Programmatic and Direct Manipulation, Together at Last

Ravi Chugh Brian Hempel Mitchell Spradlin Jacob Albers

University of Chicago, USA

{rchugh, brianhempel, mhs, jacobalbers} @ uchicago.edu

Abstract

Direct manipulation interfaces and programmatic systems have distinct and complementary strengths. The former provide intuitive, immediate visual feedback and enable rapid prototyping, whereas the latter enable complex, reusable abstractions. Unfortunately, existing systems typically force users into just one of these two interaction modes.

We present a system called SKETCH-N-SKETCH that integrates programmatic and direct manipulation for the particular domain of Scalable Vector Graphics (SVG). In SKETCH-N-SKETCH, the user writes a program to generate an output SVG canvas. Then the user may directly manipulate the canvas while the system immediately infers a program update in order to match the changes to the output, a workflow we call *live synchronization*. To achieve this, we propose (i) a technique called *trace-based program synthesis* that takes program execution history into account in order to constrain the search space and (ii) heuristics for dealing with ambiguities. Based on our experience with examples spanning 2,000 lines of code and from the results of a preliminary user study, we believe that SKETCH-N-SKETCH provides a novel workflow that can augment traditional programming systems. Our approach may serve as the basis for live synchronization in other application domains, as well as a starting point for yet more ambitious ways of combining programmatic and direct manipulation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logics and Meanings of Programs*]: Program Analysis; H.5.2 [*Information Systems Applications*]: User Interfaces

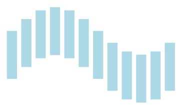
Keywords Sketch-n-Sketch, Prodirect Manipulation, SVG

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908103

1. Introduction

Direct manipulation user interfaces [29] such as Adobe Illustrator, Microsoft PowerPoint, and GIMP [17] and programmatic systems such as Processing [3, 28] each have distinct strengths. The former provide intuitive, immediate visual feedback and enable rapid prototyping, whereas the latter allow for designing and reusing complex abstractions. Neither mode of interaction is preferable for all tasks.

Motivation. Imagine designing a series of identical shapes, laid out along the contours of a sine wave. Designing the first "prototype" shape using a visual, direct manipulation tool like Illustrator or PowerPoint works well. But after copying the shape, pasting it multiple times becomes tedious, and achieving the sinusoidal pattern requires painstaking effort because built-in features (*e.g.* rulers, snapping to other shapes, and uniform spacing) are unlikely to help. Worse yet are the edits required to change high-level parameters of this design (*e.g.* the number of shapes, the spacing between them, or the amplitude of the sine wave). The user will want to scrap the previous effort and start from scratch. On the other hand, by using a programmatic system, it is easy to churn out variations of the high-level pattern by changing parameters and re-running the program. But the disconnect between the software artifact and its output can limit the pace of design, especially when the appropriate parameters in the program are difficult to identify.



In a recent position paper, we identified several domains for both visual and textual data where users are forced to make the unfortunate choice between programmatic or direct manipulation tools [12]. We proposed the notion that software systems ought to smoothly integrate these two modes of use, a combination we call *prodirect manipulation*. Ideally, a user could manipulate the output of a program and the system would simultaneously update the program to keep it synchronized with the changes. In this paper, we present the first realization of that goal.

Challenges. Suppose that a program e generates k output values and that a user action changes the first j values: $\{v_1 \rightsquigarrow v'_1, \ldots, v_j \rightsquigarrow v'_j, v_{j+1}, \ldots, v_k\}$. To synthesize a program e' that generates the updated output, three primary challenges must be addressed: (1) The meaning of the user's updates must be defined in a way that constrains the program synthesis search space; (2) The search algorithm must run quickly and find program updates that are easy for the user to understand; and (3) When multiple candidate solutions exist, the ambiguity must be dealt with in a way that facilitates the responsive, interactive workflow characteristic of direct manipulation interfaces.

Key Idea 1: Trace-Based Program Synthesis. To address challenge (1), we propose a technique called *trace-based program synthesis* that comprises two components. First, we instrument the evaluation of the program e so that each of the k values v_i it produces comes with a trace t_i, which forms a set of constraints $\{v_1 = t_1,\ \ldots,\ v_k = t_k\}$ relating the program to its output. Second, to reconcile e with the j updated values, we specify that an ideal candidate update would be a program e' whose output satisfies the system of constraints $\mathcal{C} \mathrel{\hat{=}} \{v'_1 = t_1,\ \ldots,\ v'_j = t_j,\ v_{j+1} = t_{j+1},\ \ldots,\ v_k = t_k\}$.

Key Idea 2: Small Updates. To address challenge (2), our design principle is that only "small" program updates may be inferred; "large" changes may require user intervention and are thus less conducive to our goal of immediate synchronization. In particular, we attempt only to change value literals in the program, represented as a *substitution* ρ that maps locations in the abstract syntax tree to new values.

Key Idea 3: Heuristics for Disambiguation. Even with the seemingly modest goal of inferring small updates, there is still the problem of dealing with multiple solutions. To address challenge (3), we first decompose $\mathcal{C}$ into k separate constraints $\mathcal{C}_1 \mathrel{\hat{=}} \{v'_1 = t_1\}$ through $\mathcal{C}_k \mathrel{\hat{=}} \{v'_k = t_k\}$ and then use heuristics to force each $\mathcal{C}_i$ to have at most one solution ρ_i. This enables us to define a *trigger* function $\lambda(v'_1,\ \ldots,\ v'_j).\ \rho$ that, given the concrete values changed by the user, computes a solution ρ_i for each constraint $\mathcal{C}_i$ and combines them into a single substitution $\rho \mathrel{\hat{=}} \rho_1 \cdots \rho_k$. This substitution is applied to the original program in real-time during the user action, resulting in a new program ρe that is evaluated and ready for subsequent user manipulation.

Contributions and Outline. These three key ideas are general and may be developed for several settings. In this paper, we focus on instantiating our approach for the specific domain of Scalable Vector Graphics (SVG). In particular:

- We propose a framework for inferring program updates called *trace-based synthesis*, which may apply broadly to domains where users wish to (indirectly) manipulate programs by (directly) manipulating their outputs. (§3)
- We define heuristics for disambiguating between possible updates and define *triggers* that keep a program synchronized with changes to its SVG output in real-time. (§4)
- We implement our ideas in a Web-based system called SKETCH-N-SKETCH and evaluate its interactivity. (§5)
- We use SKETCH-N-SKETCH to design many examples difficult to develop or edit using existing tools. (§6)

Our implementation and examples, as well as additional tutorial documentation and videos, are publicly available at `http://ravichugh.github.io/sketch-n-sketch`.

2. Overview

We will provide an overview of SKETCH-N-SKETCH by considering the program in Figure 1 that generates the sine wave box design in § 1. Suppose the user clicks on the third box and drags it to a new position down and to the right (depicted in Figure 1C). In this section, we will describe how SKETCH-N-SKETCH synthesizes four candidate updates to the program, each of which computes the position of the third box to match the user's direct manipulation but have different effects on the remaining boxes (depicted in Figure 1D).

A Little Programming Language. The programming language in SKETCH-N-SKETCH is a core, untyped functional language called `little`, which includes base values (floating-point numbers, booleans, and strings) and lists (represented as cons-cells, or pairs). For reference, we define the syntax of `little` in Figure 2. All expression and value forms are standard except for numbers.

Because numeric attributes are often directly manipulated in graphical user interfaces, each numeric literal in `little` comes with three pieces of additional information: a *location* ℓ, which is an integer inserted by the parser that identifies its position in the abstract syntax tree (AST); an optional, user-specified *location annotation* α, where `!` is used to *freeze* the number; and an optional, user-specified *range annotation* β, where `{` n_1 `-` n_2 `}` denotes the intended range of the number. We discuss these annotations further later in this section.

Representing SVG Values. An *SVG node* is represented as a list `[ svgNodeKind attributes children ]` with three values where the string `svgNodeKind` describes the kind of the node (*e.g.* `'rect'`, `'circle'`, or `'line'` for particular shapes, or `'svg'` for a canvas, or collection, of shapes); `attributes` is a list of attributes (*i.e.* key-value pairs); and `children` is a list of child nodes. The intended result of a `little` program is a node with kind `'svg'`. The result of evaluating the program in Figure 1B takes the form below, which gets translated directly to the SVG format [35] and resembles the screenshot in § 1 when rendered.

```
['svg' []
  [ ['rect' [['x'  50] ['y' 120] ...] []]
    ['rect' [['x'  80] ['y'  90] ...] []]
    ['rect' [['x' 110] ['y'  68] ...] []] ... ] ]
```

Our translation of key-value attributes provides a thin wrapper over the target SVG format, allowing `little` programs to specify arbitrary XML attributes using strings. We discuss the translation further in Supplementary Appendices [13], but the details are not needed in the rest of the paper.

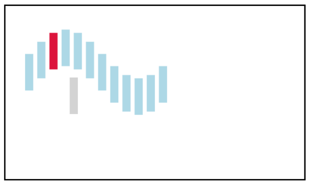
(A) Excerpt from `prelude.little`

```
(defrec range (λ(i j)
  (if (> i j) nil
      (cons i (range (+ 1^ℓ1 i) j)))))

(def zeroTo (λn (range 0^ℓ0 (- n 1))))
```

(B) `sineWaveOfBoxes.little`

```
(def [x0 y0 w h sep amp] [50 120 20 90 30 60])
(def n 12!{3-30})
(def boxi (λi
  (let xi (+ x0 (* i sep))
  (let yi (- y0 (* amp (sin (* i (/ twoPi n)))))
    (rect 'lightblue' xi yi w h)))))

(svg (map boxi (zeroTo n)))
```

(C) Suppose the user clicks on the third box from the left (colored darker in red for emphasis) and drags it to a new position down and to the right (colored lighter in gray):



(D) SKETCH-N-SKETCH synthesizes four candidate updates to the program, which have the following effects:

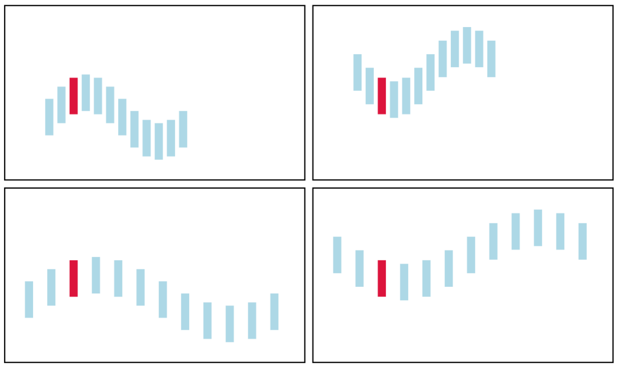


Figure 1. Sine Wave of Boxes in SKETCH-N-SKETCH

2.1 Locations and Traces

To enable direct manipulation, the `little` evaluator produces run-time *traces* to track the evaluation of numeric values (for attributes such as `'x'`, `'y'`, `'width'`, and `'height'`). There are two kinds of traces t that are used to infer program updates based on direct manipulation changes.

Locations. The simplest kind of trace records that a numeric literal n originates from a particular source-code location ℓ in the AST of the program. For the program in Figure 1B, the `little` parser annotates all numbers $\mathtt{50}^{\ell_2}$, $\mathtt{120}^{\ell_3}$, $\mathtt{20}^{\ell_4}$, $\mathtt{90}^{\ell_5}$ $\mathtt{30}^{\ell_6}$, $\mathtt{60}^{\ell_7}$, and $\mathtt{12}^{\ell_8}$ with unique location identifiers. These locations do not affect program evaluation or SVG rendering. When the user manipulates a numeric value n^ℓ in the canvas (*e.g.* by dragging or stretching a box), SKETCH-N-SKETCH updates the value at program location ℓ in real-time.

In the rest of the paper, when a number n is immediately bound to a variable `x`, we choose the canonical name x for the location — resulting in the value n^x — rather than one of the form ℓ_k. In the examples that follow, we sometimes annotate numeric literals explicitly with locations for explanatory purposes even though the programmer does not write them — they are inserted implicitly by our parser.

Expression Traces. Smaller traces are combined into larger ones during the evaluation of primitive operations, such as addition, multiplication, *etc.* Whereas the `'width'` and `'height'` attribute values originate from atomic AST locations, the `'x'` value of each box is the result of evaluating `xi` $\triangleq$ `(+ x0 (* i sep))` with different bindings for `i`. Each run-time value that is bound to `i` is generated by the function `zeroTo` (from a `Prelude` library included in every program), which computes the list of integers from `0` to `n-1`.

When evaluating a primitive operation, the run-time semantics of `little` records the structure of the expression along with the resulting value. Below are the values of `xi` for each of the first three boxes, respectively, and their corresponding traces. Each value-trace pair forms an equation:

$$\mathtt{50} = \mathtt{(+}\ x_0\ \mathtt{(*}\ \ell_0\ sep\mathtt{))} \tag{1}$$

$$\mathtt{80} = \mathtt{(+}\ x_0\ \mathtt{(*}\ \mathtt{(+}\ \ell_1\ \ell_0\mathtt{)}\ sep\mathtt{))} \tag{2}$$

$$\mathtt{110} = \mathtt{(+}\ x_0\ \mathtt{(*}\ \mathtt{(+}\ \ell_1\ \mathtt{(+}\ \ell_1\ \ell_0\mathtt{))}\ sep\mathtt{))} \tag{3}$$

The locations ℓ_0 and ℓ_1 identify literals `0` and `1`, respectively, in the `Prelude` function `zeroTo` that computes increasing integers. The value-trace equations for the remaining boxes are analogous. These equations, together with the following substitution that records location-value mappings from the source program, relate the program to its output.

$$\rho_0 \triangleq [x_0 \mapsto \mathtt{50}, sep \mapsto \mathtt{30}, \ell_0 \mapsto \mathtt{0}, \ell_1 \mapsto \mathtt{1}, \ \ldots]$$

Dataflow-Only Traces. The reduction rule E-OP-NUM (Figure 2) builds expression traces in parallel with the evaluation of primitive operations, producing values n^t. Traces t record data flow but not control flow. This design is based on the insight that programs generating output in visual domains are often structured so that the control flow of the program is similar across multiple runs. We have not found this limitation to be a problem in practice for the examples we have developed. However, it may be useful to enrich traces with additional information in subsequent work.

2.2 Synthesizing Program Updates

The main idea behind *trace-based program synthesis* is to use value-trace equations in order to infer program updates that conform to output values changed by the user. In the setting of direct manipulation interfaces, changing attributes of visual objects corresponds to changing the values on the left-hand side of the equations above. In our `sineWaveOfBoxes` example, the result of dragging the third box directly to the right so that its new `'x'` value is `155`, is to replace Equation 3 with the following equation:

$$\texttt{155} = (\texttt{+}\, x_0\ (\texttt{*}\ (\texttt{+}\,\ell_1\ (\texttt{+}\,\ell_1\ \ell_0))\ sep)) \qquad (3')$$

Our goal is to synthesize an updated program that satisfies Equation 3'; satisfying all other (unchanged) equations would be ideal, but this one is "more important" because it was induced by the user's change.

Local Updates. We aim only to infer *local updates*, which are substitutions that map locations to updated numeric values. We describe design and implementation decisions in § 4 and § 5 that limit the cost of solving equations to ensure responsive interaction with the user. With our approach, SKETCH-N-SKETCH can infer four substitutions based on Equation 3': $\rho_1 \doteq \rho_0[x_0 \mapsto \texttt{95}]$, $\rho_2 \doteq \rho_0[sep \mapsto \texttt{52.5}]$, $\rho_3 \doteq \rho_0[\ell_0 \mapsto \texttt{1.5}]$, and $\rho_4 \doteq \rho_0[\ell_1 \mapsto \texttt{1.75}]$.

These substitutions, if applied to the program in Figure 1B, would produce various effects. The first option would translate all of the boxes in unison. The second would increase the spacing between boxes. The third and fourth would, respectively, translate all boxes or the change the separation, but both would also change the number of boxes because the constants `0` and `1` at locations ℓ_0 and ℓ_1 were used in the original program to compute integer indices. The user is unlikely to want either the third or fourth options, since the list of integers `0` to `n-1` specifies the number of boxes. Moreover, the locations ℓ_0 and ℓ_1 appear in the `Prelude`, so changing their values would affect the behavior of other programs!

Frozen Constants. By default, SKETCH-N-SKETCH will consider updating the value of any program location used in a value-trace equation in order to reconcile the program with the user's changes. The user can direct the synthesis procedure not to change the value of particular constants by *freezing* them, denoted with exclamation points (*e.g.* `3.14!`). All numbers in `Prelude` are automatically frozen, so solutions ρ_3 and ρ_4 are not actually considered by SKETCH-N-SKETCH. Without freezing either `x0` or `sep`, however, the ambiguity between ρ_1 and ρ_2 remains.

2.3 Heuristics

Pausing and asking the user to choose between updating x_0 or sep would stymie the interactive, *live synchronization* we strive for. Instead, we employ heuristics for automatically resolving ambiguities that attempt to strike a balance between interactivity and predictability. Our key insight is that the essence of a local update is the set of constants $\mathcal{L}$ (*i.e.* program locations) that are changed and *not* necessarily their new values. Even if we arbitrarily decide that a particular user action should cause a set $\mathcal{L}_1$ of constants to change rather than a set $\mathcal{L}_2$, we can often assign a different user action to manipulate the constants in $\mathcal{L}_2$.

If the user drags the first box horizontally to a new `'x'` position n, the change would induce the value-trace equation $n = (\texttt{+}\, x_0\ (\texttt{*}\ \ell_0\ sep))$ based on Equation 1, which can be solved by changing the value of either x_0 or sep. In preparation, we arbitrarily choose to update x_0 to n in order to solve the new equation if and when the user drags this box. If the user drags the second box, again, either x_0 or sep could be changed in order to solve the equation $n = (\texttt{+}\, x_0\ (\texttt{*}\ (\texttt{+}\,\ell_1\ \ell_0)\ sep))$ based on Equation 2. Because we already assigned x_0 to vary if the user drags the first box, we choose to vary sep if the user drags the second box. If the user drags the third box, again, either x_0 or sep could be changed to solve the induced equation $n = (\texttt{+}\, x_0\ (\texttt{*}\ (\texttt{+}\,\ell_1\ (\texttt{+}\,\ell_1\ \ell_0))\ sep))$ based on Equation 3. Because each location has already been assigned to vary in response to some user action, we arbitrarily choose x_0.

We continue to assign program updates in this fashion by trying to balance the number of times a location set is assigned to some user action in the canvas. When the user performs an action, we use the pre-determined location set, together with the concrete values from the mouse manipulation, to compute a local update (*i.e.* substitution), apply it to the original program, run the resulting new program, and render the new output canvas.

2.4 Sliders

There are several situations in which it can be difficult or impossible to make a desired change to the program via direct manipulation: (1) when there is no natural, visual representation of the program parameter of interest; (2) when a visual representation may be hard to directly manipulate, for example, because the shape is too small or there are too many adjacent shapes; and (3) when ambiguous updates are hard to resolve using the built-in heuristics or by deciding which constants to freeze or not.

SKETCH-N-SKETCH provides a feature that can help in all three situations. If a number is annotated with a range, written $n^{\ell}\texttt{\{}n_{min}\texttt{-}n_{max}\texttt{\}}$, then SKETCH-N-SKETCH will display a slider in the output pane that can be used to manipulate the n value between n_{min} and n_{max} (as opposed to having to edit the program). Therefore, sliders can provide control over otherwise hard-to-manipulate attributes.

In the `sineWaveOfBoxes` example, the number of boxes `n` is the only parameter that is hard to directly manipulate. The heuristics choose to update `n` in response to certain actions, but the effects are not intuitive. Therefore, on line 2 of Figure 1B, we freeze the value of `n` (so that it will never change as a result of directly manipulating the boxes) and declare the range `{3-30}` so that we can easily adjust the number of boxes using a slider.

Syntax of Expressions

$$
\begin{array}{rcl}
e & ::= & N \mid s \mid b \mid \texttt{[]} \mid \texttt{[}e_1\,\texttt{|}\,e_2\texttt{]} \mid x \\
& \mid & (\lambda\, p\, e) \mid (e_1\, e_2) \mid (op_m\, e_1 \cdots e_m) \\
& \mid & (\texttt{let}\, p\, e_1\, e_2) \mid (\texttt{letrec}\, p\, e_1\, e_2) \\
& \mid & (\texttt{case}\, e\, (p_1\, e_1) \cdots (p_m\, e_m)) \\
p & ::= & x \mid n \mid s \mid b \mid \texttt{[]} \mid \texttt{[}p_1\,\texttt{|}\,p_2\texttt{]} \\
N & ::= & (n^\ell, \alpha, \beta) \quad \alpha ::= \cdot \mid \texttt{!} \quad \beta ::= \cdot \mid \{n_1 \texttt{-} n_2\} \\
op_0 & \in & \{\texttt{pi}, \ldots\} \\
op_1 & \in & \{\texttt{not}, \texttt{cos}, \texttt{sin}, \texttt{arccos}, \texttt{round}, \texttt{floor}, \ldots\} \\
op_2 & \in & \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{<}, \texttt{=}, \texttt{mod}, \texttt{pow}, \ldots\}
\end{array}
$$

Syntax of Values

$$
\begin{array}{rcl}
v, w & ::= & n^t \mid s \mid b \mid \texttt{[]} \mid \texttt{[}v_1\,\texttt{|}\,v_2\texttt{]} \mid (\lambda\, p\, e) \\
t & ::= & \ell \mid (op_m\, t_1 \cdots t_m)
\end{array}
$$

Operational Semantics (excerpted) $\boxed{e \Downarrow v}$

$$
\frac{n = [\![\,(op_m\, n_1 \cdots n_m)\,]\!] \qquad t = (op_m\, t_1 \cdots t_m)}{(op_m\, {n_1}^{t_1} \cdots {n_m}^{t_m}) \Downarrow n^t}\ \text{[E-Op-Num]}
$$

Figure 2. Syntax and Semantics of `little`

3. Trace-Based Program Synthesis

In the previous section, we defined the semantics of `little` to produce run-time traces for numeric attributes. In this section, we formulate *trace-based program synthesis* as a way to define the relationship between a program and updates to its output, without regard to the particular domain.

User Actions. Suppose that a `little` program e generates output that contains k numeric values. With a single action, the user may directly manipulate j of the numeric values, for some $1 \leq j \leq k$. The following table shows how, as alluded to in §1, the j updated values, together with the $k-j$ unchanged ones, form a system of constraints that, ideally, an updated program e' would satisfy:

Program	Output	Updates	Constraints	
	$w_1 = {n_1}^{t_1}$	$\rightsquigarrow n'_1$	$n'_1 = t_1$	(hard)
	$\cdots$	$\cdots$	$\cdots$	(hard)
$e \;\Rightarrow$	$w_j = {n_j}^{t_j}$	$\rightsquigarrow n'_j$	$n'_j = t_j$	(hard)
	$\cdots$		$\cdots$	(soft)
	$w_k = {n_k}^{t_k}$		$n_k = t_k$	(soft)

The user's changes may lead to an unsatisfiable set of equations (when considering only local updates). We treat equations induced by changes as "hard" constraints that a solution ought to satisfy, whereas the rest are "soft" constraints that should be satisfied if possible. This design principle prioritizes explicit changes made by the user, which is the goal of our workflow. Next, we will formally define what constitutes a valid solution to a system of constraints.

Contexts and Substitutions. We define a *value context* V below to be a value with $m > 0$ placeholders, or *holes*, labeled $\bullet_1$ through $\bullet_m$. We define the application of a value context to a list of values as $V(v_1, \cdots, v_m) \;\hat{=}\; V[v_1/\bullet_1]\cdots[v_m/\bullet_m]$. A *substitution* ρ is a mapping from program locations ℓ to numbers n. When applied to an expression, the bindings of a substitution are applied from left-to-right. Thus, the rightmost binding of any location takes precedence. We use juxtaposition $\rho\,\rho'$ to denote concatenation, and we write $\rho \oplus (\ell \mapsto n)$ to denote $\rho[\ell \mapsto n]$. We define *value context similarity* below to relate values that are structurally equal up to the values of numeric constants.

$$
V ::= \bullet_i \mid n^t \mid s \mid b \mid \texttt{[]} \mid \texttt{[}V_1\,\texttt{|}\,V_2\texttt{]} \mid (\lambda\, p\, e)
$$

$$
\frac{}{V_1 \sim V_1} \qquad \frac{}{{n_1}^t \sim {n_2}^t} \qquad \frac{V_1 \sim V'_1 \quad V_2 \sim V'_2}{\texttt{[}V_1\,\texttt{|}\,V_2\texttt{]} \sim \texttt{[}V'_1\,\texttt{|}\,V'_2\texttt{]}}
$$

Definition: Faithful Updates. If

(a) $e \Downarrow v$, where $v = V(w_1, \ldots, w_k)$; and

(b) the user updates $w_1, \ldots, w_j$ to $w'_1, \ldots, w'_j$,

then a substitution ρ is *faithful* if

(c) $\rho e \Downarrow v' = V'(w''_1, \ldots, w''_k)$ where $V' \sim V$; implies

(d) $w''_i = w'_i$ for all $1 \leq i \leq j$.

Premises (a) and (b) identify the list of j values manipulated by the user, and properties (c) and (d) capture the notion that hard constraints induced by these changes should be satisfied by the update ρ. The value similarity relation checks that two value contexts are structurally equal but says nothing about the soft constraints from the original program (namely, it does not say $w''_i = w_i$ for all $j < i \leq k$). In a setting where multiple updates are synthesized, ranking functions could be used to optimize for soft constraints.

It is important to note that our definition states "(c) *implies* (d)" rather than the stronger property "(c) *and* (d)" because the control flow may change and produce $V' \not\sim V$. We choose the weaker version because we do not intend to reason about control flow either in traces or our synthesis algorithm (§5) when considering how one program compares to another. In other settings, it may be worthwhile to require the stronger version, which would necessitate a richer trace language that records control-flow information.

Definition: Plausible Updates. We define an alternative, weaker correctness criterion. In particular, we define a *plausible update* to be one that satisfies *some* (*i.e.* at least one) of the user's updates. Concretely, a plausible update is defined just like a faithful one, except that the following condition replaces (d) in the original definition:

(d') $w''_i = w'_i$ for some $1 \leq i \leq j$

The general framework presented in this section can be instantiated with solvers (which we will refer to as Solve) that aim for different points along this spectrum of faithful and plausible updates.

4. Live Synchronization for SVG

Given changes to the output of a program, in the previous section we defined how value-trace equations can be used to specify candidate program updates in order to reconcile the changes. In this section, we describe how to compute program updates in real-time for the specific domain of Scalable Vector Graphics (SVG). First, we identify what constitutes a user action in this setting. Second, we formulate how to compute *triggers* that dictate program updates based on such actions. For the latter, we propose heuristics to automatically resolve ambiguities that result from trace-based program synthesis problem instances.

In the following, we write $v[\texttt{'k'}]$ to refer to the value of attribute `'k'` in the `little` SVG value v. We also define the abbreviations $\mathsf{Num}(n^t) \triangleq n$ and $\mathsf{Tr}(n^t) \triangleq t$.

User Actions. Consider a value r that represents a rectangle positioned at $(r[\texttt{'x'}], r[\texttt{'y'}]) = (n_x{}^{t_x}, n_y{}^{t_y})$. Suppose the user clicks the mouse button somewhere inside the borders of r (rendered visually) and then drags the cursor dx pixels in the x-direction and dy pixels in the y-direction. As a result, the new desired position of r is given by $(n'_x, n'_y) = (n_x + dx, n_y + dy)$. Our goal is to reconcile this change to the position of r with the original program that generated it. One option is to wait until the user finishes dragging the rectangle, that is, when the user releases the mouse button. At that point, we could invoke $\mathsf{Solve}(\{n'_x = t_x, n'_y = t_y\})$ to compute a set of substitutions. Our goal with live synchronization, however, is to immediately a apply program update during the user's actions.

4.1 Mouse Triggers

When the user clicks on a shape, we compute a *mouse trigger* $\tau = \lambda(dx, dy).\ \rho$, which is a function that, based on the distance the mouse has moved, returns a substitution to be immediately applied to the program.

For now, let us assume that all shapes are rectangles and that user actions manipulate only their `'x'` and `'y'` attributes. For every shape r_i in the canvas, there are two steps to compute a mouse trigger. First, for each attribute `'x'` and `'y'`, we choose exactly one number (*i.e.* location) in the program to modify *before* the user initiates any changes to $(r_i[\texttt{'x'}], r_i[\texttt{'y'}])$. The results of this step are two univariate equations to solve. Second, we define a mouse trigger that invokes the solver with each equation and then combines their resulting substitutions. Once mouse triggers have been computed for all shapes, the editor is prepared to respond to any user action with a local update to the program. We will now describe each step in detail.

Shape Assignments. Our task is to determine a *shape assignment* γ that maps each shape to an attribute assignment. We define an *attribute assignment* θ to map attribute names (*i.e.* `little` strings) to program locations. We refer to the range of an attribute assignment as a *location set*.

Let box_i refer to each rectangle from `sineWaveOfBoxes` in left-to-right order. Using a procedure Locs to collect all non-frozen locations that appear in a trace, we see that the `'x'` and `'y'` attributes are each computed using two locations: $\mathsf{Locs}(\mathsf{Tr}(box_i[\texttt{'x'}])) = \{x_0, sep\}$ and $\mathsf{Locs}(\mathsf{Tr}(box_i[\texttt{'y'}])) = \{y_0, amp\}$. As a result, there are four possible attribute assignments for each shape:

$$\begin{aligned}
\theta_1 &\triangleq [\texttt{'x'} \mapsto x_0, \texttt{'y'} \mapsto y_0] \\
\theta_2 &\triangleq [\texttt{'x'} \mapsto x_0, \texttt{'y'} \mapsto amp] \\
\theta_3 &\triangleq [\texttt{'x'} \mapsto sep, \texttt{'y'} \mapsto y_0] \\
\theta_4 &\triangleq [\texttt{'x'} \mapsto sep, \texttt{'y'} \mapsto amp]
\end{aligned}$$

These assignments correspond to the four options (top-left, top-right, bottom-left, and bottom-right, respectively) depicted in Figure 1D.

"Fair" and Other Heuristics. As described in §2, our default strategy is to choose an attribute assignment whose range (*i.e.* location set) has not yet been assigned to any other shape in the output canvas. When all possible assignments have been chosen an equal number of times (*i.e.* when they have been treated "fairly"), then we arbitrarily choose. As a result, we "rotate" through each of the four attribute assignments, assigning $\gamma(box_i) = \theta_j$, for all i where $j = 1 + (i \bmod 4)$.

The fair heuristic will not always make choices that the user would prefer best. However, we find that even simple heuristics such as this one already enable a large degree of desirable interactivity. Therefore, designing more sophisticated heuristics could be a fruitful avenue for future work. In Supplementary Appendices [13], we describe a second heuristic that we have implemented, which "biases" towards program locations that are used in relatively few run-time traces and, thus, have fewer opportunities to be assigned to a zone. We will not discuss this alternative in detail, because the vast majority of the examples we have written to date, including the ones discussed in this paper, work at least as well using the fair heuristic.

Computing Triggers. The next task is to prepare for when the user might click a box_i in the output and drag it dx pixels in the x-direction and dy pixels in the y-direction.

Let ρ_0 be the mapping from locations to numbers in the original `sineWaveOfBoxes` program and let γ_0 be the shape assignment computed using the heuristics described above. For each box_i, we evaluate the helper procedure $\mathsf{ComputeTrigger}(\rho_0, \gamma_0, box_i)$, where SolveOne is a solver that is given exactly one univariate equation to solve:

$$\begin{aligned}&\mathsf{ComputeTrigger}(\rho, \gamma, v) \mathrel{\hat{=}} \\ &\quad \lambda(dx, dy).\ \rho \oplus (\ell_x \mapsto \mathsf{SolveOne}(\rho, \ell_x, n'_x = t_x)) \\ &\qquad\qquad \oplus (\ell_y \mapsto \mathsf{SolveOne}(\rho, \ell_y, n'_y = t_y))\end{aligned}$$

$$\text{where}\quad {n_x}^{t_x} = v[\texttt{'x'}] \quad n'_x \mathrel{\hat{=}} n_x + dx \quad \ell_x = \gamma(v)(\texttt{'x'})$$
$$\qquad {n_y}^{t_y} = v[\texttt{'y'}] \quad n'_y \mathrel{\hat{=}} n_y + dy \quad \ell_y = \gamma(v)(\texttt{'y'})$$

When the user drags some box_i, its new attribute values n'_x and n'_y (directly manipulated by the user) are used to solve the value-trace equations using the locations assigned by $\gamma(v)$. This substitution is then applied to the original program, the new program is run, and the new output is rendered as the user moves the mouse. When the user releases the mouse button, we compute new shape assignments and mouse triggers in anticipation of the next user action.

Recap: Design Decisions. There are two aspects of our approach that warrant emphasis. The first that is we choose exactly one location to modify per updated attribute, even though there may be additional solutions (*i.e.* local updates) that modify multiple locations. For example, Equation 3' can also be satisfied by the substitution $\rho_0[x_0 \mapsto 55][sep \mapsto 20]$ (among many others). By considering only "small" local updates, however, we reduce the space of possible updates to synthesize and choose from.

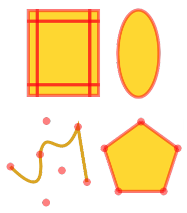
The second is that our solutions are only plausible, not faithful, because the same location may appear in multiple attributes being directly manipulated (and, therefore, multiple equations). For example, consider the box generated by the following expression:

```
(let xy 100 (rect 'red' xy xy ... ...) ...
```

The attribute assignment $[\texttt{'x'} \mapsto xy, \texttt{'y'} \mapsto xy]$ is the only one to consider, but the corresponding system of constraints on xy is overconstrained; the new values computed by $\mathsf{SolveOne}(\rho, xy, 100 + dx = xy)$ will differ from $\mathsf{SolveOne}(\rho, xy, 100 + dy = xy)$ whenever $dx \neq dy$. We could choose to apply an update only when the individual solutions agree, or, more conservatively, disallow the shape from being manipulated at all. Instead, we simply apply the individual substitutions in an arbitrary (implementation-specific) order, which has the effect of satisfying at least one of the constraints imposed by the user action. This approach trades synthesizing only faithful updates in exchange for additional opportunities to directly manipulate output values.

4.2 Other Shapes and Zones

For the purposes of presentation, so far we described a single type of user action, namely, dragging the interior of a rectangle. In practice, there are many other kinds of user actions. For each kind of SVG shape, we define *zones* that identify and name particular visual areas of a shape that can be directly manipulated by the user in order to affect particular attributes. The screenshot below depicts zones for several kinds of shapes.



As we have described, dragging the INTERIOR zone of a rectangle allows the user to manipulate its `'x'` and `'y'` attributes. Not all zones are tied to exactly two attributes, however. For example, the rectangle RIGHTEDGE zone is tied to one attribute (`'width'`) and the BOTLEFTCORNER zone is tied to three (`'x'`, `'width'`, and `'height'`). Furthermore, not all attributes vary *covariantly* with dx or dy. For example, when the user manipulates the BOTLEFTCORNER of a rectangle, the `'width'` attribute varies *contravariantly* with dx (and, at the same time, `'x'` varies covariantly). Nevertheless, the approach we described for assigning triggers for INTERIOR zones generalizes in a straightforward way to the remaining shapes and zones. One slight change is that shape assignments are indexed by shape and zone, for example, $\gamma(v)(\text{INTERIOR})(\texttt{'x'})$. We provide more details in Supplementary Appendices [13].

5. Implementation

We have implemented SKETCH-N-SKETCH (available at `http://ravichugh.github.io/sketch-n-sketch`) in approximately 6,000 lines of Elm [16] and JavaScript code. When the user hovers over a zone, our implementation displays a caption that indicates whether the zone is "Inactive" or "Active" and, for the latter, identifies the constants (*i.e.* location set) that will change if the user manipulates it. Furthermore, we highlight these constants in yellow before the user begins manipulating the zone; in green while they are being updated during manipulation; and in red if the solver fails to compute a solution based on the user's update. We use gray to highlight constants that contributed to an attribute value but were not selected by the heuristics.

In the rest of this section, we describe the simple value-trace equation solver that we currently use and we evaluate the overall interactivity of our tool. In Supplementary Appendices [13], we describe additional features of our implementation.

5.1 Solving Value-Trace Equations

The mouse triggers defined in the previous section require a procedure $\mathsf{SolveOne}(\rho, \ell, n' = t)$ that, given the substitution ρ from the previous program and a location ℓ, computes a new value for ℓ that satisfies the equation $n' = t$. Currently, we implement a simple solver that supports only "single-occurrence" equations, where the location ℓ being solved for occurs exactly once. Our top-down procedure uses the inverses of primitive operations to recursively solve a univariate equation in a syntax-directed manner (see [13] for details). Not all primitive operations have total inverses, so SolveOne sometimes fails to compute a solution.

As we will discuss below, supporting this syntactic class of equations is already enough to enable program synthesis for a variety of interesting examples. Our solver is easy to implement and deploy in our Web-based setting and fast enough to provide interactivity. Future work, however, may incorporate more powerful solvers (such as MATLAB or Z3 [14]) while taking care to ensure that synthesis is quick enough to incorporate into an interactive, portable, direct manipulation editor.

5.2 Interactivity

The goal of SKETCH-N-SKETCH is to provide immediate, live synchronization updates in response to direct manipulation changes. For a user action to be "successful" requires that the particular zone be Active, that the solver computes an update in response to the mouse manipulation, and that the resulting update is applied to the program and re-evaluated within a short period of time. We discuss each of these aspects in turn based on measurements collected from 68 `little` programs of varying complexity, spanning more than 2,000 lines of code in total. Below, we discuss summary statistics across all examples; for reference, detailed tables can be found in [13].

5.2.1 Active Zones

For any particular zone, our assignment algorithm may consider zero, one, or more candidate location assignments based on the traces of its attributes. A zone is Inactive when there are zero candidates and is Active otherwise. Across all of our examples, there were a total of 3,772 shapes with 14,106 zones, of which 991 (7%) were Inactive and 13,115 (93%) were Active.

Zones	**14,106**	
Inactive	991	7%
Active	13,115	
Unambiguous	4,856	34%
Ambiguous	8,259	59%

Ambiguity. Among Active zones, 4,856 (34% of all zones) had exactly one candidate location assignment and 8,259 (59% of all zones) had more than one (3.83 candidates on average). To provide responsive interaction, it is important to deal with ambiguities because they are so frequent. Our heuristics resolve ambiguities without user intervention. It may be fruitful to explore other approaches, such as showing multiple options for the user to choose from (particularly when there are relatively few), or allowing the user to make multiple user actions before attempting to infer an update.

5.2.2 Solving Equations

Next, we evaluate the solvability of equations that correspond to Active zones. Consider a program with initial location substitution ρ and shape assignment γ, and a shape v with an active zone ζ. For each attribute `'k'` that ζ controls, $\gamma(v)(\zeta)(\texttt{'k'})$ identifies a location ℓ to update in order to solve the equation $n + d = t$, where n^t is the original value of $v[\texttt{'k'}]$, ℓ is one of the locations in t, and d is the change dictated by a user action. Across all examples, there are 28,222 such $(\rho, v, \zeta, \ell, n, t)$ tuples. Because traces are often shared by multiple shapes and zones, we filter out tuples that are identical modulo v and ζ, leaving 4,574 unique (ρ, ℓ, n, t) tuples. In the following, we refer to each of these tuples as a "pre-equation."

Unique Pre-Equations	**4,574**	
Outside Fragment	919	20%
Inside Fragment	3,655	
No Solution for $d = 1$	194	4%
Solution for $d = 1$	3,461	
No Solution for $d = 100$	438	10%
Solution for $d = 100$	3,023	66%

Syntactic Fragment. The majority of pre-equations (3,655, which constitutes 80%) fall into the syntactic fragment handled by our solver. We paid little attention to the structure of traces when writing examples, so we have been surprised that this number is so high. We fully expected to incorporate a more full-featured solver early in our work, but we have been able to leave this to future work without severely hampering the examples we have written so far.

The remaining 919 (20%) pre-equations fall outside the fragment and are guaranteed not to be solvable. Our current attribute assignment algorithm does not take this into consideration and will sometimes assign such pre-equations to a zone. It would be worthwhile to avoid making such choices in the future.

Solvability. For each pre-equation (ρ, ℓ, n, t), we would like to know whether the solver can compute an update if the user manipulates the given attribute to be $n + d$. Rather than symbolically analyzing the space of possible user changes, we tested $\mathsf{SolveOne}(\rho, \ell, n + d = t)$ with two concrete values, namely, $d = 1$ and $d = 100$. Of the 3,655 pre-equations in the fragment, 3,461 were solvable for $d = 1$ (*i.e.* a green highlight) and the remaining 194 (4% of all unique pre-equations) were not (*i.e.* a red highlight). Note that simply computing an update does not necessarily mean that the change is acceptable to the user.

Of the 3,461 pre-equations solvable for $d = 1$, 3,023 (66% of all unique pre-equations) were also solvable with $d = 100$. The remaining 438 (10% of all pre-equations) were not. Upon inspection, several of these equations are of the form $n + d = f(\cos \ell)$, where f is some function of $\cos \ell$. Because the cosine function is bounded to the range $[-1, 1]$, the equation does not always have a solution. Indeed, there is a mismatch between the interpretation of user updates in the Cartesian plane and attributes like rotation that have more natural representations in other coordinate systems. In our experience, we have found that manipulating rotation angles in SKETCH-N-SKETCH often works better with explicit sliders or using separate built-in rotation zones in our implementation, which we have not described in the paper.

5.2.3 Performance

In our experience, SKETCH-N-SKETCH is responsive for many, but not all, of our examples. We have not attempted to measure the observed frame rate of SKETCH-N-SKETCH, which depends on several factors beyond our implementation. We have, however, measured the performance of four critical aspects of our implementation: parsing and evaluating a program, preparing for a user action, and solving a pre-equation. We performed our experiments on an Intel Core i7 (four cores, 2.6-GHz) running Mac OS X 10.9.5. For "Parse," "Eval," and "Prepare," we tested the operation five times on every example using Firefox 45 and five times on every example using Chrome 49. For "Solve," we tested the operation on Chrome 49 twice per pre-equation across all examples. The "Min" and "Max" columns report the minimum and maximum times across all runs; "Med" and "Avg" report the median and average across all runs. Detailed statistics by example may be found in [13].

Operation	**Min**	**Med**	**Avg**	**Max**
Parse	9 ms	53 ms	77 ms	520 ms
Eval	<1 ms	5 ms	12 ms	165 ms
Prepare	1 ms	13 ms	200 ms	6,789 ms
Solve	<1 ms	<1 ms	<1 ms	14 ms

As the user drags the mouse during direct manipulation, SKETCH-N-SKETCH repeatedly solves the trace equations for the zone being manipulated and re-evaluates the program to immediately display the interaction results. The average time to "Solve" each trace equation is negligible, <1 ms on average, because our solver uses a simple, syntax-directed procedure. Re-evaluation takes longer, 12 ms on average. Our implementation re-runs the entire program even though much of the output may not change. In the future, it would be useful to optimize the implementation to recompute only the parts of the program needed (*e.g.* [11]).

The slowest operations reported above, "Parse" and "Prepare," are not run during direct manipulation. "Prepare" encapsulates the computation of both shape assignments and triggers for all zones. We only perform this computation when the program is run initially and after the user finishes dragging a zone. Some of the data structures and algorithms we use for computing candidate location assignments and choosing from among them are rather naive and can be optimized in the future.

6. Examples

We have used SKETCH-N-SKETCH to implement a variety of designs. In this section, we will highlight observations that pertain specifically to the combination of programmatically defined graphics and direct manipulation. The implementations resemble typical programs in other functional languages, but for the domain of SVG.

6.1 Programmatic Abstractions

Our current implementation does not allow new shapes to be added directly using the GUI. Nevertheless, we have used SKETCH-N-SKETCH to effectively program and manipulate several designs that would be difficult to edit or maintain using existing direct manipulation tools such as Illustrator and PowerPoint. Figure 3 provides thumbnails for some of the examples we will discuss.

Variables as Abstractions. SKETCH-N-SKETCH does not attempt to infer any abstractions. It only propagates abstractions that result from shared constants in the program. Therefore, our `little` programs are structured to use variables (bound to constants) to encode explicit relationships between attributes. Once these relationships have been defined, the SKETCH-N-SKETCH editor preserves them during direct manipulation. Many examples benefit from using variables as abstractions, such as: our SKETCH-N-SKETCH logo, which comprises three black polygons evenly spaced by white lines; the logo for the Chicago Botanic Garden (`www.chicagobotanic.org`), which contains several Bézier curves reflected across a vertical axis; the Active Transportation Alliance logo (`www.activetrans.org`), which uses several points along a path to depict a city skyline; and a logo adapted from the Lillicon [4] project, where several curves are used to define a semi-circle. For each example, a single direct manipulation update changes all related attributes, without the need for any secondary edits.

Derived Shapes. It is useful to define abstractions on top of the primitive SVG shapes. We define an `nStar` function (and include it in `Prelude`) that creates an `n`-sided star centered at `(cx,cy)` and rotated `rot` radians in the clockwise direction, where the distance from the center to the outer points is `len1` and the distance to the inner points is `len2`.

```
(def nStar
  (λ(fill stroke w n len1 len2 rot cx cy) ...))
```

We use `nStar` to implement the City of Chicago flag, which contains four evenly-spaced six-sided stars. By directly manipulating the POINT zones of a star in live mode, we can control the outer and inner distances of all four stars. Modifying length parameters this way can be surprising. For example, using negative lengths leads to interesting patterns, even though one might not think to try them when programming without immediate visual feedback.

Group Box Pattern. We occasionally find it useful to create a transparent rectangle in the background with the width `w` and height `h` of an entire design. Then, the BOTRIGHTCORNER zone of this box will, predictably, be assigned the location set $\{w, h\}$. If we define all other shapes relative to

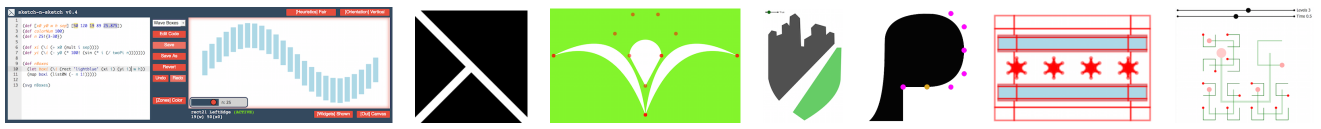


Figure 3. Examples (left to right): Sine Wave of Boxes, SKETCH-N-SKETCH Logo, Chicago Botanic Garden Logo, Active Transportation Alliance Logo, Icon from Lillicon [4], City of Chicago Flag, Hilbert Curve Animation

w and h, we gain direct manipulation control over the size of the entire design. In future work, it may be useful to provide built-in support for grouping shapes.

Dealing with Ambiguities. We often start programming a design with all constants unfrozen except those that are not design parameters, such as 2 in the expression `(* 2! (pi))`. Then, after seeing how direct manipulation induces changes, we edit the program to freeze some constants. Finally, to deal with any remaining undesirable automatic choices from the heuristics, we add range annotations to certain numbers so that we can unambiguously and easily manipulate them with sliders instead.

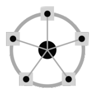
We performed a preliminary study (described in Supplementary Appendices [13]) that demonstrates the existence of scenarios (A) where using sliders is preferable to relying on heuristics for disambiguation, and (B) where relying on heuristics is preferable to using sliders. A systematic user study would be a useful direction for future work.

"Animations." In several examples, like for the rendering of Hilbert curves, we use sliders to control the SVG design as a function of a numeric parameter. The effect is that we can "animate" the design as we manipulate the slider. In the future, we plan to support dynamic, time-varying animations as language and editor primitives.

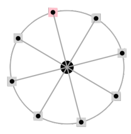
Procedural vs. Relational Constructions. There is a trade-off between procedural programming (in a functional language like `little`) and constraint-oriented programming (in a system like SketchPad [32]). A detailed comparison of programming graphic designs in these two styles may be an interesting avenue for future work.

6.2 Detailed Case Study: Ferris Wheel

To provide a sense of when direct manipulation works smoothly, and how to deal with situations when it does not, we discuss one of our examples in more detail. In this design, we manipulate a ferris wheel comprising a number of equal-length spokes emanating from a central hub, each of which has a passenger car at its end.



Phase 1: Initial Development. In Figure 4A, we define a `little` program that embodies our design; for now, ignore the parts typeset in boxed blue. We define several parameters on lines 1 and 2: the center `(cx,cy)` of the wheel; the number `numSpokes` and length `spokeLen` of the spokes; the radius `rCenter` for the central disc; the width `wCar` of each passenger car; the radius `rCap` of each car's hubcap; and the rotation `rotAngle` for the entire design. We draw several components of the wheel on lines 4 through 11 using circles and rectangles, and we draw the spokes in terms of the `nStar` function described earlier. The `cars` are defined so that they remain vertical even when the wheel is rotated, in order to accurately portray the physical characteristics of a ferris wheel in motion. The visual rendering of the output is shown above.



Phase 2: Direct and Programmatic Edits. Suppose we wish to edit the program so that the output resembles the picture on the right. In particular, we will adjust the size and location of the wheel, the size of the passenger cars, the number of spokes, the rotation angle, and the color of the first car. These changes will require a combination of programmatic and direct manipulation edits.

First, we want to change the size and location of the wheel. When we hover over the INTERIOR of the `rim`, SKETCH-N-SKETCH shows a caption to indicate that cx and cy will be updated. When we hover over the EDGE of the `rim`, we see that $spokeLen$ will be updated. In other words, SKETCH-N-SKETCH has chosen the following assignments:

$$\begin{array}{lcl}(\texttt{rim}\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'cx'} \mapsto cx, \texttt{'cy'} \mapsto cy] \\ (\texttt{rim}\ ,\ \textsc{Edge}) & \mapsto & [\texttt{'r'} \mapsto spokeLen]\end{array}$$

These are, in fact, the only choices that could have been made, because the traces for the relevant attributes were atomic locations. Dragging these zones makes it easy to adjust the location and size of the overall design.

Next, suppose we want to change the size of the passenger `cars`. The `'width'` of each rectangle is defined by a single location, $wCar$. Therefore, the assignment maps the RIGHTEDGE of every car to $wCar$:

$$(\texttt{cars}_i\ ,\ \textsc{RightEdge}) \quad \mapsto \quad [\texttt{'width'} \mapsto wCar]$$

Dragging any of these RIGHTEDGE zones allows us to easily change the `'width'` of all `cars`.

Now, suppose we want to change the number of spokes and the rotation angle. When hovering over the INTERIOR of several cars, we see that, based on the heuristics, SKETCH-N-SKETCH has chosen to vary $numSpokes$ and $rotAngle$ for several cars.

$$\begin{array}{lcl}(\texttt{cars}_0\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'x,y'} \mapsto wCar] \\ (\texttt{cars}_1\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'x,y'} \mapsto numSpokes] \\ (\texttt{cars}_2\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'x,y'} \mapsto rotAngle] \\ (\texttt{cars}_3\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'x,y'} \mapsto spokeLen] \\ (\texttt{cars}_4\ ,\ \textsc{Interior}) & \mapsto & [\texttt{'x,y'} \mapsto numSpokes]\end{array}$$

(A) Initial `ferrisWheel.little` program in black and manual code edits in `boxed blue`.

```
1  (def [cx cy spokeLen rCenter wCar rCap] [220 300 80 20 30 7])
2  (def [numSpokes rotAngle] [5 !{3-15}  0 !{-3.14-3.14}])
3
4  (def ferrisWheel
5    (let rim      [(ring 'darkgray' 6 cx cy spokeLen)]
6    (let center   [(circle 'black' cx cy rCenter)]
7    (let frame    [(nStar 'transparent' 'darkgray' 3 numSpokes spokeLen 0 rotAngle cx cy)]
8    (let spokePts (nPointsOnCircle numSpokes rotAngle cx cy spokeLen)
9    (let cars     (mapi (λ[i [x y]] (squareCenter (if (= 0 i) 'pink' 'lightgray') x y wCar)) spokePts)
10   (let hubcaps  (map (λ[x y] (circle 'black' x y rCap)) spokePts)
11     (concat [rim cars center frame hubcaps]) )))))))
12
13 (svg ferrisWheel)
```

(B) Traces for the `'x'` and `'y'` attributes of the five `'rect'` cars:

$$CAR_x(i) \doteq HUBCAP_x(i) - (wCar/2) \qquad HUBCAP_x(i) \doteq cx + spokeLen * \cos((\pi/2) - rotAngle + 2 * \pi * (i/numSpokes))$$
$$CAR_y(i) \doteq HUBCAP_y(i) - (wCar/2) \qquad HUBCAP_y(i) \doteq cy - spokeLen * \sin((\pi/2) - rotAngle + 2 * \pi * (i/numSpokes))$$

Figure 4. Ferris Wheel Example in SKETCH-N-SKETCH

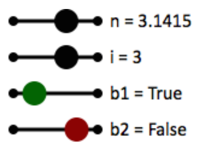
Dragging some of the `cars` has strange effects. To understand why, consider the traces of their `'x'` and `'y'` attributes, shown in Figure 4B; we have simplified the traces slightly (using constant folding) and displayed them using infix notation to improve readability. The sines and cosines that appear in the traces come from the `nPointsOnCircle` library function, which we use to position the `cars` at the end of each spoke. If we drag $\texttt{cars}_1$ or $\texttt{cars}_4$, the updated value for *numSpokes* is approximately `0.3`, which has the unintended effect of changing the number of spokes. In fact, this is an example where condition (c) of the definition of plausible updates is not satisfied; the new program does not compute an output value that is structurally equivalent to the original. If we drag $\texttt{cars}_2$, *rotAngle* is updated but the rotation is not smooth and intuitive (this kind of equation was discussed in §5.2.2). So, we use the editor's Undo feature to restore the original values of `numSpokes` and `rotAngle`.

Because we cannot easily manipulate the `numSpokes` and `rotAngle` parameters, we annotate them with ranges on line 2; these changes are depicted with blue boxes. Furthermore, we annotate them as frozen so that no direct manipulation zones (such as the INTERIOR ones for cars) change these values. Instead, we rely on the sliders to control them.

Finally, suppose we want to change the color of the first car, which will make it easier to observe how the wheel is rotated. Currently, SKETCH-N-SKETCH does not infer updates that introduce new control-flow into the program, so we edit the expression on line 9 to choose a different color for the car with index `0`. As a result of our programmatic edits, direct manipulation, and indirect manipulation via sliders, the output of our final program resembles the image at the beginning of this section. Furthermore, having identified what changes are easy to make with direct manipulation and what changes to make via sliders, we can quickly make subsequent changes to the design parameters.

6.3 Helper Value Design Pattern

The sliders provided by SKETCH-N-SKETCH, which we refer to as *user interface widgets*, are similar to the notions of *instruments* [2] and *surrogate objects* [25], both of which aim to provide GUI-based control over attributes that are not traditionally easy to directly manipulate [29]. Next, we show how to derive *custom* user interface widgets directly in `little`. Our key observation is to implement "helper" shapes whose attributes affect other parameters of interest.



User-Defined Widgets. Suppose we are unhappy with the sliders built-in to SKETCH-N-SKETCH (§2.4). We can design our own in `little`, which are used by the program below and depicted in the adjacent screenshot. One slider controls a floating-point number `n`, one controls an integer `i`, and two control booleans `b1` and `b2`.

```
(def [n  s1] (numSlider ... 0! 5! 'n = ' 3.1415^{ℓ1}))
(def [i  s2] (intSlider ... 0! 5! 'i = ' 3.1415^{ℓ2}))
(def [b1 s3] (boolSlider ... 'b1 = ' 0.25^{ℓ3}))
(def [b2 s4] (boolSlider ... 'b2 = ' 0.75^{ℓ3}))
```

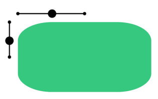
Directly manipulating the sliders *indirectly* manipulates the constants at locations ℓ_1, ℓ_2, ℓ_3, and ℓ_4 (and, hence, the values bound to `n`, `i`, `b1`, and `b2`).

Both `numSlider` and `intSlider` are defined in terms of a `slider` helper function:

```
(def slider (λ(round x0 x1 y min max s src) ...))
```

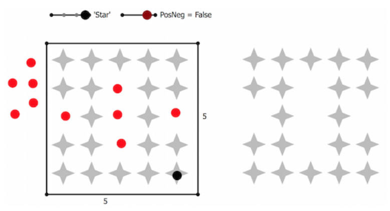
The former returns `src` clamped to the range [`min`, `max`], if necessary; the latter, furthermore, rounds `src` to the nearest integer. We refer to the number supplied as the `src` parameter to be the "source" number (or "seed") used to derive the "target" value, which is the first element of the pair returned by `slider`. The second element of the pair is the list of shapes that comprise its visuals. The idea is to place a "ball" on the line between `(x0,y)` and `(x1,y)` at a distance proportional to `(src - min) / (max - min)`. The editor provides a button for hiding shapes marked with a special `'HIDDEN'` attribute, which we add to these helper shapes. We employ the same approach to implement `boolSlider` for directly manipulating booleans. In particular, a `boolSlider` is tied to a source value between `0.0` and `1.0`, where values less than (resp. greater than) `0.5` represent `true` (resp. `false`).

Rounded Rectangles. The zones supported by Sketch-n-Sketch control only the primary attributes for each SVG shape kind (*e.g.* `'x'`, `'y'`, `'width'`, and `'height'` for rectangles). By combining user-defined sliders and the thin wrapper around the full SVG specification language, it is easy to write a `little` function that abstracts over additional parameters, such as `'rx'` and `'ry'` for specifying rounded corners, and draws sliders (scaled based on the primary attributes) next to the rectangle to control them.



```
(def roundedRect (λ(fill x y w h rx ry) ...))
```

Tile Pattern. Our last example demonstrates how custom UI widgets can control more than just individual parameters. In the screenshot below, the left (resp. right) half shows the canvas with helper shapes displayed (resp. hidden). We employ three new kinds of helper shapes in this design. First, `xySlider` is a "two-dimensional" slider that allows the control of two parameters simultaneously. In this example, we draw the `xySlider` directly atop the grid.



Dragging its handle, the black circle in the lower-right corner, around the grid provides an intuitive way to change the number of rows and columns. Next, we use `enumSlider` (drawn above the grid) to select from a list of different shapes. Then, we define red circles (to the left of the grid) to be "tokens" that denote "selection" when dragged over particular tiles in the grid. We define a helper function `isCovered` to check whether any token is currently placed over the tile centered at `(cx,cy)`. Once we are done using these helper objects to manipulate the grid, we use the built-in editor feature to toggle the visibility of helper objects, leaving us with the final design shown in the right half of the screenshot above.

Recap: Customizing the UI. Sketch-n-Sketch could provide built-in support for some of the helper objects we described (custom sliders and rounded rectangles). However, no matter how many features are built-in, we believe there will always be situations where a custom tool would be a better fit for the task at hand. With prodirect manipulation, the user can push the frontier beyond what is provided. Exploring this boundary between primitive and custom UI widgets may be fruitful, both for designing useful libraries as well as motivating new built-in features.

7. Discussion

In this paper, we presented an approach for live synchronization of a program and changes to its output, by instrumenting program evaluation to record run-time traces, phrasing user updates in terms of a new framework called trace-based program synthesis, and designing heuristics to automatically resolve ambiguities. One may think of programs in our approach as *sketches* (in the program synthesis sense [30]) where the holes are numeric constants, and the requirements for filling holes (*i.e.* changing numbers) come from the *sketches* (in the drawing sense) in the graphical user interface. Hence the name Sketch-n-Sketch.

7.1 Related Work

In a recent position paper [12], we provided a broad overview of relevant *program synthesis* (*e.g.* [23, 24, 31]), *programming by example* (*e.g.* [1, 18, 26]), and *bidirectional programming* techniques (*e.g.* [22]). Here, we focus our discussion on projects related to vector graphics.

Several projects use programming languages, direct manipulation interfaces [29], or some combination to provide expressive means for manipulating visual output. We classify them using the following interaction modes identified by Bret Victor in a talk on drawing tools [7]: "Use" for using built-in functionality through menus and buttons; "Draw" for directly manipulating domain objects; and "Code" for writing programs that manipulate domain objects.

Dynamic Drawing (Use + Draw). Victor's prototype interactive drawing editor [7], Apparatus [34], and Programming by Manipulation [21] provide expressive direct manipulation capabilities that serve as a way to build programs in restricted, domain-specific languages. By design, these tools tend to prohibit or discourage the user from manipulating content via the "indirect" mechanism of code.

Although this choice may be desirable for many application domains and end users, we believe there are limits to what can be accomplished using features and transformations provided by any tool. Therefore, our work targets users who wish to work both via direct and programmatic manipulation (*i.e.* Draw + Code).

Programs that Generate Graphics (Code). Processing [3] is a language and environment for generating visual output that has been popular both in classroom and commercial settings. Follow-on projects, such as Processing.js [28], provide similar development environments for Web programming. These systems provide immediate and interactive output, but they do not provide ways to directly manipulate output in order to modify the program that generated it.

GUIs that Generate Programs (Draw + Code). Graphical user interfaces (GUIs) for creating visual output in many domains often generate "code behind" what the user directly manipulates. Such tools include PaintCode [27], DrawScript [15], SVG-edit [33], and Adobe Fireworks for graphic design. Programs generated by these tools, however, are typically just as low-level as the output itself, making them difficult to modify, maintain, and reuse.

Constraint Programming (Draw + Code). Constraint-oriented programming systems, such as the classic SketchPad [32] and ThingLab [5] systems as well as their more recent incarnations [6, 20], are characterized by (i) declarative programming models that allow programs to specify constraints between program elements, and (ii) constraint solvers that attempt to satisfy these constraints, often using iterative and approximate numerical methods. Together with full-featured GUIs, SketchPad and ThingLab provide user experiences that tightly integrate programmatic and direct manipulation.

Our goal is to support a similar workflow but for more traditional, deterministic programming models, which are used more regularly in a variety of domains. That is, we wish to factor all constraint solving to a program synthesis phase, rather than including it within the semantics of the programming language itself.

Synthesis for Vector Graphics. The problem of *beautifying* user drawings has been long-studied in the graphics community and has recently been approached with programming-by-example techniques [9, 10, 19]. These approaches synthesize artifacts in domain-specific representations and languages.

In order to eliminate the need for secondary direct manipulation edits, Lillicon [4] synthesizes different representations for the same graphic design based on the intended edits. In SKETCH-N-SKETCH, the user must pick a particular representation. But because this representation is a general-purpose program, we can often build abstractions that are preserved by prodirect manipulation, which avoids the need for secondary edits.

7.2 Future Work

We mentioned several ways to build on our work throughout the paper, including smarter heuristics and richer trace languages that record control flow (*e.g.* [8]). We foresee several additional opportunities.

Trace-Based Program Synthesis. There are several "knobs to turn" within the framework defined in §3. The current formulation synthesizes updates given a run-time trace and a single updated value. In other settings, it may be fruitful to consider multiple traces, a history of user edits, and a history of previous program updates. Furthermore, it may be useful to rank candidate solutions according to the (soft) constraints not changed by the user.

Live Synchronization for Other Domains. We plan to retarget our approach (language instrumentation, synthesis algorithm, and prodirect manipulation editor) to meet the specific challenges of different domains, such as layout in text documents, formulas in spreadsheets, dynamic animations in presentations and data visualizations, and multiple rendering configurations for Web applications.

Prodirect Manipulation. The vision of prodirect manipulation, which we identified in a position paper [12], comprises three goals: (a) the ability to directly modify the output of a program and infer updates in real-time to match the changes (live synchronization); (b) the ability to synthesize program expressions from output created directly via the user interface; and (c) the ability to temporarily break the relationship between program and output so that "larger" changes can be made, and then reconcile these changes with the original program (called *ad hoc synchronization*).

We addressed the first goal in this paper. For the second, we plan to investigate ways to design direct manipulation operations that generate programmatic relationships. For the third goal, we plan to develop richer trace-based synthesis algorithms to infer larger, "structural" program updates, in contrast to the small, local updates we sought in this paper. These directions of future work will help fully realize the long-term vision of combining programming languages and direct manipulation interfaces.

Acknowledgments

The authors would like to thank Shan Lu, Kathryn McKinley, Loris D'Antoni, and the anonymous reviewers for valuable suggestions that improved the final version of this paper.

References

[1] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[2] Michel Beaudouin-Lafon. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Conference on Human Factors in Computing Systems (CHI)*, 2000.

[3] Ben Fry and Casey Reas. Processing. `https://processing.org/`.

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

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

[6] Alan Borning and Bert Freudenberg. ThingLab. `https://github.com/cdglabs/thinglab`.

[7] Bret Victor. Drawing Dynamic Visualizations. `http://worrydream.com/`.

[8] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic Debugging. In *International Conference on Software Engineering (ICSE)*, 2011.

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

[10] Salman Cheema, Sarah Buchanan, Sumit Gulwani, and Joseph J. LaViola, Jr. A Practical Framework for Constructing Structured Drawings. In *International Conference on Intelligent User Interfaces (IUI)*, 2014.

[11] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit Self-Adjusting Computation for Purely Functional Programs. In *International Conference on Functional Programming (ICFP)*, 2011.

[12] Ravi Chugh. Prodirect Manipulation: Bidirectional Programming for the Masses. In *International Conference on Software Engineering, Visions of 2025 and Beyond Track (ICSE V2025)*, 2016.

[13] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and Direct Manipulation, Together at Last (Supplementary Appendices). `http://arxiv.org/abs/1507.02988`, 2016.

[14] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[15] DrawScript. `http://drawscri.pt`.

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

[17] GNU Project. The gnu image manipulation program. `http://www.gimp.org/`.

[18] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

[19] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing Geometry Constructions. In *Programming Language Design and Implementation (PLDI)*, 2011.

[20] Hesam Samimi and Alex Warth. Sketchpad14. `http://www.cdglabs.org/sketchpad14`.

[21] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by Manipulation for Layout. In *Symposium on User Interface Software and Technology (UIST)*, 2014.

[22] Zhenjiang Hu, Andy Schurr, Perdita Stevens, and James F. Terwilliger. Dagstuhl Seminar on Bidirectional Transformations (BX). *SIGMOD*, 2011.

[23] Etienne Kneuss, Viktor Kuncak, Ivan Kuraj, and Philippe Suter. Synthesis Modulo Recursive Functions. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2013.

[24] Viktor Kuncak, Mikael Mayer, Ruzica Piskac, and Philippe Suter. Complete Functional Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[25] Bum Chul Kwon, Waqas Javed, Niklas Elmqvist, and Ji Soo Yi. Direct Manipulation Through Surrogate Objects. In *Conference on Human Factors in Computing Systems (CHI)*, 2011.

[26] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. User Interaction Models for Disambiguation in Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2015.

[27] PaintCode. `http://www.paintcodeapp.com`.

[28] Processing.js. `http://processingjs.org/`.

[29] Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, August 1983.

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

[31] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From Program Verification to Program Synthesis. In *Symposium on Principles of Programming Languages (POPL)*, 2010.

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

[33] SVG-edit. `https://code.google.com/p/svg-edit/`.

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

[35] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.1 (Second Edition). `http://www.w3.org/TR/SVG11/`.