Sketch-n-Sketch:
Output-Directed Programming for SVG

Brian Hempel, Justin Lubin, Ravi Chugh
Direct Manipulation is Everywhere.
Programming
Programming + Direct Manipulation?

Refactored Program
Ordinary, Text-Based Programming
+
Direct Manipulation on Output
=
Output-Directed Programming
Prior Output-Directed Programming

Schreiber et al. (2017)
Transmorphic

McDirmid (2015, 2016)
APX

Wang et al. (2012)

Kwok & Webster (2016)
Carbide Alpha

Hanna (2005)
Vital

Chugh et al. (2016)
Live Synchronization SnS

Hempel & Chugh (2016)
Sketch-n-Sketch 2016

Schuster & Flanagan (2016)

Schreiber et al. (2017)
Transmorphic

Mayer et al. (2018)
Bidirectional SnS
Prior Output-Directed Programming

Schreiber et al. (2017)
Transmorphic

McDirmid (2015, 2016)
APX

Hanna (2005)
Vital

Chugh et al. (2016)
Live Synchronization SnS

Hempel & Chugh (2016)
Sketch-n-Sketch 2016

Schuster & Flanagan (2016)

Wang et al. (2012)

Kwok & Webster (2016)
Carbide Alpha

Mayer et al. (2018)
Bidirectional SnS
Building on Sketch-n-Sketch 2016

Hempel & Chugh (UIST 2016)
Building on Sketch-n-Sketch 2016
Building on Sketch-n-Sketch 2016
Building on Sketch-n-Sketch 2016

```
def rect1
    (let [left top right bot] [107 147 290 318]
    (let bounds [left top right bot]
    (let color 371
        [ (rectangle color 'black' 0 0 bounds) ])))))

def line2
    (let [x1 y1 x2 y2] [122 157 284 315]
    (let [color width] [294 5{0-40}]
        [ (line color width x1 y1 x2 y2) ]))

def line3
    (let [x1 y1 x2 y2] [106 312 218 240]
    (let [color width] [10 5{0-40}]
        [ (line color width x1 y1 x2 y2) ]))
```
Building on Sketch-n-Sketch 2016
Building on Sketch-n-Sketch 2016
Building on Sketch-n-Sketch 2016

```lisp
(def newGroup4 (line2_color line2_width color [left top right bot])
  (def bounds [left top right bot])
  (def [rect1_right rect1_left] [right left])
  (def [rect1_bot rect1_top] [bot top])

(def rect1
  (let bounds [rect1_left rect1_top rect1_right rect1_bot]
    [ (rectangle color 'black' 0 0 bounds) ])

(def line2
  [ (line line2_color line2_width rect1_left rect1_top rect1_bot rect1_top rect1_right rect1_bot rect1_right rect1_top) ]

(def line3
  (let [x2 y2] [ (* 0.5! (+ rect1_left rect1_right)) (* 0.5! (+ rect1_bot rect1_top))]
    [ (line line2_color line2_width rect1_left rect1_top rect1_bot rect1_top rect1_right rect1_bot rect1_right rect1_top) ]

  [ (group bounds (concat [ rect1 line2 line3 ]) )]
)

(def blobs
  [with [107 147 306 344] (newGroup4 210 22 [0-40] 403) ]
)```
Building on Sketch-n-Sketch 2016

Want a rhombus:

But tools overly rigid!
What kinds of programs can be constructed *entirely* through output manipulations?
Contribution

**UI Insight**
DM on More Than Output!
- Intermediate Value
- Widgets
- Expression Focusing

**PL Insight**
Generic Tools, Too!
- Generic Provenance Tracing
- Generic Refactorings

See Paper (but not SVG-specific)
Demo
Rhombus with Veins
Widgets for Intermediate Values

Points | Offsets | Lists | Calls
---|---|---|---
\([79, 89]\) | \(x + 102\) | \([pt1, pt2, pt3]\) | rhombusFunc \([79, 89]\) 49 78

Expression Focusing + Generic Refactorings
Big Q

What kinds of programs can be constructed *entirely* through output manipulations?
Examples

(i) Koch Snowflake  (ii) Precision Floor Plan  (iii) Mondrian Arch  (iv) Balance Scale  (v) Box Volume

(vi) Xs  (vii) Battery  (viii) Ladder  (ix) Logo (via Three Tris)  (x) N Boxes  (xi) Ferris Wheel

(xii) Tree Branch  (xiii) Target  (xiv) Pencil Tip  (xv) Arrows  (xvi) Rails
WWID: PBD Benchmarks

(i) Koch Snowflake  (ii) Precision Floor Plan

(iii) Mondrian Arch  (iv) Balance Scale

(v) Box Volume  (vi) Xs

Features needed for 9 remaining tasks:

- Text boxes
- list operations
- intersections of lines with edges
- overlapping & containment constraints
- multiple constraint solving
- arbitrary if-then-else branches
Future Work

**Widget Visibility**

Soooo many!

Contextual visibility only helps a little.

**Change Explanation**

Multiple results. Necessary, but 😞

Better change descriptions?

**ODP for Novcies**

ODP is tantalizing.

But we haven’t shown it’s easy.
DM on More Than Output!
Intermediate Value Widgets
Expression Focusing

Generic Tools, Too!
Generic Refactorings (via generic tracing)

search online for “sketch n sketch”
Thank you!
Thank you!
Related Work: Non-standard Programs

Drawing with Constraints

- Toby Schachman Apparatus

Programming by Demo (PBD)

- Chasins et al. (2018) Rousillon

Parametric CAD

- Pierra et al. (1996) EBP

Constraint-Oriented Programming

- Sutherland Sketchpad
- Borning (1980) ThingLab
1. Inserts function call, assigns it to a variable.

2. Attempts to add `newVar` and `[newVar]` to the list literals in the program.

3. Succeeds when number of shapes in the output increases by the expected amount.
**MAKE EQUAL**

1. Use numeric traces (Chugh et al. PLDI '16) to set up an equation:
   \[ 114 \text{lineX1} = 245 \text{rectCX} - 80 \text{rectHalfW} \]
let a = 3 in
let b = 5 in
a + b
⇒
8
let a = 3 in
let b = 5 in
a + b
⇓
8
let $a = 3_a$ in
let $b = 5_b$ in

$a + b$

$\Downarrow$

8
let $a = 3_a$ in
let $b = 5_b$ in
$a + b$
\[\downarrow\]
$8_{a+b}$
1. Use numeric traces (Chugh et al. PLDI '16) to set up an equation:
   \[ 114 \text{lineX1} = 245 \text{rectCX} - 80 \text{rectHalfW} \]

2. Choose a constant to solve for & remove. Solve. (External solver: REDUCE).
   \[ 114 \text{lineX1} \rightarrow 245 \text{cxRect} - 80 \text{halfWRect} \]
   \[ 80 \text{halfWRect} \rightarrow 245 \text{cxRect} - 114 \text{lineX1} \]
   \[ 245 \text{cxRect} \rightarrow 114 \text{lineX1} + 80 \text{halfWRect} \]

3. If a needed constant is not bound to a variable, insert a new `let` binding at a scope visible to its usages.

4. Ranking heuristic:
   1. Smallest AST (often all the same size).
   2. Shortest distance between constants removed (measured in lines).
   3. Prefer removing constants later in the program (less like to cause a dependency inversion).
ABSTRACT

1. Interpret the selection as a late (“proximal”) set of program expressions. (Probably could be looser.)

2. Choose one of those expressions to be the return expression of the function.

3. Iteratively find let bindings that (a) have free variables and (b) are only used in the function body and add those bindings to the function body.

4. Any remaining free variables become arguments.
1. Set up an expression filter: Find \([x, y]\) pair values in provenance (execution history) of selected shapes and thereby identify relevant \(x\) expressions, \(y\) expressions, and \(point\) expressions in the program.

2. Interpret the programmer’s selections to a single expression that contains either (a) one of the above \(point\) expressions, or (b) both an \(x\) and \(y\) expression from above. Use `ABSTRACT` to make this single expression a function over a single point.

3. Map that new function over the point list.
1. Internally: Insert template code with *value holes* in place of the snaps. (A value hole is a temporary expression that contains a value.)

\[
\begin{align*}
[x, y] &= [123, 456] \\
\text{rect1} &= \text{rect} \ldots [??_{123}, ??_{456}] \ldots
\end{align*}
\]

2. Examine the provenance of the value in each to fill the hole by either:

1. Using an existing variable (from the execution environment or from the static scope, possibly moving an existing binding into scope).
2. Introducing (and using) a new variable for an existing expression.
3. Deconstructing some variable in the environment with a pattern match to expose a needed value (and using the introduced variable).

\[
[\_, y] = \text{somePoint}
\]
1. Functions that take two points, or a point and a distance, are drawable.

2. Types may be tagged with a set of *roles*, explaining the type’s semantic meaning. (E.g. “This number is a *width*. This number is a *color*.”) Called “brands” in APX. Similar to measure types, but not type-checked.

3. Roles are introduced by type aliases.
   
   ```
   type alias Color = Num
   rect :: ... → Color → ...
   ```

4. Roles propagate during the unification step of type inference.

5. Addition domain-specific rules for propagation, e.g.:
   
   ```
   a_{Num:{X}} + b_{Num:{}} \Rightarrow a_{Num:{X}} + b_{Num:{HorizontalDistance}}
   ```

6. Roles also determine the defaults for arguments.
Provenance

Canvas Selection → UI → Values → Interpret Provenance → Expressions(s)
Four Kinds of Provenance

Numeric Traces (Chugh et al. PLDI ’16)

Offsets (numbers tagged with other coordinate)

“Based On” Provenance

“Parents” Provenance
“Based On” Provenance

What expressions are associated with a value selected in the output?

For a particular value, what other values at other execution steps were used to produce it?

\[ \Gamma \vdash e \Downarrow v^e, \{v_1, \ldots, v_n\} \]
Could you hide the code? 
Fundamental limitations? 
Other Limitations? 
Will the techniques generalize? 
Future Work
Could you hide the code?

Maybe for simpler cases.

Can you represent the computation visually? (VPLs 😞)

Code only → simulate computer.
Output only → simulate code.

Consider the hover-to-preview interaction today.

(Later APX demos did hide the code)
Fundamental Limitations?

So far: “Select and Act” in small steps. Good for mouse, because that’s all a mouse can do. Generally avoided large inference steps: ambiguity. (exceptions: RELATE, REPEAT BY INDEXED MERGE)

\[ \text{bandwidth}_{\text{keyboard}} > \text{bandwidth}_{\text{mouse}} \]

...voice input?
Fundamental Limitations?

Impossible to display all intermediates.

Solution so far: *contextual visibility*.

But this is fundamental: 
#intermediates >>>>>> screen space
Other Limitations?

Not much work on breaking relationships. (Edit history?)

More details need to be worked out so tools compose reliably. (Syntactic binding locations, e.g. Xs example.)
Will the techniques generalize?

“Select & Act”

Canvas Selection → UI → Values → Interpret Provenance → Expressions(s)
Future Work

Transform DSL over value selections

Unified provenance

Visualize non-visual code