalion, Pictorial Transformations (PT) [33] also offered program construction via step-by-step manipulation of a display of the live program values. PT offered custom visualizations, so that, e.g., a boolean flag on an object could be represented by whether the object was drawn with a thick or thin border. Predicates specified by the user could reference these graphical properties directly (e.g., “is the border thick?”). PT allowed the user to express more complicated algorithms than Pygmalion, including those involving lists.

In the SmallTalk tradition, Self [78] displayed live objects graphically and allowed a user to send messages to those objects via direct manipulation (demonstrated in video form in [73]). Although value-centric, the interactions provided by Self and related systems like the Morphic UI framework [50] differ from the systems above in that manipulations in Self-like systems modify state, not the algorithm.

Non-Graphical ODP A few existing output-directed programming systems have targeted ordinary non-graphical programming, with varying degrees of expressivity offered by their output-directed interactions.

JDial [34] records variable values during execution of an imperative Java program and allows the programmer to directly change incorrect values in the execution trace. The desired corrections specify a program repair problem. JDial then uses sketch-based synthesis [72] to make changes to the program in order to satisfy the user’s direct manipulation of the trace. JDial is limited to small program repairs and does not offer program construction features.

Vital [26] visualizes algebraic data type (ADT) values in a Haskell notebook environment. Ordinary copy-paste operations may be performed on sub-values of the visualized ADTs. These copy-paste operations are realized by changing the textual code in the appropriate notebook cell. Vital explored interpreting copy-paste in alternative ways—by value or by reference—but overall the kinds of code editing tasks that can be accomplished by copy-paste of subvalues is limited.

ALVIS Live [35] is an educational programming environment for helping students learn to write iterative array algorithms, such as findMax. Like SKETCH-N-SKETCH and MANIPOSYNTH, ALVIS Live incorporates a two-pane editor, with a code box and an area displaying the values in the program. The programmer may step forward and backward in time to see how values change during execution. Additionally, a toolbox is available allowing the programmer to draw operations on the canvas, and thereby insert new code into the program. Unlike the system proposed below, ALVIS Live does not utilize program synthesis. Additionally, the ALVIS Live environment and the commands in its textual language are both specialized for iterative array operations—it is unclear how its output-directed tools might be more broadly applicable.

6.4 Challenges

Besides the labor of building an interactive programming system, implementing MANIPOSYNTH faces many possible challenges.

MANIPOSYNTH will rely on holes to represent incomplete portions of the program. How to run a program with holes has been described [60]. But MANIPOSYNTH holes can also carry candidate expressions and therefore holes...
might also be used to indicate which portions of the pro-
gram are open to modification by the synthesizer—any
portion of the program wrapped in « » is open to auto-
matic modification. It is unknown whether this mecha-
nism alone will be a reasonable way to denote which parts
of the program can and cannot be modified. For exam-
ple, introducing a case split can cause significant restruc-
turing of the existing code and might require that the en-
tire function be considered open to modification; the case
split introduction might fail because of the existence of
a single expression not wrapped in « ». Also, it remains
an open question whether automatically introducing case
splits will make the programmer feel like their code is not
under their control.

Although programming by examples synthesis (PBE)
has been described for functional languages \[61, 20\], in-
cluding in the presence of holes \[48\], MANIPOSYNTH
may require additional synthesis constraint semantics.
The drag-to-extract interaction described above implies a
very specific constraint: the programmer wants specific
value \(x\) from location \(l\) within data structure \(y\). This con-
straint is an assertion about the evaluation trace that pro-
duced \(x\). Although supporting such trace constraints is
not likely a complicated addition, existing PBE synthe-
sizers for functional languages do not yet support trace
constraints.

Additionally, because TVs have multiple forms (Fig-
ure 15), questions about the particular semantics of
TVs are likely to arise. If such questions arise,
MANIPOSYNTH will need to choose an interpretation of
TVs that is simple and predictable.

Finally, MANIPOSYNTH could suffer from limited ap-
application. Although MANIPOSYNTH will likely excel for
some data structure manipulation tasks, it is an open ques-
tion whether the approach is suitable for a wide variety of
tasks or only a few cherry-picked scenarios.
7 Future Work

SKETCH-N-SKETCH, TSE, and MANIPOSYNTH invite a number of possible future directions for research.

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

Editors from toSvg functions. TSE instruments the execution of a toString function to produce an interactive editor for a custom data type. The same tracing scheme could instead be applied to a user-specified toSvg function, thus automatically turning a graphical representation, such as the interval on the number line in Figure 8d, into an interactive editor. A further intriguing direction is to also extend SKETCH-N-SKETCH so that it may used to help construct these toSvg functions.

Spreadsheet interactions for data sets. MANIPOSYNTH focuses on interactions on single data values. Larger data sets might be more more conveniently displayed and manipulated in a spreadsheet-style view. Flowsheets [8] demonstrates how expression outputs might be visualized using a spreadsheet layout, albeit without program synthesis. Using input-output examples, program synthesis has been commercially realized in spreadsheets with Excel’s FlashFill [24]. More recently, FlashFill was adapted to Pandas dataframes to produce readable Python code in Jupyter notebooks [14], albeit as a code generator and not quite in a bidirectional manner as in the systems above. A complete and fluid bidirectional spreadsheet with strong program synthesis capabilities does not yet exist.

8 Conclusion

Output-directed programming (ODP) shows promise as a mechanism for more naturally specifying programs, but the paradigm is under-explored. To push the boundaries of ODP, I offer two ODP systems and one supporting mechanism.

SKETCH-N-SKETCH is a development environment for creating programs that output vector graphics. With SKETCH-N-SKETCH it is possible to create parametric designs without ordinary text editing—although ordinary text editing is always available.

TSE is a mechanism for automatically generating structure editors for a custom data value, given only a toString function for that type. TSE can thus support future ODP systems by allowing the programmer to easily create manipulable, domain-specific representation of their custom data types.

Finally, MANIPOSYNTH is a proposed system for output-directed programming of functional data structure algorithms. MANIPOSYNTH will explore a number of novel techniques for incorporating programming by example and non-linear editing.

These software systems will thereby justify my thesis:

Non-trivial vector graphics programs and functional data structure manipulation programs can be constructed by output-directed programming interactions.
References


