Fusing Direct Manipulations into Functional Programs

XING ZHANG, Peking University, China
RUIFENG XIE, Peking University, China
GUANCHEN GUO, Peking University, China
XIAO HE*, University of Science and Technology Beijing, China
TAO ZAN, Longyan University, China
ZHENJIANG HU, Peking University, China

Bidirectional live programming systems (BLP) enable developers to modify a program by directly manipulating the program output, so that the updated program can produce the manipulated output. One state-of-the-art approach to BLP systems is operation-based, which captures the developer's intention of program modifications by taking how the developer manipulates the output into account. The program modifications are usually hard coded for each direct manipulation in these BLP systems, which are difficult to extend. Moreover, to reflect the manipulations to the source program, these BLP systems trace the modified output to appropriate code fragments and perform corresponding code transformations. Accordingly, they require direct manipulation users be aware of the source code and how it is changed, making "direct" manipulation (on output) be "indirect".

In this paper, we resolve this problem by presenting a novel operation-based framework for bidirectional live programming, which can automatically fuse direct manipulations into the source code, thus supporting code-insensitive direct manipulations. Firstly, we design a simple but expressive delta language *DM* capable of expressing common direct manipulations for output values. Secondly, we present a fusion algorithm that propagates direct manipulations into the source functional programs and applies them to the constants whenever possible; otherwise, the algorithm embeds manipulations into the "proper positions" of programs. We prove the correctness of the fusion algorithm that the updated program executes to get the manipulated output. To demonstrate the expressiveness of *DM* and the effectiveness of our fusion algorithm, we have implemented FuseDM, a prototype SVG editor that supports GUI-based operations for direct manipulation, and successfully designed 14 benchmark examples starting from blank code using FuseDM.

Additional Key Words and Phrases: bidirectional live programming, direct manipulation, operation-based bidirectional transformation

ACM Reference Format:
Xing Zhang, Ruifeng Xie, Guanchen Guo, Xiao He, Tao Zan, and Zhenjiang Hu. 2018. Fusing Direct Manipulations into Functional Programs. *J. ACM* 37, 4, Article 111 (August 2018), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Bidirectional live programming (BLP) not only allows software developers to see continuous changes in the output instantly as they write the program, but also enables them to directly manipulate

*Corresponding author.
Authors' addresses: Xing Zhang, zhangstar@stu.pku.edu.cn, Peking University, Beijing, China; Ruifeng Xie, xieruifeng@pku.edu.cn, Peking University, Beijing, China; Guanchen Guo, guanchenguo@stu.pku.edu.cn, Peking University, Beijing, China; Xiao He, hexiao@ustb.edu.cn, University of Science and Technology Beijing, Beijing, China; Tao Zan, zan@lyun.edu.cn, Longyan University, Longyan, Fujian, China; Zhenjiang Hu, huzj@pku.edu.cn, Peking University, Beijing, China.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

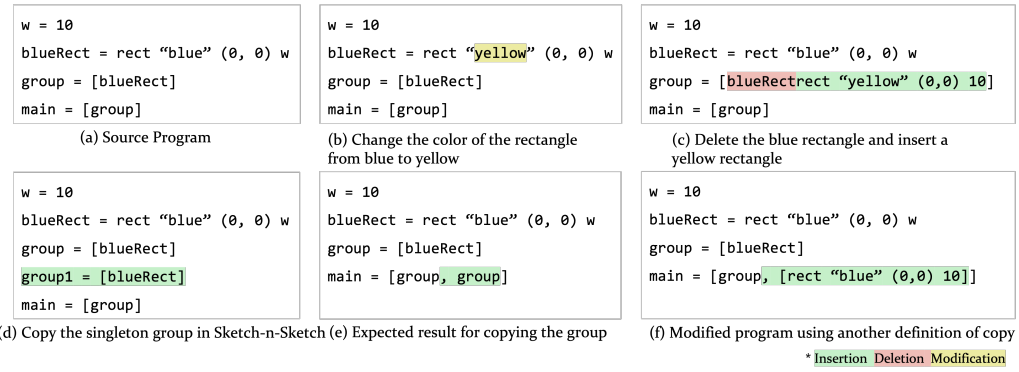


Fig. 1. An SVG Example

the output and automatically reflect the changes back to the source program, so that the updated program can produce the manipulated output.

Existing BLP systems generally fall into two categories, i.e., *state-based systems* and *operation-based systems*. *State-based systems*, such as CapStudio [Fukahori et al. 2014], Sketch-n-Sketch [Chugh et al. 2016; Mayer et al. 2018], Bidirectional Preview [Zhang and Hu 2022], and BiOOP [Zhang et al. 2023], only consider the snapshot (i.e., the state) of the manipulated output, limiting the expression of developers' intention to modify the source program. On the other hand, *operation-based systems*, such as the follow-ups of Sketch-n-Sketch[1] [Hempel and Chugh 2016; Hempel et al. 2019], take the process of manipulations into account, thus providing developers with better control over program modifications by applying different direct manipulations. To be concrete, consider the program in Figure 1a that outputs a singleton group consisting of a blue rectangle. In a state-based system, the following two manipulations—$DM1$, which changes the rectangle's color from blue to yellow, and $DM2$, which deletes the blue rectangle and then adds a yellow rectangle—lead to the identical output, resulting in the same source program as shown in Figure 1b. In contrast, in an operation-based system, $DM2$ probably transforms the program into the one in Figure 1c. This implies that the developer can obtain different updated programs through distinct manipulation operations, even though the manipulated outputs are the same.

Existing operation-based BLP systems, such as Sketch-n-Sketch, keep trace links [2] between the program output and the source code fragments and require direct manipulation users know these links. When a manipulation is performed on the output, they locate the code fragments according to the trace links and apply program transformation to modify the code. Specifically, Sketch-n-Sketch provides several manipulation operations for editing scalable vector graphics (SVG) in a visual editor. These operations perform transformations on the expressions (in the source code) associated with the selected output graphics/values. For example, *Group*ing graphics together means putting the expressions that generate the selected graphics into a list in the source code.

[1]There are different versions of Sketch-n-Sketch proposed in many papers, including both state-based and operation-based ones. For simplicity, in the rest of this paper, "Sketch-n-Sketch" refers to the operation-based versions when there is no conflict in the context.

[2]Sketch-n-Sketch[Hempel et al. 2019] mentions "...the Sketch-n-Sketch evaluator performs tracing on every execution step: the resultant value is tagged with the expression being evaluated as well as pointers to the prior (tagged) values used in the immediate computation; transitively following these prior tagged values reveals the dependencies of the computation."

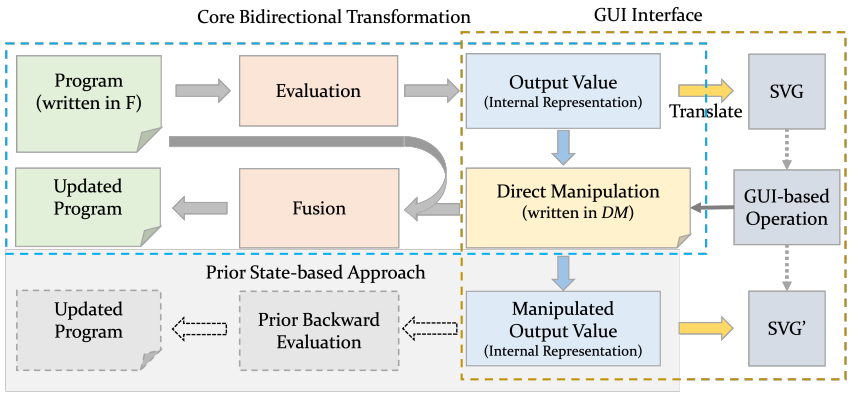


Fig. 2. Framework of FuseDM

Despite the appeal of the existing operation-based systems, there are two limitations. First, although they provide a large suite of direct manipulation operations, these operations are hard-coded and inconvenient to extend. For example, the tool developers may want to define another *Copy* operation that inserts a constant graphic identical to the copied one (like Figure 1f), rather than repeating the copied graphics as the existed *Copy* operation does. To this end, developers need to change the system by reimplementing the new *Copy* operation. Second, the direct manipulations they support are code-sensitive, in the sense that they require the developer to know the details of the program. When the developer is manipulating the output, she/he must be aware of the code fragments to be changed. Otherwise, the developer may obtain an unexpected result. Consider Figure 1a again. Assume that the developer wants to create a copy of the graphic group in the output. She/he selects a widget named "group" in the visual editor (but does not check the code it corresponds to) and invokes the manipulation operation *Copy* to duplicate the group. Sketch-n-Sketch finds the expression provenance "Line 3" in Figure 1a corresponding to the selected widget and copies the expression itself[3], resulting in Figure 1d. However, the updated program fails to get the expected output that should consist of two identical graphic groups. Figure 1e is an acceptable program that meets the developer's intention.

To address the above limitations, we shall present code-insensitive direct manipulation operations for output values and automatically fuse them into the source program to get an updated program where proper expressions are modified. In other words, we hope to propagate direct manipulation into the program and embed it into the "proper positions".

There are three main challenges to realizing our goal. First, we need to define a general code-insensitive representation of common direct manipulations, which should be expressive enough for developers to describe various modification intentions, but also simple to be fused into general source programs. Second, we should be able to locate "proper positions" of the source program to insert the direct manipulation (rather than trivially appending it to the end of the source program). Third, we should guarantee that the fusion of the direct manipulation into the source program is correct, i.e., the execution of the fused program will produce the manipulated output.

In this paper, we propose a new operation-based framework for bidirectional live programming with a key technique that can fuse code-insensitive direct manipulations into general-purpose functional programs. It supports the direct manipulation users, who know nothing about the

[3]Sketch-n-Sketch may find that the selected widget is an intermediate computation, not the final output, so only the expression is copied and not added to the output.

source program, to intentionally manipulate the output. Our main technical contributions can be summarized as follows.

- We design a simple but expressive delta language *DM* (Section 3) for manipulating output values, including data structure modification, list folding, constraint creation, etc. It can express common direct manipulations and allow tool developers to customize more.
- We present a powerful fusion algorithm (Section 4) that fuses direct manipulations into general-purpose functional programs. It can propagate direct manipulations to constants whenever possible and otherwise embed them into the "proper position" of source programs. We prove the correctness of the algorithm in that execution of the updated program yields the same manipulated output (Section 4.3). Two distinguished features are:
  - Compared to existing state-based work, it is not limited to modifying constant literals in programs, but can modify the program structure as intended by the developer;
  - Compared to existing operation-based work, it captures more direct manipulations by the delta language to avoid hard-coded implementations for each manipulation and supports modifying the program by code-insensitive direct manipulations for output values.
- We have implemented a prototype SVG editor called FuseDM to demonstrate the expressiveness of the delta language *DM* and the effectiveness of the fusion algorithm. Our tool supports various code-insensitive direct manipulations for editing SVG, which works similarly to the ones for graphics in Sketch-n-Sketch but do not require tracing the source program. Based on those direct manipulations, we have successfully worked out 14 nontrivial benchmark examples (Section 5).

2 OVERVIEW

To give an overall impression of our approach, we shall present the framework of FuseDM and demonstrate how a developer completes an SVG task through our tool with a concrete example.

2.1 Framework

The framework of FuseDM is given in Figure 2, including the core bidirectional transformation [Czarnecki et al. 2009] and the GUI interface.

In the core bidirectional transformation, in one direction, the source program written in a general-purpose functional language F (defined in Section 4.1) executes through a standard evaluator and gets the internal output representation. In another direction, a direct manipulation written in the delta language *DM* (defined in Section 3.1) alters the output to a new one and is fused into the source program through the fusion algorithm. As a contrast to our approach, the state-based approach back-propagates the modified output values back to the program (indicated by the gray blocks).

In the GUI interface, the internal output representation is translated to SVG and rendered in the browser. We realize several common GUI-based operations by using *DM*, as shown in Table 5. When a GUI-based operation is invoked, it automatically generates a piece of *DM* code and applies the *DM* code to the internal output representation to get a new output. Then, the new output is rendered as new graphics.

2.2 Illustrative Example

We use an example to illustrate how developers utilize our prototype tool, FuseDM, to develop SVG, with a focus on direct manipulations of the SVG output without having to be familiar with the source program.

Consider the task of drawing a simple butterfly using SVG. The developer may start with an initial program that generates graphics to be perfected (we also support starting from blank code).

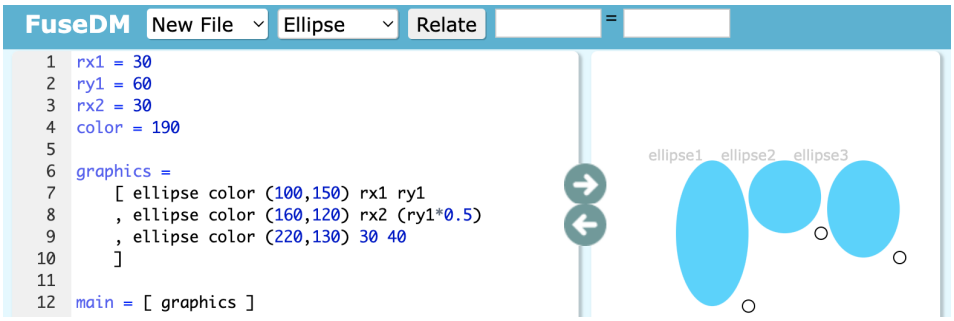
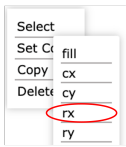


Fig. 3. A Screenshot of FuseDM with An Initial Program

(a) Relationship Editor



(b) Feature Selectors

```
ellipse1_rx = rx1
ellipse1_ry = ry1 - 5
graphics =
    [ ellipse color (100,150) rx1ellipse1_rx (ry1-5)ellipse1_ry
    , ellipse color (160,120) rx2 (ry1*0.5)
    , ellipse color (220,130) 30ellipse1_rx 40ellipse1_ry
    ]

main = [ graphics ]
```

(c) The updated program after fusing new relationships

Fig. 4. Relating Operations

Then the developer uses GUI-based operations to manipulate the graphics to the desired one step by step, while the system fuses direct manipulations into the source program so that the updated program executes again to get the exact desired butterfly.

Step 1: Starting from an Initial Program. Figure 3 shows a screenshot of our tool, which consists of an initial program on the left and a canvas on the right. Clicking on the right-facing arrow will execute the program to get an SVG output on the canvas; clicking on the left-facing arrow will fuse the direct manipulation on the output into the source program.

In the initial program, Lines 1-4 are variable declarations. Line 6 defines a graphic group (list), comprised of three ellipses. An ellipse is defined by four parameters: the color, the center coordinate, the x-axis radius, and the y-axis radius. Line 12 outputs the graphic group. On the canvas, identifiers for the graphics are depicted by light gray labels, while white circular markers facilitate resizing operations.

Step 2: Resizing Ellipses. The developer resizes the x-radius of *ellipse2* to make it smaller, which generates a direct manipulation written in *DM* as follows: It subtracts 10 from the x-radius (the 3rd argument) of the 2nd graphic of the group.

$$\texttt{modify}\ 2\ (\texttt{modify}\ 3\ (-_{\Delta}10))$$

After clicking the left-facing arrow, the assignment of the variable *rx2* in Line 3 is subtracted by 10 and becomes 20, because our fusion algorithm propagates and applies the direct manipulation to constants in the source program whenever possible.

It is worthwhile noticing that *propagating the direct manipulation to constants is not forever successful* because the fusion process produces results that can be described as inconsistent proposed

modifications of the same variable, which must be reconciled. In this case, we embed the direct manipulation into a "proper position" of the source program. For example, the developer resizes the y-radius of *ellipse1*, i.e., *ry1* in Line 7 should be changed. However, since *ellipse2* is not manipulated, *ry1* in Line 8 should not be touched. Consequently, we cannot straightforwardly modify the assignment of *ry1* in Line 1; otherwise, *ellipse2* will be affected. The fusion finds the conflict that prevents further propagation. Finally, it decomposes the delta and embeds the sub-delta, e.g., $(-_\Delta 5)$, into Line 7, replacing *ry1* with $ry1 - 5$.

Step 3: Making Two Wings the Same Size. The developer uses relating operations to make two wings (*ellipse1* and *ellipse3*) of the butterfly the same size. Relating operations relate certain properties of graphics and add a new relationship (constraint) for outputs.

Specifically, the developer clicks *ellipse3* and selects the x-radius feature as the relating goal, as shown in Figure 4b. Then "`ellipse3_rx`" is filled in the left input box of Figure 4a. Similarly, the developer fills "`ellipse1_rx`" in the right input box. Clicking the "Relate" button assigns the x-radius of *ellipse1* to the one of *ellipse3*. Note that the right input box can be edited to fill in arbitrary arithmetic expressions[4], such as "`ellipse1_rx * 2`". Then the developer sets their y-radii in the same way. As a result, *ellipse1* and *ellipse3* become the same size. The direct manipulation for setting x-radii is encoded as follows:

$$\texttt{intro}\ \textit{ellipse1\_rx}\ \texttt{by}\ (\texttt{nth}\ 0 \circ \texttt{nth}\ 2 \circ \texttt{id})\ \texttt{into}$$
$$\texttt{modify}\ 2\ \texttt{modify}\ 2\ (\texttt{repl}\ \textit{ellipse1\_rx}).$$

The direct manipulation assigns the x-radius of *ellipse1* to the one of *ellipse3*, using the constraint creation (`intro` *cx* `by` *select* `into` *dv*) defined in Section 3.1. Specifically, it introduces a new variable, *ellipse1_rx*, assigned by the x-radius of *ellipse1*. The selector $\texttt{nth}\ 0 \circ \texttt{nth}\ 2 \circ \texttt{id}$ is to extract the x-radius of *ellipse*1, i.e., the 3rd parameter of the 1st graphic. Finally, the x-radius of *ellipse3* is replaced by *ellipse1_rx*. The program fused with this direct manipulation is given in Figure 4c, where *rx1* (resp. $ry1 - 5$) is assigned to the newly-introduced variable *ellipse1_rx* (resp. *ellipse1_ry*) outside the graphic declarations; the x-radii (resp. y-radii) of *ellipse1* and *ellipse3* are set to the *ellipse1_rx* (resp. *ellipse1_ry*).

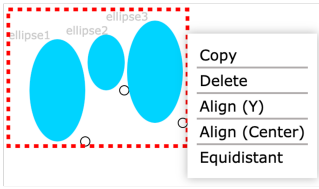
This operation shows that FuseDM automatically propagates the constraint creation into the expressions that produce the selected properties, without trace links, and updates the expressions according to the intention explicitly expressed in the written direct manipulation. In contrast, Sketch-n-Sketch realizes relating operations by tracing back the related program expressions and replacing them with synthesized expressions relying on a constraint solver, which may produce unpredictable updates.

Step 4: Aligning Body and Wings. The developer wants the centers of the body (*ellipse2*) and two wings of the butterfly (*ellipse1* and *ellipse3*) to be aligned on the y-axis. This operation (i.e., "Align (Y)") is realized by using recursive operations that manipulate the graphics of a group in a batch. In FuseDM, the developer selects the outer border of the graphic group and clicks the "Align (Y)" button, as shown in Figure 5a. The direct manipulation is coded as follows:

$$\texttt{intro}\ y_1\ \texttt{by}\ (\texttt{nth}\ 0 \circ \texttt{nth}\ 1 \circ \texttt{snd} \circ \texttt{id})\ \texttt{into}$$
$$\texttt{dfold}\ (\lambda x.(0, x+1))\ (\lambda x.\texttt{modify}\ 1\ (\texttt{id}, \texttt{repl}\ y_1)_\Delta)\ 0.$$

The direct manipulation first extracts the y-coordinate of the center of *ellipse1* and assigns it to the variable y_1. Then it employs list folding to iteratively set the y-coordinate of each ellipse's center to

[4]We will offer mouse operations to build relationships instead of manually filling in expressions in the future.

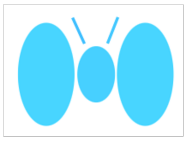


(a) Group Editor

```
y1 = 150
graphics =
    [ ellipse color (100,150y1) ellipse1_rx ellipse1_ry
    , ellipse color (160,120y1) rx2 (ry1*0.5)
    , ellipse color (220,130y1) ellipse1_rx ellipse1_ry
    ]

main = [ graphics ]
```

(b) The updated program after fusing the alignment operation



(c) Final Graphics

```
main =
    [ graphics
    , line 190 (136,210) (150,237)
    , line 190 (185,210) (174,237)
    ]
```

(d) Insertions of two lines

Fig. 5. Recursive Operations and Final Graphics

$$
\begin{array}{rcll}
\text{Deltas } dv & ::= & \texttt{id} \mid \texttt{repl}\ aexp \mid dv_1 \circ dv_2 & \text{Universal Deltas} \\
& \mid & +_\Delta\ atom \mid *_\Delta\ atom & \text{Arithmetic Deltas} \\
& \mid & (dv_1, dv_2)_\Delta & \text{Tuple Deltas} \\
& \mid & dv_1 ::_\Delta dv_2 \mid \texttt{delete}\ n \mid \texttt{insert}\ n\ atom \mid \texttt{modify}\ n\ dv & \text{List Deltas} \\
& \mid & \texttt{dfold}\ derive\ todelta\ acc & \text{List Folding} \\
& \mid & \texttt{intro}\ x\ \texttt{by}\ select\ \texttt{into}\ dv & \text{Constraint Creations}
\end{array}
$$

$$
\begin{array}{rcl}
\text{Atomic Expressions } atom & ::= & c \mid x \mid (atom_1, atom_2) \mid atom_1 :: atom_2 \\
\text{Arithmetic Expressions } aexp & ::= & atom \mid aexp_1 + aexp_2 \mid aexp_1 * aexp_2 \mid \cdots \\
\text{Selectors } select & ::= & \texttt{id} \mid proj \circ select \\
\text{Projectors } proj & ::= & \texttt{head} \mid \texttt{tail} \mid \texttt{fst} \mid \texttt{snd}
\end{array}
$$

Fig. 6. Syntax of Delta Language *DM*

y_1. The list folding `dfold` is defined in Section 3.1, which is similar to the `foldl` function in Haskell. In this case, `dfold` degenerates into the `map` function. Figure 5b displays the updated program that has been fused with this direct manipulation.

Step 5: Adding Antennae. The developer draws two lines as the butterfly's antennae, as shown in Figure 5c. Take the first line as an example. The corresponding generated direct manipulation is as follows, which denotes the insertion of a constant graphic into the 1st index:

$$\texttt{insert}\ 1\ (\texttt{line}\ 190\ (136, 210)\ (150, 237)).$$

By fusing the two direct manipulations, two constant graphics are inserted into the *main* list, as shown in Figure 5d.

3 DELTA LANGUAGE FOR DIRECT MANIPULATIONS

To achieve our goal of fusing direct manipulations into general source programs, we define a delta language (*DM*) for manipulating values, which is simple enough to be fused into programs but expressive enough to describe common direct manipulations and provide the flexibility to customize more complex ones. We give *DM*'s syntax in Section 3.1 and its semantics in Section 3.2.

3.1 Syntax of Delta Language *DM*

The syntax of *DM* is defined in Figure 6. The subscript 'Δ' is used to distinguish the notations of *DM* from those of the language F defined in Section 4. Deltas *dv* mainly include universal deltas, arithmetic deltas, tuple deltas, list deltas, and constraint creations. Specifically, universal deltas can apply to any type of value, including the identity `id`, compositions $dv_1 \circ_\Delta dv_2$, and replacements `repl` *aexp*. Arithmetic deltas include additions $+_\Delta atom$ and multiplications $*_\Delta atom$.

List deltas include delta constructors $dv_1 ::_\Delta dv_2$, deletions `delete` *n*, insertions `insert` *n atom*, modifications `modify` *n dv*, and the list folding `dfold` *derive todelta acc*, where *n* is an integer. The list folding is similar to `foldl` in Haskell. Here, *derive*—a function written in F (defined in Section 4.1)—takes the current accumulator *acc* (an atomic expression) as input and derives a new accumulator *acc*′. Additionally, it computes the arguments to be applied by the function *todelta*, which computes a delta for each list element being recursively processed.

Constraint creations `intro` *x* `by` *select* `into` *dv* introduce a new variable *x*, then extract a sub-value from the output value by the selector *select*, and then bind the variable *x* to that sub-value. The variable *x* can then be accessed in *dv*. Selectors *select* extract a specific value from a data structure, including the identity selector `id`, and composition selectors with projectors consisting of `head`, `tail`, `fst`, and `snd`. For simplicity, we use `nth` as a macro of `head` and `tail` to represent projectors for lists.

Atomic expressions *atom* include constants *c* (defined in Figure 4.1), variables *x*, and data structures (e.g., tuples and lists). The arithmetic expressions *aexp* include atomic expressions *atom*, additions $aexp_1 + aexp_2$, multiplications $aexp_1 * aexp_2$, etc.

To demonstrate that the delta language *DM* is expressive and extensible, we implement a few common direct manipulations for editing SVG output in a prototype editor, as summarized in Table 5. We have already given some examples in Section 2. Here are a few more classic examples of direct manipulation.

Example 3.1 (Copy). The copy operation is used to duplicate a graphic, producing an identical copy of the original. We assume that the modification intention implied by the copy operation is to repeat the copied graphic rather than introduce a new, identical one. Consider a canvas with only one graph, which is then copied. The copy operation is written as follows: It first extracts the existing graphic using the selector `head` ∘ `id` and assigns it to the newly introduced variable *x*. Then it inserts *x* into the 1st index behind the existing graphic.

$$\texttt{intro}\ x\ \texttt{by}\ (\texttt{head} \circ \texttt{id})\ \texttt{into}\ (\texttt{insert}\ 1\ x) \qquad \square$$

Example 3.2 (Group). We can define a group operation as putting more than one graphic into a list, making it possible to support simultaneous movement, resizing, and more recursive operations on them by utilizing the list folding. For example, grouping the two graphics together can be written as follows: It first extracts two graphics and assigns them to g_1 and g_2 respectively; then deletes them from the original list; and finally inserts a list consisting of g_1 and g_2.

$$\begin{aligned}&\texttt{intro}\ g_1\ \texttt{by}\ (\texttt{head} \circ \texttt{id})\ \texttt{into}\\ &\texttt{intro}\ g_2\ \texttt{by}\ (\texttt{tail} \circ \texttt{head} \circ \texttt{id})\ \texttt{into}\\ &\texttt{delete}\ 0 \circ \texttt{delete}\ 0 \circ \texttt{insert}\ 0\ [g_1, g_2]\end{aligned} \qquad \square$$

$$\text{[D-Id]}\ \frac{}{\texttt{id} \triangleright v \rightsquigarrow v} \qquad \text{[D-Repl]}\ \frac{\emptyset \vdash aexp \Rightarrow v'}{\texttt{repl}\ aexp \triangleright v \rightsquigarrow v'} \qquad \text{[D-Com]}\ \frac{dv_1 \triangleright v \rightsquigarrow v' \qquad dv_2 \triangleright v' \rightsquigarrow v''}{dv_2 \circ dv_1 \triangleright v \rightsquigarrow v''}$$

$$\text{[D-Add]}\ \frac{}{+_\Delta atom \triangleright n \rightsquigarrow n + atom} \qquad \text{[D-Mul]}\ \frac{}{*_\Delta atom \triangleright n \rightsquigarrow n * atom}$$

$$\text{[D-Tuple]}\ \frac{dv_1 \triangleright v_1 \rightsquigarrow v_1' \qquad dv_2 \triangleright v_2 \rightsquigarrow v_2'}{(dv_1, dv_2)_\Delta \triangleright (v_1, v_2) \rightsquigarrow (v_1', v_2')} \qquad \text{[D-Cons]}\ \frac{dv_1 \triangleright v_1 \rightsquigarrow v_1' \qquad dv_2 \triangleright v_2 \rightsquigarrow v_2'}{dv_1 ::_\Delta dv_2 \triangleright v_1 :: v_2 \rightsquigarrow v_1' :: v_2'}$$

$$\text{[D-Mod-1]}\ \frac{n > 0 \qquad \texttt{modify}\ (n-1)\ dv \triangleright v_2 \rightsquigarrow v_2'}{\texttt{modify}\ n\ dv \triangleright v_1 :: v_2 \rightsquigarrow v_1 :: v_2'} \qquad \text{[D-Mod-2]}\ \frac{dv \triangleright v_1 \rightsquigarrow v_1'}{\texttt{modify}\ 0\ dv \triangleright v_1 :: v_2 \rightsquigarrow v_1' :: v_2}$$

$$\text{[D-Fold]}\ \frac{\begin{array}{c}\emptyset \vdash derive\ acc \Rightarrow (arg, acc') \qquad todelta = \lambda x.dv \qquad dv_1 = dv[x \mapsto arg] \\ dv_1 \triangleright v_1 \rightsquigarrow v_1' \qquad \texttt{dfold}\ derive\ todelta\ acc' \triangleright v_2 \rightsquigarrow v_2'\end{array}}{\texttt{dfold}\ derive\ todelta\ acc \triangleright v_1 :: v_2 \rightsquigarrow v_1' :: v_2'}$$

$$\text{[D-Constraint]}\ \frac{v_1 = v\ |_v\ select \qquad dv[x \mapsto v_1] \triangleright v \rightsquigarrow v'}{\texttt{intro}\ x\ \texttt{by}\ select\ \texttt{into}\ dv \triangleright v \rightsquigarrow v'}$$

Fig. 7. Semantics of Delta Language *DM* (selected rules)
$dv \triangleright v \rightsquigarrow v'$

Example 3.3 (Equidistant). The equidistant operation makes the graphics of a group equally spaced on the x-axis (assume that the interval is 10), which may be written as follows: It first extracts the x-coordinate of the 1st graphic by the selector and assigns it to x_0; then it uses the list folding to replace the x-coordinate of the i-th graphic with x_0 plus $i * 10$. More detailed semantics can be found in the next section.

$$\begin{array}{l}\texttt{intro}\ x_0\ \texttt{by}\ (\texttt{head} \circ \texttt{nth}\ 1 \circ \texttt{fst} \circ \texttt{id})\ \texttt{into} \\ \texttt{dfold}\ (\lambda i.(i * 10, i + 1))\ (\lambda dis.\texttt{modify}\ 1\ (\texttt{repl}\ (x_0 + dis), \texttt{id}))\ 0\end{array}$$

□

Example 3.4 (Every N). In addition to common direct manipulations, we can also customize more problem-specific operations, such as Every N. We often have a need for a set of graphs to repeat a pattern periodically, as in the "Target" example in Figure 17. If we want a set of concentric circles in red and green, we can use the Every 2 operation written as follows: It uses list folding, where the initial index is 0; the first function *derive* returns "green" for even indexes and "red" for odd ones and increments the index; then the second function *todelta* modifies each graphic's color to the value computed by *derive*.

$$\begin{array}{l}\texttt{dfold}\ (\lambda i.\texttt{case}\ i\%2\ \texttt{of}\ \{0 \rightarrow (\text{“green”}, i + 1), 1 \rightarrow (\text{“red”}, i + 1)\}) \\ (\lambda color.\texttt{modify}\ 0\ (\texttt{repl}\ color))\ 0\end{array}$$

□

3.2 Semantics of Delta Language *DM*

The semantics of *DM* is defined in Figure 7. We discuss *DM*'s semantics here because, though rarely used in the fusion algorithm in Section 4, they define what change means, providing the target for our correctness proof later. The judgment $dv \triangleright v \rightsquigarrow v'$ states that "after applying a delta dv to the value v, v becomes v'", where the definition of values v is given in Figure 9 including constants, tuples, and lists. Below, we explain rules for each category of deltas. Notably, we compute variables by using variable substitution $[x \mapsto v]$, which replaces the variable x with a value v, instead of using a context. Additionally, the conversion of values to atomic expressions is straightforward and is therefore omitted.

Table 1. Detail Explanation of D-Fold with An Example

Step	Detail	Example
1	Apply *derive* to the seed *atom* and get a tuple $(arg, atom')$, where $atom'$ is the next seed.	$\emptyset \vdash derive\ 0 \Rightarrow (1, 1)$
2	Suppose *todelta* is $\lambda x.dv$. Substitute the variable x in dv by arg and get dv_1.	$(+_\Delta x)[x \mapsto 1] = (+_\Delta 1)$
3	Apply dv_1 to v_1 and get v_1'.	$+_\Delta 1 \triangleright 0 \rightsquigarrow 1$
4	Apply `dfold` *derive todelta atom'* to v_2 and get v_2'.	`dfold` *derive todelta* $1 \triangleright [0, 0] \rightsquigarrow [0, 1]$

Consider an operation that adds 1 to odd indexes of the list $[0, 0, 0]$. Here is a possible solution using list folding, where the initial *acc* is 0 and *todelta* is "`\x.+Δx`". *derive* may be "`\i.case i%2 of 0->(1,i+1) | 1->(0,i+1)`".

$$(v_1 :: v_2) \mid_v \texttt{head} \circ select = v_1 \mid_v select \quad \text{[S-Head]} \qquad (v_1 :: v_2) \mid_v \texttt{tail} \circ select = v_2 \mid_v select \quad \text{[S-Tail]}$$
$$(v_1, v_2) \mid_v \texttt{fst} \circ select = v_1 \mid_v select \quad \text{[S-Fst]} \qquad (v_1, v_2) \mid_v \texttt{snd} \circ select = v_2 \mid_v select \quad \text{[S-Snd]}$$
$$v \mid_v \texttt{id} = v \quad \text{[S-Id]}$$

Fig. 8. Semantics of Selectors for Values
$v \mid_v select = v'$

3.2.1 Universal Deltas. Universal deltas include three axioms. The rule D-Id states that the identity `id` keeps the value v unchanged. The rule D-Com states that, when the composition $dv_2 \circ dv_1$ applies to v, first v is applied by dv_1 and becomes v', and then v' is applied by dv_2 and becomes v''.

The rule D-Repl ignores the value v and directly replaces it with a new value, where "$\emptyset \vdash aexp \Rightarrow v'$" evaluates arithmetic expressions to values defined in Section 4. The semantics of *DM* ensures that all variables in the arithmetic expression *aexp* have been substituted with concrete numerical constants if the delta is closed (i.e., all variables are bound).

3.2.2 Arithmetic Deltas. Arithmetic deltas can apply to numeric values, including adding and multiplying a number. The rule D-Add states that, when applying $+_\Delta atom$ to a numerical value n, the value increases by *atom*. The rule D-Mul is similar in that it multiplies the value n by *atom*.

3.2.3 Tuple and List Deltas. As the names implies, tuple deltas are for tuple values and list deltas are for list values. The rule D-Tuple states that, when applying the tuple delta $(dv_1, dv_2)_\Delta$ to the tuple value (v_1, v_2), v_1 (resp. v_2) is applied by dv_1 (resp. dv_2) and becomes v_1' (resp. v_2'). The rule D-Cons is similar to D-Tuple.

The application rules of deletion, insertion, and modification all iterate through the list value and then find the position of the index to modify. We use modification as an example to describe this process. The rule D-Mod-1 states that when applying `modify` n dv to a list construction $v_1 :: v_2$ and n is larger than zero, v_2 is applied by `modify` $(n - 1)$ dv and becomes v_2'. The rule D-Mod-2 states that when n equals zero, v_1 is applied by dv and becomes v_1'.

To make our delta language *DM* simple, we use list folding to express common recursive deltas, instead of introducing generic fix points. As demonstrated by the case study in Section 5, list folding is useful enough in practice. Overall, list folding applies deltas to each element from left to right. To avoid introducing general language constructors, such as case expressions and function calls, into *DM*, we separate the recursion logic (i.e., *derive*) and the delta generation (i.e., *todelta*). As the input of *derive*, *acc* is an atomic expression denoting an accumulator, usually the list index. Let us use an example to show how the rule D-Fold works.

Example 3.5. Consider an operation that adds 1 to odd indexes of an integer list $[0, 0, 0]$. Here is a possible solution using list folding, where the initial *acc* is 0 and *todelta* is `\x.+Δx`. *derive* may be

Table 2. Detail Explanation of D-Constraint with An Example

Step	Detail	Example
1	Extract the subvalue v_1 from v by the selector *select*.	$(1,0)\ \vert_v\ (\texttt{fst} \circ \texttt{id}) = 1$
2	Substitute x in dv with v_1.	$(\texttt{id}, \texttt{repl}\ (2 * x))[x \mapsto 1] = (\texttt{id}, \texttt{repl}\ (2 * 1))$
3	Apply the substituted delta to v and get v_1'.	$(\texttt{id}, \texttt{repl}\ (2 * 1)) \triangleright (1,0) \rightsquigarrow (1,2)$

Consider an example "to make the second element of $(1,0)$ equal to twice its first element" by the constraint creation "`intro` x `by` `(fst` $\circ$ `id)` `into` `(id,repl` $(2 * x)$`)`."

```
\i.case i%2 of 0->(1,i+1) | 1->(0,i+1).
```

When applying the list folding to the list $[0,0,0]$, the rule D-Fold works as Table 1. Finally, the list changes to $[1,0,1]$. □

3.2.4 Constraint Creations. A constraint creation introduces a new variable called "x" to represent the functional relationship between sub-parts of a value, which is utilized for alignment, point joining, format brush, etc. Specifically, we use a selector (explained later) to extract one sub-value and assign it to x, and then replace another sub-value with an arithmetic expression containing x by D-Repl. As shown in Table 2, we explain the rule D-Constraint using an example "to make the second element of $(1,0)$ equal to twice its first element" by the following constraint creation

$$\texttt{intro}\ x\ \texttt{by}\ (\texttt{fst} \circ \texttt{id})\ \texttt{into}\ (\texttt{id}, \texttt{repl}\ (2 * x)).$$

Selectors are defined in Figure 8. The subscript 'v' of '$\vert_v$' denotes that the semantics of selectors is for values. (Later we define the semantics of selectors for expressions, using "$\vert_e$".) The selector `head` $\circ$ *select* (resp., `tail` $\circ$ *select*) selects the head element (resp., the remainder list except for the head element) of a list to perform subsequent *select*. The selector `fst` $\circ$ *select* (resp., `snd` $\circ$ *select*) selects the first (resp., second) element of a tuple to perform subsequent *select*. The identity `id` returns the original value itself.

4 FUSING DIRECT MANIPULATION INTO PROGRAMS

The core of our approach is to fuse code-insensitive direct manipulations into general functional programs, i.e., to propagate deltas to program constants whenever possible; otherwise, to embed deltas into the "proper positions". In this section, we first give the syntax of a core general-purpose functional language F (Section 4.1) used to write source programs. Then, we present the key fusion algorithm (Section 4.2) that fuses direct manipulations for values written in *DM* into source functional programs written in F. Lastly, we demonstrate the correctness of the fusion algorithm (Section 4.3) by showing that when fused with the direct manipulation, the modified program executes to get the manipulated output.

4.1 A Core Functional Language for Source Programs

In Figure 9, we define a core functional language F for writing source programs. Expressions include basic expressions, arithmetic expressions, case expressions, constructors, and fix points (`fix` e). Basic expressions include constants c, variables x, abstractions $\lambda p.e$, and applications $e_1\ e_2$. Constants include numbers n, booleans b, strings s, and the empty list `[]`. Constructors include tuples (e_1, e_2) and list constructions $e_1 :: e_2$. Values include constants c, tuples (v_1, v_2), list constructions $v_1 :: v_2$, and function closures $(E, \lambda p.e)$. Environments E are not standard; they are mappings from variables to pairs composed of values v and their corresponding deltas dv that are to be applied, denoted as v^{dv}.

$$
\begin{array}{rcl}
\text{Expressions } e & ::= & c \mid x \mid \lambda p.e \mid e_1\ e_2 \mid e_1 \oplus e_2 \\
& \mid & \texttt{case}\ e\ \texttt{of}\ \{p_i \rightarrow e_i\}_{i=1}^{n} \\
& \mid & (e_1, e_2) \mid e_1 :: e_2 \mid \texttt{fix}\ e \\
\text{Constants } c & ::= & n \mid b \mid s \mid [\,] \\
\text{Patterns } p & ::= & c \mid x \mid (p_1, p_2) \mid p_1 :: p_2 \\
\text{Arithmetic Operators } \oplus & ::= & (+) \mid (*) \mid (\&\&) \mid (>) \mid (==) \mid \cdots \\
\text{Values } v & ::= & c \mid (v_1, v_2) \mid v_1 :: v_2 \mid (E, \lambda p.e) \\
\text{Environments } E & ::= & \emptyset \mid E, x \mapsto v^{dv} \\
\text{(Extended) Deltas } dv & ::= & \cdots \mid (E, \lambda p.e)
\end{array}
$$

Fig. 9. Syntax of F

Deltas extend the definition in Figure 6 with additional function closures. We supplement the application rule D-Lam for function closures as follows, which replaces the original environment E and function body e with E' and e', respectively. Function changes can be decomposed into two orthogonal changes: one is the free variable change carried by E' and the other is the function body change reflected in e'.

$$(E', \lambda p.e') \triangleright (E, \lambda p.e) \rightsquigarrow (E', \lambda p.e')$$

4.2 Fusion Algorithm

We present a fusion algorithm that fuses direct manipulations written in *DM* into functional source programs written in F to get a new program. During fusion, if it can propagate the delta to the program constants, it applies the delta to update the constants. Otherwise, it embeds the delta, as a function in F, into a "proper position" of the program.

There are three primary challenges in fusion: (1) When the delta cannot be propagated into program constants, it needs to be embedded into a "proper position" in the program, rather than simply being composed with the source program. (2) The program fused with a delta executes to exactly get the manipulated output that is applied by that delta (Soundness). (3) The program should remain unchanged when there is no direct manipulation of the output (Stability), defaulted as the identity `id`.

The fusion algorithm, addressing the above challenges, is mainly presented in Figures 11, 12, and 13. The judgment $dv \triangleright E \vdash e \rightsquigarrow E' \vdash e'$ states that "fusing the delta dv into the program $E \vdash e$ yields a new program $E' \vdash e'$", where we denote programs by expression-environment pairs $E \vdash e$. We sometimes say "e (with E) is applied by dv" in reference to fusion.

Below, we explain how the fusion algorithm works in a concrete example in Section 4.2.1. Then, we give a detailed explanation of fusion rules in Sections 4.2.2 and 4.2.3. We define the "proper positions" of the source program to embed deltas in Section 4.2.4. As for soundness and stability, we discuss them in Section 4.3.

4.2.1 Fusion Example. Figure 10a presents the fusion of a tuple delta $(*_\Delta 2, \texttt{id})_\Delta$, which doubles the value of the first element and keeps the second element unchanged, into a program consisting of let-bindings that executes to get a tuple $(1, 1)$.

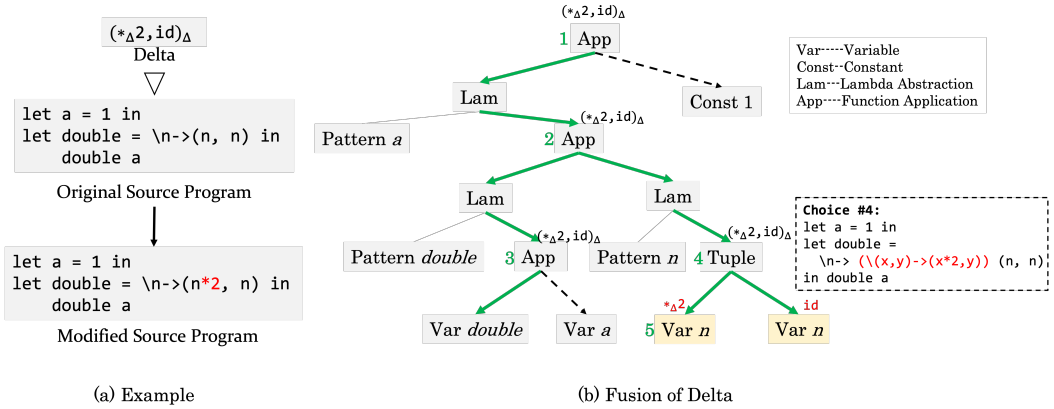


Fig. 10. Fusion Example

The fusion of the delta is a depth-first traversal of the program structure, as shown in Figure 10b (where let-bindings are represented as function applications). The fusion rules come in two varieties. As shown in Figure 10b, *propagation rules* (prefixed "P-") are the core, which carries the delta through the program structure, including three function applications, until it reaches the tuple (n, n); *application rules* (prefixed "A-") decompose the delta into two and apply each delta to each sub-expression of the tuple, i.e. applying the specific effects of deltas to expressions.

The delta, unable to be propagated to the constant 1, necessitates embedding into the program, owing to the fact that the variable n in two sub-expressions of the tuple is updated by inconsistent deltas, i.e., $*_\Delta 2$ and `id`. As shown in Figure 10b, there are five choices marked in green numbers to embed the delta. For example, embedding the delta at the fourth position generates the program beside it, with the embedded delta indicated in red.

The fusion algorithm chooses to embed the delta at the 5th position, i.e., the light yellow nodes of the variable n. (The embedding of `id` is omitted.) This is because we define the "proper position" to embed the delta as the one where an update conflict occurs; therefore, we discard the other four choices. More precisely, we prefer to embed deltas into the deepest possible substructures. For example, the fusion strives to propagate deltas to the functions or parameters being called, rather than letting them stay at the root of the function call. We define an important merge operator in the fusion algorithm to embed deltas by reconciling inconsistencies in variable bindings.

The final program is shown in Figure 10a, which executes to get $(2, 1)$ that equals the original output value $(1, 1)$ being applied by the delta $(*_\Delta 2, \texttt{id})_\Delta$. Therefore, soundness is satisfied.

4.2.2 Propagation Rules. Propagation rules are the core that carry deltas across the main program structure. The core idea is to use variable bindings (environments) to propagate deltas of variables, inspired by value propagation in Sketch-n-Sketch [Mayer et al. 2018] but adapted to delta propagation. As a reference, we also give the standard evaluation rules (prefixed "E-")

Variables. The evaluation rule E-Var states that, if the variable x is bound to v^{dv} in the environment E, the value v is applied by the delta dv and becomes v'. Accordingly, the propagation rule P-Var states that, when applying a delta dv' to a variable x, the environment E is like the original except that x is bound to a new delta $dv' \circ dv$ that composes dv' with the original delta dv. This means, in environments, only deltas are updated while values remain the same. Note that we compose the

Evaluation $E \vdash e \Rightarrow v$ | **Fusion** $dv \triangleright E \vdash e \rightsquigarrow E' \vdash e'$

$$[\text{E-Var}]\ \frac{E(x) = v^{dv} \qquad dv \triangleright v \rightsquigarrow v'}{E \vdash x \Rightarrow v'} \qquad [\text{P-Var}]\ \frac{E(x) = v^{dv}}{dv_1 \triangleright E \vdash x \rightsquigarrow E[x \mapsto v^{dv_1 \circ dv}] \vdash x}$$

$$[\text{E-Lam}]\ \frac{}{E \vdash \lambda p.e \Rightarrow (E, \lambda p.e)} \qquad [\text{P-Lam}]\ \frac{}{(E', \lambda p.e') \triangleright E \vdash \lambda p.e \rightsquigarrow E' \vdash \lambda p.e'}$$

$$[\text{E-App}]\ \frac{E \vdash e_1 \Rightarrow (E_f, \lambda x.e_f) \qquad E \vdash e_2 \Rightarrow v_2 \qquad E_f, x \mapsto v_2^{\mathrm{id}} \vdash e_f \Rightarrow v}{E \vdash e_1\ e_2 \Rightarrow v}$$

$$[\text{P-App}]\ \frac{\begin{array}{c} E \vdash e_1 \Rightarrow (E_f, \lambda x.e_f) \qquad E \vdash e_2 \Rightarrow v_2 \qquad dv \triangleright E_f, x \mapsto v_2^{\mathrm{id}} \vdash e_f \rightsquigarrow E'_f, x \mapsto v_2^{dv_2 \circ \mathrm{id}} \vdash e'_f \\ (E'_f, \lambda x.e'_f) \triangleright E \vdash e_1 \rightsquigarrow E_1 \vdash e'_1 \qquad dv_2 \triangleright E \vdash e_2 \rightsquigarrow E_2 \vdash e'_2 \qquad (E', e''_1, e''_2) = E_1{}^{e'_1} \otimes^{e'_2} E_2 \end{array}}{dv \triangleright E \vdash e_1\ e_2 \rightsquigarrow E' \vdash e''_1\ e''_2}$$

$$[\text{E-Case}]\ \frac{E \vdash e_0 \Rightarrow v_0 \qquad E_m = match\ v_0\ p_j \qquad E \cup E_m \vdash e_j \Rightarrow v_j}{E \vdash \mathsf{case}\ e_0\ \mathsf{of}\ \{p_i \to e_i\}_{i=1}^n \Rightarrow v_j}$$

$$[\text{P-Case}]\ \frac{\begin{array}{c} E \vdash e_0 \Rightarrow v_0 \qquad E_m = match\ v_0\ p_j \qquad dv \triangleright E \cup E_m \vdash e_j \rightsquigarrow E_j \cup E'_m \vdash e'_j \\ subst\ E'_m\ p_j \triangleright E \vdash e_0 \rightsquigarrow E_0 \vdash e'_0 \qquad (E', e''_0, e''_j) = E_0{}^{e'_0} \otimes^{e_j{}'} E_j \end{array}}{dv \triangleright E \vdash \mathsf{case}\ e_0\ \mathsf{of}\ \{p_i \to e_i\}_{i=1}^n \rightsquigarrow E' \vdash \mathsf{case}\ e''_0\ \mathsf{of}\ \{p_i \to e_i\}_{i=1 \wedge i \neq j}^n \cup \{p_j \to e''_j\}}$$

Fig. 11. Propagation Rules with Standard Evaluation Rules

new delta with the old one rather than directly replacing it, because we allow the propagation of composition deltas in A-Com (explained in Section 4.2.3).

Lambda Abstractions. The rule P-Lam states that, when applying the closure $(E', \lambda p.e')$, the program $E \vdash \lambda p.e$ becomes $E' \vdash \lambda p.e'$, which can be decomposed into two updates: replacing the updated function body and propagating the updated bindings of free variables to the subsequent fusion.

Function Applications. The rule E-App is standard except that the bound variable x is equipped with a default delta `id`. For simplicity, we assume only variable patterns in functions rather than arbitrary ones, as in our implementation. The intuitive idea is explained by the example "fusing $(+_\Delta 1, +_\Delta 2, +_\Delta 3)_\Delta$ into the program $(\emptyset, a \mapsto 0^{\mathrm{id}}) \vdash (\lambda x.(1, x, a))\ a$" in Table 3.

The operator $E_1{}^{e'_1} \otimes^{e'_2} E_2$ in Step 5 first merges variable bindings obtained from fusion of e_1 and e_2, and then embeds inconsistent deltas into e'_1 (resp. e'_2) to get e''_1 (resp. e''_2), formally defined in Section 4.2.4. In this example, merging bindings of the variable a in E_1 and E_2 returns two parts: one is the consistent suffix `id` to be continually propagated in the environment; the other is the inconsistent prefix deltas $+_\Delta 3$ and $+_\Delta 2$ to be embedded into e_1 and e_2, respectively. Finally, a in e_1 becomes $a + 3$ while a in e_2 becomes $a + 2$, which is exactly what we mean by the "proper position" to embed deltas. To be more accurate, we define the "proper positions" by free occurrences of variables that are updated inconsistently, which prefers the deeper sub-structure of the program, discussed in Section 4.2.4. The final program is as follows, which evaluates to the manipulated output $(2, 2, 3)$.

$$(\emptyset, a \mapsto 0^{\mathrm{id}}) \vdash (\lambda x.(2, x, a + 3))\ (a + 2).$$

Case Expressions. The fusion algorithm keeps the same execution path as the standard evaluation. Overall, the rule P-Case first applies the delta dv to the selected branch e_j, collects the delta for the scrutinee e_0, and then fuses it into e_0. The fusion algorithm guarantees that the modified program executes again with the same path, as proved in Appendix.

Table 3. Detail Explanation of P-App with An Example

	Detail	Example
1	Evaluate the values of the function e_1 and argument e_2.	By E-Lam, $(\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash (\lambda x.(1, x, a)) \Rightarrow ((\emptyset, a \mapsto 0^{\mathtt{id}}), \lambda x.(1, x, a))$ By E-Var, $(\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash a \Rightarrow 0$
2	Fuse the delta dv into the function body e_f and get e'_f and E'_f.	$(+_\Delta 1, +_\Delta 2, +_\Delta 3)_\Delta \triangleright (\emptyset, a \mapsto 0^{\mathtt{id}}, x \mapsto 0^{\mathtt{id}}) \vdash (1, x, a) \rightsquigarrow (\emptyset, a \mapsto 0^{+_\Delta 3 \circ \mathtt{id}}, x \mapsto 0^{+_\Delta 2 \circ \mathtt{id}}) \vdash (2, x, a)$, where the constant 1 is applied by $+_\Delta 1$ and becomes 2 using A-Add (explained in Section 4.2.3); the bindings of x of a are updated using P-Var.
3	Fuse the modified function closure $(E'_f, \lambda x.e')$ into the function e_1 and get e'_1 with E_1.	By P-Lam, $((\emptyset, a \mapsto 0^{+_\Delta 3 \circ \mathtt{id}}), \lambda x.(2, x, a)) \triangleright (\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash \lambda x.(1, x, a) \rightsquigarrow (\emptyset, a \mapsto 0^{+_\Delta 3 \circ \mathtt{id}}) \vdash \lambda x.(2, x, a)$
4	Fuse the delta dv_2 into the argument e_2 and get e'_2 with E_2. (If x is not updated, then dv_2 is $\mathtt{id}$, because $\mathtt{id} \circ \mathtt{id} = \mathtt{id}$.)	By P-Var, $+_\Delta 2 \triangleright (\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash a \rightsquigarrow (\emptyset, a \mapsto 0^{+_\Delta 2 \circ \mathtt{id}}) \vdash a$
5	Merge environments and solve conflicts. Refer to Section 4.2.4.	$((\emptyset, a \mapsto 0^{\mathtt{id}}), \lambda x.(2, x, a + 3), a + 2) = (\emptyset, a \mapsto 0^{+_\Delta 3 \circ \mathtt{id}})^{\lambda x.(2,x,a)} \otimes^a (\emptyset, a \mapsto 0^{+_\Delta 2 \circ \mathtt{id}})$

Consider the example "fusing $(+_\Delta 1, +_\Delta 2, +_\Delta 3)_\Delta$ into the program $(\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash (\lambda x.(1, x, a))\ a$".

$$[\text{P-Add}]\ \frac{+_\Delta n \triangleright E \vdash e_1 \rightsquigarrow E_1 \vdash e'_1 \quad (E', e''_1, e'_2) = {E_1}^{e'_1} \otimes^{e_2} E}{+_\Delta n \triangleright E \vdash e_1 + e_2 \rightsquigarrow E' \vdash e''_1 + e'_2} \qquad [\text{P-Mul}]\ \frac{E \vdash e_2 \Rightarrow n_2 \quad +_\Delta (n/n_2) \triangleright E \vdash e_1 \rightsquigarrow E_1 \vdash e'_1 \quad (E', e''_1, e'_2) = {E_1}^{e'_1} \otimes^{e_2} E}{+_\Delta n \triangleright E \vdash e_1 * e_2 \rightsquigarrow E' \vdash e''_1 * e'_2}$$

Fig. 12. Propagation Rules for Arithmetic Expressions (selected rules)

Example 4.1. Consider the fusion of a case expression as follows. The program executes to 0 and then is applied by a delta $+_\Delta 1$. The variable a in the *false* branch is applied by $+_\Delta 1$ by P-Var. Then, the scrutinee $a > 0$ is applied by the identity, which stays unchanged. Finally, the merge operator ${E_0}^{e'_0} \otimes^{e_j} {}' E_j$ embeds $+_\Delta 1$ into a in the *false* branch to get $a + 1$. See more details about the merge operator and embedding deltas in Section 4.2.4. The final program executes to 1.

$$+_\Delta\ 1 \triangleright (\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash \mathtt{case}\ a > 0\ \mathtt{of}\ \{\mathtt{true} \to 2, \mathtt{false} \to a\}$$
$$\rightsquigarrow (\emptyset, a \mapsto 0^{\mathtt{id}}) \vdash \mathtt{case}\ a > 0\ \mathtt{of}\ \{\mathtt{true} \to 2, \mathtt{false} \to a + 1\}$$

□

Arithmetic Expressions. How to apply deltas to arithmetic expressions varies in different deployments. In Figure 12, we give potential heuristic rules for additions (P-Add) and multiplications (P-Mul), both applying deltas to left operands. Other strategies include modifying the right operands or applying the same half deltas to both operands, which is not our focus.

4.2.3 Application Rules. Application rules (prefixed "A-") defined in Figure 13 decompose deltas and apply sub-deltas to sub-expressions, which is similar to the semantics of applying deltas to values defined in Figure 7. Below, we mainly focus on what is important to note when applying deltas to expressions.

Before propagation, if the delta is Id, then the rule A-Id directly returns the original program. This is more efficient than state-based systems that compare whether the new output is the same as the original one to decide whether to propagate or not. Similarly, the rule A-Repl directly replaces

$$[\text{A-Id}]\ \frac{}{\texttt{id} \triangleright E \vdash e \rightsquigarrow E \vdash e} \qquad [\text{A-Add}]\ \frac{n = n_1 + n_2}{+_\Delta n_1 \triangleright E \vdash n_2 \rightsquigarrow E \vdash n} \qquad [\text{A-Mul}]\ \frac{n = n_1 * n_2}{*_\Delta n_1 \triangleright E \vdash n_2 \rightsquigarrow E \vdash n}$$

$$[\text{A-Repl}]\ \frac{}{\texttt{repl}\ aexp \triangleright E \vdash e \rightsquigarrow E \vdash aexp} \qquad [\text{A-Com}]\ \frac{dv_1 \triangleright E \vdash e \rightsquigarrow E_1 \vdash e_1 \quad dv_2 \triangleright E_1 \vdash e_1 \rightsquigarrow E_2 \vdash e_2}{dv_2 \circ dv_1 \triangleright E \vdash e \rightsquigarrow E_2 \vdash e_2}$$

$$[\text{A-Cons}]\ \frac{dv_1 \triangleright E \vdash e_1 \rightsquigarrow E_1 \vdash e_1' \quad dv_2 \triangleright E \vdash e_2 \rightsquigarrow E_2 \vdash e_2' \quad (E', e_1'', e_2'') = E_1 {}^{e_1'}\!\otimes^{e_2'} E_2}{dv_1 :: dv_2 \triangleright E \vdash e_1 :: e_2 \rightsquigarrow E' \vdash e_1'' :: e_2''}$$

$$[\text{A-Constraint}]\ \frac{\begin{array}{c} e \mid_e select_x = e_2; e_f \quad E \vdash e_2 \Rightarrow v_2 \quad dv \triangleright E, x \mapsto v_2^{\texttt{id}} \vdash e_f \rightsquigarrow E_1, x \mapsto v_2^{dv_2 \circ \texttt{id}} \vdash e_f' \\ dv_2 \triangleright E \vdash e_2 \rightsquigarrow E_2 \vdash e_2' \qquad (E', e_f'', e_2'') = E_1 {}^{e_f'}\!\otimes^{e_2'} E_2 \end{array}}{\texttt{intro}\ x\ \texttt{by}\ select\ \texttt{into}\ dv \triangleright E \vdash e \rightsquigarrow E' \vdash (\lambda x.e_f'')\ e_2''}$$

Fig. 13. Application Rules (selected rules)

Table 4. Detail Explanation of A-Constraint with An Example

Step	Detail	Example
1	Extract the sub-expression e_2 from e by the selector *select* and replace it by x.	$(1,0) \mid_e (\texttt{fst} \circ \texttt{id})_x = 1; (x, 0)$
2	Evaluate the sub-expression e_2.	$\emptyset \vdash 1 \Rightarrow 1$
3	Apply the delta dv to the replaced expression e_f with a new environment that binds x to the pair of the value v and the identity.	$(\texttt{id}, \texttt{repl}\ (2 * x)) \triangleright \emptyset, x \mapsto 1^{\texttt{id}} \vdash (x, 0) \rightsquigarrow \emptyset, x \mapsto 1^{\texttt{id}} \vdash (x, 2 * x)$
4	Apply the delta dv_2 to the extracted sub-expression e_2.	$\texttt{id} \triangleright \emptyset \vdash 1 \rightsquigarrow \emptyset \vdash 1$
5	Merge environments and solve conflicts. Refer to Section 4.2.4.	$(\emptyset, \lambda x.(x, 2 * x)), 1) = \emptyset^{\lambda x.(x, 2*x)} \otimes^1 \emptyset$

Consider applying the delta `intro` x `by` `(fst ∘ id)` `into` `(id, repl` $(2 * x)$`)` to the source program `(1,0)`. The final program becomes $(\lambda x.(x, 2 * x))\ 1$ that outputs $(1, 2)$.

the original expression with the new one without regard for the original. Except for A-Id and A-Repl, the remaining application rules work only after propagating the delta into the deepest sub-structure of the program. This is demonstrated by rules A-Add and A-Mul, which do not take action until they have propagated to constants.

For tuple deltas and list deltas, the only difference between applying them to values versus expressions lies in the additional conflict reconciliation and inconsistent deltas embedding into sub-expressions required for the latter, as shown in the rule A-Cons.

For composition deltas, the rule A-Com states that, when applying $dv_2 \circ dv_1$ to a program $E \vdash e$, it is first applied by dv_1 and becomes $E_1 \vdash e_1$, which is later applied by dv_2 and becomes $E_2 \vdash e_2$. This is feasible because we update the variable binding by composing the later-applied delta to the first-applied delta in the rule P-Var.

Constraint creation introduces newly-formed relationships into the program by incorporating new variables. It utilizes a fresh expression, which contains the newly added variables, to articulate the functional relationship that supplants the original sub-expression. This process requires a collaborative interaction with the rule A-Repl. We explain the detailed process in Table 4 with the following example:

$$\text{[S-Head]}\ \frac{e_1 \mid_e select_x = e; e_f}{e_1 :: e_2 \mid_e (\texttt{head} \circ select)_x = e; e_f :: e_2} \qquad \text{[S-Tail]}\ \frac{e_2 \mid_e select_x = e; e_f}{e_1 :: e_2 (\texttt{tail} \circ select)_x = e; e_1 :: e_f}$$

$$\text{[S-Fst]}\ \frac{e_1 \mid_e select_x = e; e_f}{(e_1, e_2) \mid_e (\texttt{fst} \circ select)_x = e; (e_f, e_2)} \quad \text{[S-Snd]}\ \frac{e_2 \mid_e select_x = e; e_f}{(e_1, e_2) \mid_e (\texttt{snd} \circ select)_x = e; (e_1, e_f)} \quad \text{[S-Id]}\ \frac{}{e \mid_e \texttt{id}_x = e; x}$$

Fig. 14. Semantics of Selectors for Expressions
$e \mid_e select_x = e_1; e_f$

Example 4.2. Consider applying the delta `intro` x `by` (`fst` ∘ `id`) `into` (`id`, `repl` (2 ∗ x)) to the source program (`1`,`0`). The final program becomes $(\lambda x.(x, 2 * x))$ 1 that outputs $(1, 2)$.

Figure 14 shows the semantics of selectors for expressions, which are distinguished from selectors for values by the subscript 'e' of "$|_e$". Note that the newly introduced variable's name is marked as a subscript of selectors. For the output $e_1; e_f$, e_1 is the selected expression to be assigned to the introduced variable; e_f is the result of replacing the selected expression e_1 in e with the introduced variable. As shown in the last three steps in Table 4, we treat the newly introduced variable the same as the originally existing variables in P-App. Deltas generated by direct manipulations must ensure that variables within expressions to replace original ones in replacements are appropriately bound within constraints, so that the rule A-Repl directly replaces the original expression with a new expression containing newly introduced variables. In Step 5, the merge operator merges two empty environments that have absolutely no conflict, so there is no need to embed deltas.

4.2.4 "Proper Positions" to Embed Deltas. During the fusion of deltas into source programs, we mean the "proper positions" to embed deltas as ones where a variable appears in both sub-expressions and is applied by inconsistent deltas. To satisfy the overall correctness of fusion, the above-defined "proper position" is the possible deepest one of the program in the depth-first traversal path given by our algorithm. By "deepest", we mean that we propagate delta to the deepest substructures of the program whenever possible, e.g., for function calls to functions and arguments and for data structures to their substructures, as shown in the example of Figure 10, rather than stopping at the roots of expression constructors.

We define an important merge operator ${E_1}^{e_1} \otimes^{e_2} E_2$ that appears in all rules that need to handle multiple sub-expressions, to check the conditions for embedding deltas into the "proper positions". The merge operator ${E_1}^{e_1} \otimes^{e_2} E_2$ is first proposed in Sketch-n-Sketch [Mayer et al. 2018], called the *two-way merge*, which compares two variable bindings and fails as soon as it encounters inconsistency. In order to always successfully merge and give users a result, they propose a *three-way merge* $E_1 \otimes_E E_2$, which compares the two bindings with the original one and heuristically prefers the value that differs from the original. However, the three-way merge cannot guarantee correctness because one of the output updates is dropped, i.e., the modified program may not always get the manipulated output. The merge operator is optimized in our approach, which embeds inconsistent deltas into the program when there is an inconsistency in variable bindings. We call it the *optimized merge operator*. To compare the optimized merge operator with the two-way merge and the three-way merge, recall the following example given in Table 3.

Example 4.3. The variable a in sub-expression e_1 (i.e., $\lambda x.(1, x, a)$) and e_2 (i.e., a) is applied by different deltas (i.e., $+_\Delta 3 \circ$ `id` and $+_\Delta 2 \circ$ `id`). The two-way merge fails because of the inconsistency. The three-way merge may propagate either delta bound to a, which causes the other delta to be overwritten (i.e., lost). For example, if it chooses $+_\Delta 3 \circ$ `id`, the final program becomes the following

one, which evaluates to an undesired output $(2, 3, 3)$.

$$(\emptyset, a \mapsto 0^{+_\Delta 3 \circ \mathtt{id}}) \vdash (\lambda x.(1, x, a))\ a$$

In contrast, the optimized merge operator embeds the inconsistent deltas into sub-expressions as Step 5 in Table 3. □

Intuitively, the optimized merge operator first compares two structurally equivalent environments using the *comparing operator* ${}^{e_1}\times^{e_2}$ and then embeds conflicting bindings into sub-expressions using the *embedding operator* $\odot$. We give the formal definitions of the optimized merge operator below.

Definition 4.4 (Optimized Merge Operator). The optimized merge operator $E_1 {}^{e_1}\otimes^{e_2} E_2$ reconciles conflicts of variable bindings as follows. $(E^M, e_1', e_2') = E_1 {}^{e_1}\otimes^{e_2} E_2$ holds if

$$(E^M, E_1^I, E_2^I) = E_1 {}^{e_1}\times^{e_2} E_2, \qquad e_1' = E_1^I \odot e_1, \qquad \text{and } e_2' = E_2^I \odot e_2.$$

Definition 4.5 (Comparing Operator). The comparing operator ${}^{e_1}\times^{e_2}$ compares two structurally equivalent environments E_1 and E_2 and returns the successfully-merged part E^M and two inconsistent parts (E_1^I and E_2^I).

$$\begin{aligned}
&(E_1, x \mapsto v^{dv_1}) \ {}^{e_1}\times^{e_2}\ (E_2, x \mapsto v^{dv_2}) \\
&= \begin{cases}
((E^M, x \mapsto v^{dv_1}), E_1^I, E_2^I) & \text{if } dv_1 = dv_2 \\
((E^M, x \mapsto v^{dv_1}), E_1^I, E_2^I) & \text{if } x \notin fv(e_2) \\
((E^M, x \mapsto v^{dv_2}), E_1^I, E_2^I) & \text{if } x \notin fv(e_1) \\
((E^M, x \mapsto v^{dv'}), (E_1^I, x \mapsto v^{dv_1'}), (E_2^I, x \mapsto v^{dv_2'})) & \text{otherwise} \\
\qquad \text{where } (dv', dv_1', dv_2') = dv_1 \wedge_{suffix} dv_2 &
\end{cases} \\
&\text{where } (E^M, E_1^I, E_2^I) = E_1 {}^{e_1}\times^{e_2} E_2
\end{aligned}$$

If both environments bind the same delta to x, the first equation adds the delta to the successfully-merged environment E^M. Otherwise, if x does not appear free in e_2, the second equation adds dv_1 in E_1 to E^M, because updates to x do not affect e_2. (The third equation is similar.) The problematic case is when x appears free in both e_1 and e_2 but the deltas being applied are inconsistent. In the forth equation, because deltas are in the form of a composition, the longest common suffix dv' of dv_1 and dv_2 will be added to E^M and continue to propagate, while the inconsistent prefixes dv_1' and dv_2' are recorded by two complementary mappings E_1^I and E_2^I to be embedded into e_1 and e_2, respectively. The fact that the composition delta $dv_1 \circ dv_2$ can be split into the prefix and suffix by the "$\wedge_{suffix}$" operator and be fused separately is due to their satisfying the associative law by A-Com.

Definition 4.6 (Embedding Operator). The "$E^I \odot e$" embedding operator embeds deltas bound in the mapping E^I around free variables in e, denoted by

$$(E^I, x \mapsto v^{dv}) \odot e = E^I \odot e[x \mapsto \mathtt{exp}(dv)\ x],$$

where the "$[x \mapsto \mathtt{exp}(dv)\ x]$" operator replaces the free occurrences of the variable x in e with function applications $\mathtt{exp}(dv)\ x$, which apply the function transformed from dv by the helper transformation $\mathtt{exp}$ to the variable x. The transformation $\mathtt{exp}$ of deltas to expressions should satisfy the following property.

$$\text{If } E \vdash x \Rightarrow v \text{ and } dv \triangleright v \rightarrow v', \text{ then } E \vdash \mathtt{exp}(dv)\ x \Rightarrow v'.$$

$$(1)\quad \mathsf{exp}(+_{\Delta} n) = \lambda x.x + n$$

$$(2)\quad \mathsf{exp}((E, \lambda p.e)) = \lambda f.\lambda p.E \odot e$$

$$(3)\quad \mathsf{exp}(dv_1 \circ dv_2) = \lambda x.\mathsf{exp}(dv_1)\ (\mathsf{exp}(dv_2)\ x)$$

$$(4)\quad \mathsf{exp}(dv_1 :: dv_2) = \lambda x :: xs.(\mathsf{exp}(dv_1)\ x) :: (\mathsf{exp}(dv_2)\ xs)$$

$$\begin{aligned}(5)\quad &\mathsf{exp}(\mathtt{dfold}\ \mathit{derive}\ (\lambda a.dv)\ \mathit{acc}) = \\ &\quad \lambda ls.\mathtt{let}\ f = \lambda x.\lambda(\mathit{res}, \mathit{acc}). \\ &\qquad \mathtt{let}\ (\mathit{arg}, \mathit{acc}') = \mathit{derive}\ \mathit{acc}\ \mathtt{in} \\ &\qquad \mathtt{let}\ f_1 = \mathsf{exp}(dv[a \mapsto \mathit{arg}])\ \mathtt{in} \\ &\qquad\quad (\mathit{concat}\ \mathit{res}\ [f_1\ x], \mathit{acc}') \\ &\quad \mathtt{in}\ \mathit{foldl}\ f\ ([], \mathit{acc})\ \mathit{ls}\end{aligned}$$

$$(6)\quad \mathsf{exp}(\mathtt{intro}\ x\ \mathtt{by}\ \mathit{select}\ \mathtt{into}\ dv) = \lambda y.(\lambda x.\mathsf{exp}(dv)\ y)\ (\mathit{select}\ y)$$

Fig. 15. Transformations of Deltas to Expression Functions (selected rules)

$\mathsf{exp}(dv) = ef$

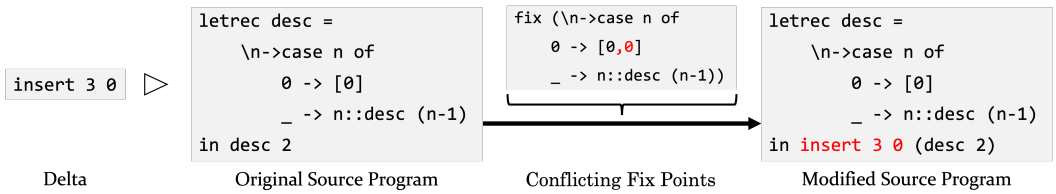


Fig. 16. Embedding Deltas into Recursion Funcion Calls

Transformations of Deltas to Expression Functions. We present a possible transformation of deltas to expression functions defined in Figure 15, which may vary in different deployments. It would be better to refactor the newly embedded functions to make the modified program more readable, but this is not our focus. We omit rules for the identity and replacements, where the identity `id` is omitted to keep original expressions unchanged and replacements `repl` directly replace original expressions with new ones. Below, we explain other details.

The first four lines of Figure 15 are straightforward. Arithmetic deltas combine with original expressions as arithmetic expressions, as shown in Step 5 of Table 3. Functions transformed from function closures replace original expressions with new functions, which are embedded with inconsistent deltas from E by the embedding operator ($\odot$). Composition deltas first apply the functions transformed from dv_2 to the original expression and then apply the function transformed from dv_1 to the previous function application. Constructor deltas transform each sub-delta into functions and apply them to sub-expressions, respectively.

We can also transform deltas into library functions in F. List folding is transformed through the library function *foldl*, defined like the one in Haskell. Other deltas, such as `insert`, `delete`, and `modify`, are transformed directly into corresponding library functions.

For constraint creations, we first transform selectors *select* into functions that decompose and extract sub-parts of original expressions. Then, we bind the sub-expression to the newly-introduced variable x when applying the function transformed from dv to the original expression.

Fix Points. Fix points require special treatment to be embedded with deltas, as it will expand all recursive calls if we directly replace fix points with new embedded functions. Therefore, we backtrack and apply the function transformed from the delta to the entry of the recursive function call. Consider the following example presented in Figure 16.

Example 4.7. The insertion delta `insert 3 0` is fused into a recursive function call, where the recursive function `desc` outputs a list of $n+1$ integers from n to 0. As the delta is propagated to the base case of `desc`, the expression `[0]` is applied by the insertion delta and becomes `[0,0]`, resulting in the conflicting definition of `desc`. Therefore, we backtrack to the entry of the recursive function call in Line 5 of the source program and embed the delta outside the function call, as marked in red in the modified program. □

Remark. When saying that the proper position is the deepest one to embed deltas, we mean the deepest substructure where delta propagation stops, specifically in the fusion path we have given above, not the optimal solution under all possible paths. Actually, the "proper position" is agnostic to the heuristic rules applied in the fusion algorithm, such as updates for arithmetic expressions and update biases for sub-expressions, which may differ across various deployments. For instance, our fusion algorithm equalizes all sub-expressions, while others might assign greater priority to certain sub-expressions, using their updated variable bindings to merge others. Therefore, some heuristic choices may embed deltas earlier, while others may continue to propagate, as shown in the following example.

Example 4.8. Consider the expression $(a, a > 0)$ with the environment $(\emptyset, a \mapsto 0^{\mathtt{id}})$, which is applied by the delta $(+_\Delta(-1), \mathtt{id})_\Delta$. If `id` always keeps the expression unchanged, then since the variable a is applied by inconsistent deltas $+_\Delta(-1) \circ \mathtt{id}$ and `id`, the delta $+_\Delta(-1)$ is embedded into the first sub-expression. The final program is $(a - 1, a > 0)$ with the unchanged environment. Alternatively, another strategy may first try applying the delta $+_\Delta(-1)$ of the first sub-expression to the second one and check if the update can succeed. Specifically, when a is decremented by 1 and becomes -1, the value of $a > 0$ is still `false` (the same as the original value). Therefore, the delta $+_\Delta(-1) \circ \mathtt{id}$ is bound to a in the environment and continues to propagate, without embedding. Note that this alternative strategy does not propagate deltas farther in any instance. □

4.3 Correctness

In this section, we discuss the correctness of our approach, i.e., the modified program generated by fusing the direct manipulation into the source program executes to produce the exact output modified by that direct manipulation.

Before defining correctness, let us recall the framework of our system presented in Figure 2. Our system contains a core bidirectional transformation [Czarnecki et al. 2009], consisting of the standard evaluation (*get* function) and the fusion (*putback* function), which maintains consistency between source programs and their outputs. The foundational work on bidirectional transformation requires that the *get* and *putback* functions satisfy the "round-tripping" properties [Foster et al. 2007]. We adopt the "round-tripping" properties to formalize the stability and soundness (correctness) of our system mentioned in Section 4.2.

The "round-tripping" properties are defined as follows. The GetPut law (stability) states that if we get an output from a program and perform no direct manipulation, the program must remain unchanged. This is straightforward because we default the direct manipulation to be the identity `id`, which keeps the same source program. The PutGet law (soundness) is more important. It states that, when a delta is fused into a program, the output of executing the modified program is equal to the output generated by applying that delta to the original output.

Table 5. Code-insensitive Direct Manipulations in FuseDM

Basic Ops	#Ex	Relating Ops	#Ex	Recursive Ops	#Ex	Mixed Ops	#Ex
Draw Graphic	14	Copy	10	⋆ Every n	1	Equidistant	4
Drag Graphic	14	Group	8	⋆Drag Group	2	⋆ Equiangular	1
Resize Graphic	14	Relate	14	⋆Resize Group	1	⋆ Equiradiidiff	1
Set Color	14					⋆ Align Centers	1
⋆Rotate	1					Align X/Y-axis	2
Delete	0						

The #Ex column indicates the number of examples in Figure 17 in which the operation is used.
Those marked with ⋆ are extended direct manipulations that are not supported by Sketch-n-Sketch.

Theorem 4.9 (GetPut). If $E \vdash e \Rightarrow v$, then $\texttt{id} \triangleright E \vdash e \longrightarrow E \vdash e$.

Theorem 4.10 (PutGet). If $E \vdash e \Rightarrow v$, $dv \triangleright v \longrightarrow v'$, and $dv \triangleright E \vdash e \longrightarrow E' \vdash e'$, then $E' \vdash e' \Rightarrow v'$.

A complete proof of the PutGet is given in Appendix. The validity of Theorem 4.11 (Merge Equivalency), defined as follows, is the key to the proof. This theorem is symmetric, which is why only one side is specified. Intuitively, it says that the value evaluated by the successfully merged environment for the expression embedded with inconsistent deltas is the same as the value evaluated by the original expression in the pre-merged environment. Specifically, it states that, when evaluated under the successfully merged environment E, the value of the expression e_1' remains equal to the value of e_1 as evaluated under E_1 before the merge. Here, the expression e_1' obtained by merging the environments E_1 and E_2, which bind free variables in e_1 and e_2 respectively, may potentially be embedded with inconsistent deltas.

Theorem 4.11 (Merge Equivalency). If $(E, e_1', e_2') = E_1 {}^{e_1}\!\otimes^{e_2} E_2$ and $E_1 \vdash e_1 \Rightarrow v$, then $E \vdash e_1' \Rightarrow v$.

5 CASE STUDY

This case study aims to demonstrate the expressiveness of our delta language, *DM*, and the effectiveness of our fusion algorithm. Firstly, we develop a prototype SVG editor, FuseDM[5], which supports common direct manipulations for editing SVG output and provides easy extensibility for several manipulations. Secondly, by implementing 14 benchmark examples from Sketch-n-Sketch using FuseDM, we demonstrate that the code-insensitive direct manipulations are effective and let users focus on manipulating outputs instead of source programs.

5.1 Expressiveness of the Delta Language *DM*

As shown in Table 5, using our delta language *DM*, we implement a few common direct manipulations for editing SVG output, which is code-insensitive. These direct manipulations, such as Drag, Group, Alignment, and Equidistant, are commonly found in various graphic editors, such as PowerPoint.

Overall, direct manipulations come into three main categories according to their implementation in *DM*. Basic operations include modifications to the properties of a single graphic, mainly expressed by arithmetic deltas. Relating operations establish relationships between graphics, such as copying and grouping, which are mainly expressed by constraint creations. Recursive operations iterate over a group of graphics, such as dragging groups, as mainly expressed by the list folding. Mixed operations are both relating and recursive operations that establish relationships for a group, such

[5]FuseDM is available at https://github.com/FuseDM/FuseDM, for which we contribute nearly 7000 lines of Elm [Czaplicki 2012] and JavaScript code. We also upload the generated code of benchmark examples to the folder "/src/Examples".

as alignment and equidistant. We list examples of how to use *DM* to implement direct manipulations in Sections 2 and 3.1.

To demonstrate the extensibility of the delta language *DM*, we also implement several extended direct manipulations that are not supported by Sketch-n-Sketch, as marked with ⋆ in Table 5. For example, an alignment operation on the y-axis is already given in Section 2, and suppose that we want to add a new operation to align circle centers. We just need to adjust the delta for alignment on the y-axis to the following one. Furthermore, arbitrary alignment can be implemented in a similar way.

$$\texttt{intro } \textit{center} \texttt{ by (nth 0} \circ \texttt{ nth 1} \circ \texttt{ id) into}$$
$$\texttt{dfold } (\lambda x.(0, x+1)) \ (\lambda x.\texttt{modify 1 (repl } \textit{center}\texttt{)}) \ 0$$

while in Sketch-n-Sketch, we may need to define a new operation with its corresponding program transformation. Another example is the "Equiangular" operation, which derives from the "Equidistant" operation. The "Equiangular" operation manipulates graphics to be distributed at equal angles around a specific point, serving as the center of the circle.

5.2 Effectiveness of Fusion

Using direct manipulations supported by FuseDM, we implement 14/16 benchmark examples from blank code, all of which are from Sketch-n-Sketch, as shown in Figure 17. By implementing these examples, we identify the strengths of FuseDM: (1) It allows developers to focus on manipulating outputs instead of source programs. (2) It guarantees correctness that the manipulated output can be obtained through executing the modified program fused with the direct manipulation.

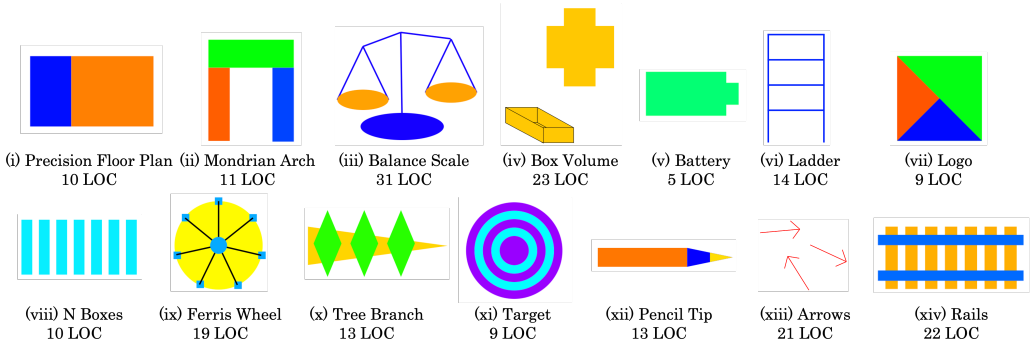
For the two unimplemented examples, one is the recursive von Koch snowflake design [von Koch 1904], which requires manipulating execution intermediates of the source program and building a recursive function. Another example is “Xs”, which are letters of different sizes but the same shape “X”, arranged in small squares. It requires precise control over creating an abstract function through “direct manipulation”, i.e., the Abstract operation that abstracts a function by extracting free variables as parameters.

Below, we discuss the general process of how we use FuseDM to implement the benchmark examples. Next, we discuss a few typical examples, comparing the sequence of operations to Sketch-n-Sketch.

5.2.1 Authoring. The #Ex column of Table 5 indicates the number of examples in Figure 17 in which the operation is used. The authoring process can be broadly classified into two categories. Instances without too many repeating graphics, such as Examples i-v, vii, xii, and xiii, mainly employ the *draw-relate* process. We first draw the initial graphics and then establish relationships between their coordinates and sizes. For example, in Example i, after drawing two rectangles, we add relationships such that the width of the left rectangle is half that of the right, and the points where the two rectangles are joined are exactly identical.

On the other hand, for the remaining instances, we mainly utilize the *group-iterate* process, where we group several graphics together and then perform recursive operations. For example, in Example xi, after grouping five circles together, we use the “Align Center” operation to make the circles concentric, and then use the “Equiradiidiff” operation to make the difference in radius between each pair of adjacent circles equal.

5.2.2 Typical Examples Compared to Sketch-n-Sketch. Below, we list two different operation sequences in FuseDM and Sketch-n-Sketch that implement the same benchmark examples. From this, we show that the graphics design using code-sensitive direct manipulations of Sketch-n-Sketch



LOC denotes the number of lines of generated source code.

Fig. 17. Benchmark Examples

can also be completed using the code-insensitive ones defined in FuseDM. To further clarify, our direct manipulations are really for the output, while theirs are for the source program.

Example 5.1. N Boxes (Example viii). In FuseDM, first we draw a blue rectangle and copy it six times. Then, we group all rectangles. Finally, we align them on the y-axis and distribute them equally spaced on the x-axis, using the Align (Y-axis) and Equidistant operations, respectively.

In Sketch-n-Sketch, we first use the "PointsBetweenSepBy" tool to create a list of equally spaced points (called *List1*) as an intermediate list in the program instead of in its final output. Then, we draw a rectangle. Finally, we use the "RepeatOnExistingList" tool to make the drawn rectangle repeat over the points in *List1*, i.e., the coordinates of the copied rectangles are set to the points in *List1*. The "RepeatOnExistingList" tool is essentially using the pre-defined list method *map* to map points of the existing list in the program to repeated rectangles. For one thing, users need to know the specific definition of "List1" in the program to understand the effect of the operation, as there may be multiple lists leading to different effects. For another, users need to have experience using *map* to more easily understand this process, as the essence of this process is manipulating the code in a different form. □

Example 5.2. Target (Example xi). In FuseDM, first we draw five circles (of the same initial color) and group them. Then, we align the centers of the circles. Afterwards, we employ the recursive operation "Every 2", enabling the color of the concentric circles to repeat in a cycle of two (in fact, we can define repetitions of any length). Finally, we use the "Equiradiusdiff" operation to make the difference of the radii of each pair of adjacent circles equal.

In Sketch-n-Sketch, we draw three concentric circles to the same center point and select them to invoke "REPEAT BY INDEXED MERGE", which maps an anonymous function that takes an index (`\i -> ...`) over the list [2,1,0]; each index is thus transformed into one of the circles, and their differences—radius and color—are turned into expression holes. Sketch-n-Sketch employs sketch-based synthesis [Solar-Lezama 2008] to resolve these holes. However, we need to select the right synthesized expressions from all the returned solutions for holes, e.g., `mod i 2 == 0` to express the alternating colors and `base + i*width` to calculate the radii of circles. After filling the holes, we get the desired target. □

To sum up, supporting code-insensitive direct manipulations is the main difference between our tool FuseDM and Sketch-n-Sketch. In Sketch-n-Sketch, developers are essentially manipulating the source program. What is more, some operations create intermediate data in the program that

fails to effectively reflect in the output, thus being unable to guarantee correctness and causing confusion for users. Besides, they use sketch synthesis to replace more customized operations, which is a good way to facilitate extensions but also requires users to be aware of the program.

5.3 Limitations

All well-defined direct manipulations expressed by the delta language (DM) can be brought to source programs by fusion. Nevertheless, the current delta language only encodes concrete direct manipulations on values and lacks the abstraction ability to parameterize direct manipulations and abstract graphics into functions. For example, the N Boxes example is implemented by drawing seven boxes repeatedly, where the number 7 is fixed and cannot be parameterized to an arbitrary number. The reason for this is that we avoid introducing lambda abstractions into the delta language to simplify the fusion of deltas into programs. Furthermore, the current delta language employs the list folding to implement recursion, whose recursive form is restrictive. This approach may not be user-friendly for developers to encode more general recursive operations. To address these limitations, we will incorporate lambda abstractions and general recursions into the delta language in the future.

6 RELATED WORK

Our work on fusing direct manipulations into general functional programs is closely related to bidirectional live programming, delta-based bidirectional transformations, and functional expression fusion.

6.1 Bidirectional Live Programming

Bidirectional live programming (BLP) tightly couples the intuitiveness of direct manipulation of outputs with the abstractness and repeatability of text-based programming. There are two main types of BLP systems: *state-based systems* and *operation-based systems*.

6.1.1 State-based BLP Systems. Most existing BLP systems are state-based, i.e., they only consider the state of the manipulated output. Specifically, they can be subdivided into those that use tracking and those that use bidirectional evaluation.

The following two works both modify the program by tracking the execution of programs. Capstudio [Fukahori et al. 2014] traces the program execution history, i.e., rendering function call log, associated with the manipulated output. When developers manipulate the output, Capstudio adds or subtracts a number from the related arguments of the function call to equal the manipulated output. Sketch-n-Sketch [Chugh et al. 2016] presents trace-based program synthesis, which tracks the provenances of the constants in the program and establishes value-trace equations with the manipulated output to be solved by a constraint solver. Unlike our work that can modify the program structure to some extent, they only modify the function application arguments or program constants, limiting the expression of developers' modification intentions.

Sketch-n-Sketch [Mayer et al. 2018], Bidirectional Preview [Zhang and Hu 2022], and BiOOP [Zhang et al. 2023] extend the standard evaluation with a backward update evaluation that propagates the updated output value to program constants by backtracking the forward evaluation. Our fusion algorithm is inspired by backward update evaluation, but there are three significant differences. (1) We take an operation-based approach that propagates deltas, whereas they take a state-based approach that propagates values. (2) Unlike our approach that can modify program structures by embedding direct manipulations, they only modify constant literals in programs. (3) We adopt the optimized merge of variable bindings, so that the updated program, which is fused

with a direct manipulation, executes to produce the exact manipulated output, whereas they cannot due to using the three-way merge.

6.1.2 Operation-based BLP Systems. Operation-based BLP systems take the process of manipulation into account, such as the two follow-ups of Sketch-n-Sketch [Hempel and Chugh 2016; Hempel et al. 2019]. The former follow-up provides GUI-based operations for drawing shapes, relating shapes to each other, grouping shapes together, and creating reusable abstractions. It realizes them by tracing links between code fragments and output and performing program transformation on the code fragments linked to the manipulated sub-values. Sketch-n-Sketch [Hempel et al. 2019] exposes intermediate execution products for manipulation and presents a mechanism for contextual drawing, requiring knowledge about source programs.

There are three significant differences between our work and the operation-based versions of Sketch-n-Sketch. (1) Our direct manipulations can express developers' intentions for modifying outputs and source programs more clearly. This is because we implement direct manipulations by using the delta language *DM*, which have determined modification acts as defined by their semantics. In Sketch-n-Sketch, on the other hand, the modification behavior of some operations is ambiguous. They use sketch-based synthesis to heuristically give developers some possible modifications for the program to choose from. (2) We support easier customization for direct manipulations using the delta language *DM*, only focusing on outputs, while customization in Sketch-n-Sketch needs to add its corresponding program transformation. (3) We support code-insensitive direct manipulations for output values. Therefore, direct manipulation users only need to focus on the output without being aware of the source program.

6.2 Delta-based Bidirectional Transformations

Bidirectional transformations (BXs) [Czarnecki et al. 2009] are a mechanism for maintaining the consistency of two (or more) related sources of information, which originates the *view update problem* in databases. Traditional algebraic frameworks for bidirectional transformations are state-based: the input and output are states of data. Whereas in delta-based bidirectional transformations, the synchronizer tries to understand what the delta is that resulted from the update and then tries to propagate the delta [Diskin et al. 2010]. In this sense, our work is a delta-based bidirectional transformation.

There have been many studies of delta-based bidirectional transformations. Bancilhon and Spyratos [1981] propose a method for defining translators that translate updates on views into updates on the underlying database. Melnik et al. [2008] propose a technique for propagating view updates incrementally in a transformation language using view maintenance. Diskin et al. [2011] build delta-based generalizations of known state-based frameworks for symmetric model transformations. Edit lenses [Hofmann et al. 2012] offer a general theory that works with descriptions of changes to data structures rather than with the structures themselves.

Our fusion algorithm can be seen as a translator that transforms the output updates expressed in the delta language *DM* into program updates. While existing work has focused on relational views, model transformations, and data structures, we are the first to study delta-based bidirectional transformations on general-purpose functional programs and their outputs.

6.3 Functional Expression Fusion

Many studies on the fusion of functional expressions, such as Takano and Meijer [1995], Ohori and Sasano [2007], and Ureche et al. [2015], remove unnecessary intermediate data structures to improve the efficiency of program execution. Below, we discuss two of the most classic efforts.

Deforestation [Wadler 1988] presents an algorithm that transforms any term composed of functions into treeless form to eliminate intermediate lists and trees. However, the technique is only applicable to a subset of first-order expressions. Chin [1992] presents the *safe fusion* that extends deforestation to all first-order programs through the adoption of the producer-consumer view of functions. Besides, with the help of high-order removal transformation, they extend deforestation to all well-typed high-order programs. We borrow the concept of "expression fusion" to describe the insertion of direct manipulations into general-purpose functional programs. The difference, however, is that in the context of bidirectional live programming, our approach does not eliminate intermediate data but simply applies direct manipulations to program constants or embeds them directly into the program as functions.

7 CONCLUSION AND FUTURE WORK

This paper presents a new operation-based framework for bidirectional live programming that fuses code-insensitive direct manipulations into source functional programs to get the manipulated output. Specifically, we design a simple but expressive delta language, *DM*, for manipulating output values that can express many commonly used direct manipulations in visual SVG editors. Then, we present the key fusion algorithm that propagates direct manipulations into program constants whenever possible; otherwise, the algorithm embeds them into the "proper positions" where variables are updated inconsistently. We guarantee that our fusion algorithm satisfies correctness, i.e., that the modified program that has been fused with a direct manipulation executes to get the exact manipulated output. To demonstrate the expressiveness of our delta language *DM* and the effectiveness of our fusion algorithm, we implement a prototype SVG editor, FuseDM, to support code-insensitive direct manipulations for editing SVG output. Using FuseDM, we successfully implemented 14 benchmark examples from Sketch-n-Sketch.

We now outline two future works. First, the delta language may be extended. To support parameterizing direct manipulations and abstracting graphics into functions, lambda abstractions need to be introduced into the delta language. Second, FuseDM's usability need to be enhanced. There are two further improvements. One is the post-processing of refactoring and reformatting the code after embedding deltas into the source program, because the present syntax-based transformation from deltas to expressions may not be easy to read. Another improvement involves integrating a constraint (relationship) solver into the visual editor that would automatically assist developers in maintaining the relationships between all graphics. This is essential, as it can be challenging for developers to manually track and maintain all the implicit relationships between graphics. In that case, for example, if the developer adds a relationship such that the heights of two rectangles are the same and then resizes the height of one rectangle, the other rectangle should resize to the same value automatically.

ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for many helpful suggestions. This work was partly supported by the National Key Research and Development Program of China (No. 2021ZD0110202) and the Natural Science Foundation of Fujian Province for Youths (No. 2021J05230).

REFERENCES

F. Bancilhon and N. Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (dec 1981), 557–575. https://doi.org/10.1145/319628.319634

Wei-Ngan Chin. 1992. Safe Fusion of Functional Expressions. *SIGPLAN Lisp Pointers* V, 1 (jan 1992), 11–20. https://doi.org/10.1145/141478.141494

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Jun 2016). https://doi.org/10.1145/2908080.2908103

Evan Czaplicki. 2012. Elm: A delightful language for reliable webapps. https://elm-lang.org/ Accessed: 2023-07-04.

Krzysztof Czarnecki, Nate Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective, Vol. 5563. 260–283. https://doi.org/10.1007/978-3-642-02408-5_19

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2010. From State- to Delta-Based Bidirectional Model Transformations. In *Proceedings of the Third International Conference on Theory and Practice of Model Transformations* (Málaga, Spain) *(ICMT'10)*. Springer-Verlag, Berlin, Heidelberg, 61–76.

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems* (Wellington, New Zealand) *(MODELS'11)*. Springer-Verlag, Berlin, Heidelberg, 304–318.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. 29, 3 (may 2007), 17–es. https://doi.org/10.1145/1232420.1232424

Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. 2014. CapStudio: An Interactive Screencast for Visual Application Development. *Conference on Human Factors in Computing Systems - Proceedings*. https://doi.org/10.1145/2559206.2581138

Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2984511.2984575

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 495–508. https://doi.org/10.1145/2103656.2103715

Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (oct 2018), 28 pages. https://doi.org/10.1145/3276497

Sergey Melnik, Atul Adya, and Philip A. Bernstein. 2008. Compiling Mappings to Bridge Applications and Databases. *ACM Trans. Database Syst.* 33, 4, Article 22 (dec 2008), 50 pages. https://doi.org/10.1145/1412331.1412334

Atsushi Ohori and Isao Sasano. 2007. Lightweight Fusion by Fixed Point Promotion. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1190216.1190241

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 306–313. https://doi.org/10.1145/224164.224221

Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. 2015. Automating Ad Hoc Data Representation Transformations. *SIGPLAN Not.* 50, 10 (oct 2015), 801–820. https://doi.org/10.1145/2858965.2814271

H. von Koch. 1904. *Sur une courbe continue sans tangente obtenue par une construction geometrique elementaire.* Norstedt & soner. https://books.google.com.hk/books?id=kf3NnQAACAAJ

Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (jan 1988), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Xing Zhang, Guanchen Guo, Xiao He, and Zhenjiang Hu. 2023. Bidirectional Object-Oriented Programming: Towards Programmatic and Direct Manipulation of Objects. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 83 (apr 2023), 26 pages. https://doi.org/10.1145/3586035

Xing Zhang and Zhenjiang Hu. 2022. Towards Bidirectional Live Programming for Incomplete Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2154–2164. https://doi.org/10.1145/3510003.3510195