Bidirectional Object-Oriented Programming: Towards Programmatic and Direct Manipulation of Objects

XING ZHANG, Peking University, China
GUANCHEN GUO, Peking University, China
XIAO HE*, University of Science and Technology Beijing, China
ZHENJIANG HU, Peking University, China

Many bidirectional programming languages, which are mainly functional and relational, have been designed to support writing programs that run in both forward and backward directions. Nevertheless, there is little study on the bidirectionalization of object-oriented languages that are more popular in practice. This paper presents the first bidirectional object-oriented language that supports programmatic and direct manipulation of objects. Specifically, we carefully extend a core object-oriented language, which has a standard forward evaluation semantics, with backward updating semantics for class inheritance hierarchies and references. We formally prove that the bidirectional evaluation semantics satisfies the round-tripping properties if the output is altered consistently. To validate the utility of our approach, we have developed a tool called BiOOP for generating HTML documents through bidirectional GUI design. We evaluate the expressiveness and effectiveness of BiOOP for HTML webpage development by reproducing ten classic object-oriented applications from a Java Swing tutorial and one large project from GitHub. The experimental results show the response time of direct manipulation programming on object-oriented programs that produce HTML webpages is acceptable for developers.

CCS Concepts: • **Software and its engineering** → *Domain specific languages.*

Additional Key Words and Phrases: Bidirectional Transformation, Direct Manipulation, Object-Oriented Programming, Language Design and Implementation

ACM Reference Format:
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 (April 2023), 26 pages. https://doi.org/10.1145/3586035

1 INTRODUCTION

In the programming language community, there has been a lot of work on designing and implementing bidirectional programming languages. In bidirectional programming, not only can the output be obtained through a forward evaluation function *get* that is applied to the input, but also the updated input can be computed through a backward evaluation function *putback* that is applied to the original input and the altered output. For example, LittleLeo [Mayer et al. 2018], *Bi-X* [Nakano et al. 2008], and *formlenses* [Rajkumar et al. 2014] are designed for website development; *lens combinators* [Foster et al. 2007] and *X* [Hu et al. 2004] for tree-like data; `little` [Chugh et al. 2016] for SVG graphics design; and CapStudio [Fukahori et al. 2014] for game application design.

Authors' addresses: Xing Zhang, zhangstar@stu.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; Zhenjiang Hu, huzj@pku.edu.cn, Peking University, Beijing, China.

2475-1421/2023/4-ART83
https://doi.org/10.1145/3586035

```
1  class MenuItem extends AbstractButton {
2      private String title;
3      public void setTitle(String t) {
4          this.title = t + "...";
5      }
6      ...
7  }
8  class Menu extends MenuItem {
9      private List<MenuItem> items;
10     ...
11 }
12 class MenuBar extends Component {
13     private List<Menu> menus;
14     ...
15 }
16 public class MenuDemo {
17     public static void main(String[] args){
18         fileMenu = new Menu();
19         fileMenu.setTitle("File(F)");
20         ...
21         color = white;
22         fileMenu.setColor(color);
23         ...
24         color = gray;
25         excelMenuItem.setColor(color);
26         ...
27     }
28 }
```

Fig. 1. A Java-like Program


```
1  <div class="dropdown">
2      <button class="dropbtn">File(F)...</button>
3      <div class="dropdown-content">
4          <a>New File</a>
5              ...
6          <a>Open</a>
7              ...
8          <a>Save</a>
9              ...
10         <a>Exit</a>
11             ...
12     </div>
13 </div>
14 <div class="dropdown">
15     <button class="dropbtn">Edit(E)</button>
16     ...
17 </div>
18 <div class="dropdown">
19     <button class="dropbtn">View(V)</button>
20     ...
21 </div>
22 <div class="dropdown">
23     <button class="dropbtn">Help(H)</button>
24     ...
25 </div>
```

Fig. 2. An HTML Menu Bar

Despite many bidirectional languages, functional or relational, as far as we know, there is no study on bidirectional *object-oriented* languages. Object-oriented programming languages (OOPLs) are more complex than functional languages because they not only describe how the output is computed from the input, but also need to keep a good program structure for maintainability and reusability through class inheritance hierarchy, mutable data, and object management. This program structure poses big challenges in designing bidirectional semantics for OOPLs because, in the backward evaluation, we have to carefully design a putback strategy that can propagate the updates on the output back to the source program that contains the class inheritance hierarchy and dynamic object structure. To see the challenges concretely, let us consider a Java-like program given in Figure 1 that outputs a menu bar in HTML, as shown in Figure 2.

The first challenge is how to *handle ambiguity in class hierarchy updating*. In Figure 1, the class Menu extends the class MenuItem in Line 8, and *fileMenu* is a Menu object that invokes the method *setTitle* inherited from MenuItem in Line 19, which gives "File(F)..." in the Menu Bar in Figure 2. Now, supposing that we want to modify the Menu Bar by changing "File(F)..." to "File(F) >", we can see that the class hierarchy in the source program can be updated in several ways to accommodate this modification to the output: (1) changing "..." to " >" in *setTitle* defined in MenuItem (Line 4), (2) adding an overriding method *setTitle* to Menu leaving MenuItem unchanged, or (3) adding a subclass of Menu to override *setTitle*. When the backward updating contains numerous modifications to the class hierarchy, the number of updating solutions will increase exponentially. It is essential to reduce the updating ambiguity under the premise that the class hierarchy is correctly updated.

The second challenge is how to *handle references*. Most OOPLs are not pure and have references [Pierce 2002]. In Figure 1, *color* is a reference assigned `white` in Line 21, and *fileMenu* is set to *color*. Then, *color* is assigned `gray` in Line 24, and *excelMenuItem* is set to *color*. Now we change the color of the "Excel" button from `gray` to `brown` by editing the HTML code in Figure 2. When propagating this change back, we need to pay attention to the update order of the two assignments (in Lines 21 and 24) and roll back the value pointed by *color* in the first updating before the subsequent one. It means that the assignment in Line 24 should be updated to `brown` first, and the assignment in Line 21 should stay `white`.

The third challenge is how to *update the object structure by directly manipulating outputs*. Execution of an object-oriented program usually produces a complex graph of interrelated dynamic

objects, where the links are the references between objects. For example, the Java-like program in Figure 1 generates some objects that form an object tree rooted at a MenuBar object (not shown). Nevertheless, the structural information of the object graph may get lost if we perform a simple conversion from the MenuBar object to HTML, because we cannot know the correspondences between the HTML fragments and the objects of the running program. We need to find a natural way to associate the output (e.g., HTML code) with the object graph so that one can manipulate the dynamic objects by altering the output.

In this paper, we tackle these challenges and propose the first bidirectional object-oriented language, focusing on GUI design scenarios. We implement a prototype programming tool for developing HTML webpages, which can support programmatic and direct manipulation of objects. It enables developers to directly manipulate the output, including objects, and automatically synchronize the manipulated output with the object-oriented program. Our main technical contributions can be summarized as follows.

- We propose a novel bidirectional object-oriented language called BiFJ (Section 3) by extending the Featherweight Java [Pierce 2002] with references and giving a new backward evaluation semantics that can propagate the updates on the output to those on the source program. We prove that our bidirectional semantics satisfies the round-tripping properties [Foster et al. 2007] when the output is changed consistently (Section 3.4). Particularly,
  - we present a *Class Structure Refactoring* algorithm for dealing with the ambiguity of back-updating the class hierarchy of an object-oriented program while ensuring that round-tripping properties are not broken (Section 3.2);
  - we give an efficient realization of side-effects (i.e., references) in bidirectional computation. Unlike functional environments, stores can be handled more efficiently without merging and only require an effect rollback in the backward evaluation (Section 3.3).
- We implement a prototype programming tool called BiOOP for developing HTML webpages, which is available online[1] (Section 4). In addition to an efficient implementation of the bidirectional semantics of the language, we propose the idea of "two-stage bidirectionalization" to deal with runtime object structures: we have a bidirectional transformation between the BiFJ source program and the object structure, and another bidirectional transformation between the object structure and the actual representation (e.g., HTML).
- We demonstrate the usefulness of our language and tool (Section 5) by successfully bidirectionalizing all (ten) classic object-oriented applications from a Java Swing tutorial[2] and one large project from GitHub. We conduct experiments with various direct manipulation interactions on the above examples and obtain the execution time for each example. The experiment results show that our system is scalable and promising to develop larger applications in the future.

2 OVERVIEW

To get a flavor of bidirectional object-oriented programming, we shall demonstrate with a concrete example how developers accomplish a GUI development task through bidirectional object-oriented programming, i.e., forward programmatic manipulation and backward direct manipulation of the generated GUI in HTML.

Consider the task of implementing an HTML menu bar for an editor, like the example in Section 1. Using our tool BiOOP, the developer may start with an initial object-oriented program that

[1]Available at https://github.com/xingzhang-pku/BiOOP. The execution result is saved in the source code, and our tool can be used by opening the oop-index.html file in the browser.
[2]Available at https://www.javatpoint.com/java-swing.

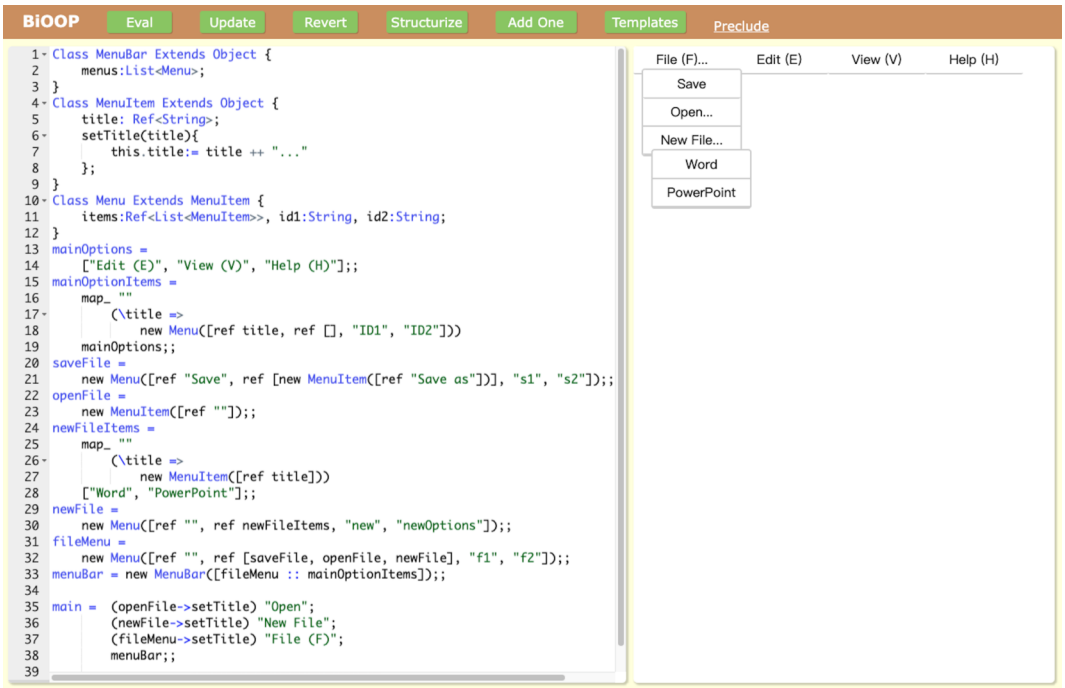

Fig. 3. An Initial Object-Oriented Program that Generates an HTML Menu Bar

generates a prototype GUI to be perfected. Then the developer directly manipulates the prototype GUI to the desired form, while propagating the updates on the GUI to the program so that the program's execution produces the desired GUI.

2.1 Initial Object-Oriented Program

We begin with an initial object-oriented program that generates a prototype GUI. Figure 3 shows a screenshot of our tool, which consists of an object-oriented program (using the Featherweight-Java-like language BiFJ defined in Section 3.1) on the left; and an HTML menu bar (generated by forward evaluation of the program) on the right.

The source program is a usual object-oriented one, which consists of three classes, some GUI components, and the main function. Lines 1-12 are user-defined class declarations. The class Menu extends the class MenuItem and automatically inherits the field *title* and the method *setTitle*. The class MenuBar has a field called *menus* which is a list of Menu objects. Lines 13-33 define some GUI components: the horizontal menu bar with four menus (i.e., File, Edit, View, and Help); the menu File with three menu items (i.e., Save, Open, and New File); and the menu New File with two menu items (i.e., Word and PowerPoint). Lines 35-38 define the main function, which initializes the title of the "Open" menu item, the "New File" menu, and the "File" menu, and displays the whole menu bar.

2.2 Direct Manipulation

Direct manipulation is one of the most interesting parts of BiOOP, which can be divided into two categories: value modification of the HTML code and structure modification of the HTML code using components (corresponding to objects in the program) as units of action. As shown in Figure

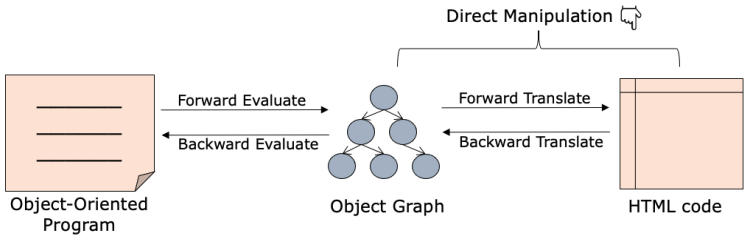

Fig. 4. Framework of BiOOP

Fig. 5. Templates for Custom Classes

4, BiOOP adopts a two-stage bidirectionalization: one is the bidirectional evaluation between the program and the object structure; the other is the bidirectional translation between the object structure and the HTML code rendered as an HTML page in the browser. Similar to Sketch-n-Sketch [Mayer et al. 2018], developers can manipulate the values (e.g., strings) in the HTML page directly or use the DevTools that come with the browser. BiOOP's innovation is the ability to manipulate the HTML page's GUI components by modifying the program's object structure.

In the second stage of Figure 4, interpreting objects as HTML code requires writing templates to specify the correspondence similar to Vue [You 2014] and Angular [Hevery and Abrons 2010]. The developer clicks the "Templates" button and fills the templates shown in Figure 5 into the pop-up form: (1) the developer fills the class name into the "Class" column; (2) the developer fills the object creation into the "Object Template" column, where the fields use variables as placeholders; (3) the developer fills the HTML defined with the variables declared in the object creation into the "HTML Template" column.

2.2.1 Value Manipulation. The developer can use the mouse and keyboard to directly manipulate the menu title "New File..." to "New File ⟩⟩", as shown in Figure 6(b). After the developer clicks the "Update" button, a new class Menu0 that extends Menu is added as shown in Figure 6(a), where the overriding method *setTitle* is distinct from *setTitle* of MenuItem (Lines 6-8) in its method body. Now, if the developer clicks the "Eval" button to execute the updated program, we can get exactly the same output as Figure 6(b), where the two angle brackets are added after "New File" while titles of the "File" menu and the "Open" menu item stay the same.

It would be interesting to see if the developer changes both "File (F)..." to "File (F) ⟩⟩" and "New File..." to "New File ⟩⟩", as shown in Figure 6(d), an overriding *setTitle* (Lines 12-14) will be inserted into the definition of Menu, which replaces "..." with " ⟩⟩", as shown in Figure 6(c). If the developer also changes "Open..." to "Open ⟩⟩" as shown in Figure 6(f), *setTitle* in MenuItem (Lines 6-8) will be updated by replacing "..." with " ⟩⟩", as shown in Figure 6(e).

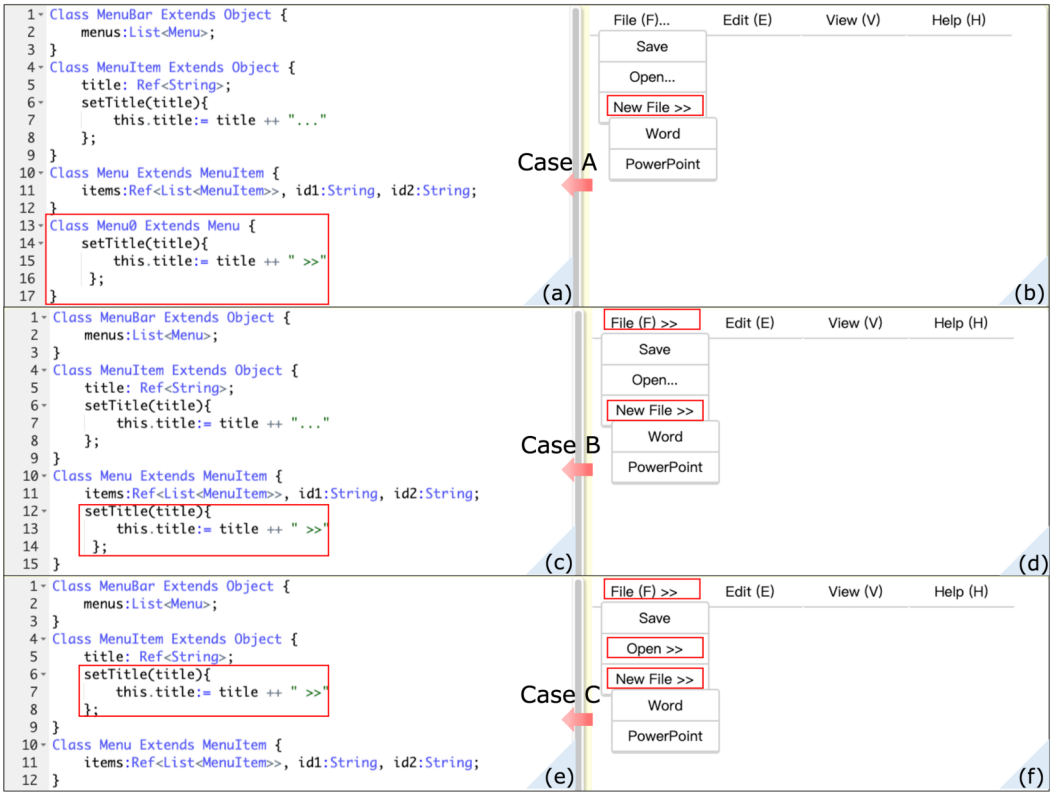

Fig. 6. Value Manipulation: modifying "New File..." to "New File ⟩⟩"

2.2.2 Structure Manipulation. We provide a set of operations to directly manipulate the object structure in BiOOP, including modifying the type of components, deleting components, and adding components. To illustrate these operations more specifically, we continue to alter the above GUI based on Case A in Figure 6.

Modifying Types. In our example, the "Save" menu has only one sub-item called "Save As", so the developer may wish to change the "Save" menu to a simple menu item, that is, a button that saves the file directly after a click. To do so, the developer clicks the "Structurize" button in the tool and right-clicks the "Save" menu. Then a small pop-up window appears nearby with three items: the first is the type of the component; the second is to modify the type; the third is to delete the component. The developer right-clicks the "Modify Type" button, as shown in Figure 7, and a list of component types appears below, which includes "MenuItem" and "Object" options (the current system only allows sub-classes to be changed to super-classes, and more options can be added later). The developer clicks the "MenuItem" option and finds that the type of the "Save" component is modified to "MenuItem" in the HTML page. Now a backward evaluation will result in the object creation declared in Line 21 of the program in Figure 3 being updated to `new MenuItem([ref "Save"])`.

Deleting Components. Suppose that the "Edit (E)" menu encounters some bugs, and the developer wants to delete it before debugging. As shown in Figure 8, after toggling the "Structurize" button,

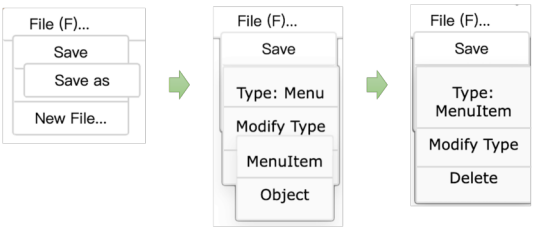

Fig. 7. Modifying Component Types: modifying the "Save" menu to the "MenuItem" type

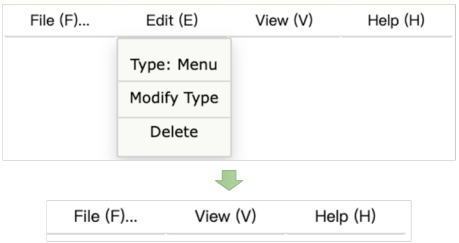

Fig. 8. Deleting Objects: deleting the "Edit (E)" menu on the top menu bar

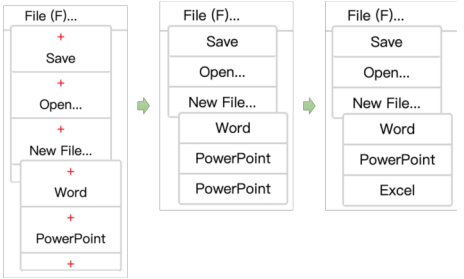

Fig. 9. Adding Objects: adding the "Excel" menu item to the "New File" menu

the developer right-clicks the "Edit (E)" menu and clicks the third option "Delete" in the pop-up menu, then the output changes to the bottom of Figure 8 where only three menus appear on the menu bar. Accordingly, the first element "Edit (E)" of the list in Line 19 is removed, as shown in Figure 10. After the developer clicks the "Eval" button, the `map_` function in Line 21 (built-in auxiliary function) transforms the string list into a Menu list, and there are only three menus (i.e., "File (F)", "View (V)", and "Help (H)") on the top menu bar in the HTML page.

Adding Components. As the editor features increase, the developer may want to add an "Excel" option to the "New File" menu. As shown in Figure 9, after the developer clicks the "Add One" button, the '+' symbols appear below the menu bar, menus, and menu items. The developer clicks the '+' symbol below the "PowerPoint" menu item in the "New File" menu. Then a "PowerPoint" option is added to the menu because when adding a new component after an existing component, our tool copies the previous one by default to provide valid initial parameters. The developer only needs to directly modify the string "PowerPoint" to "Excel" in the output. After clicking the "Update" button, an element "Excel" is added to the tail of the "New File" menu's option list defined in Line 33 of the program in Figure 10.

2.3 Summary

Through a series of direct manipulation on the HTML page, the final program and the menu bar are shown in Figure 10. The code fragments and the HTML page marked in red are updated. In summary, in an object-oriented program with the class inheritance hierarchy, recursive function `map_`, list data structures, and references (side effects), our tool can automatically synchronize the program with the HTML code by value and structure manipulation.

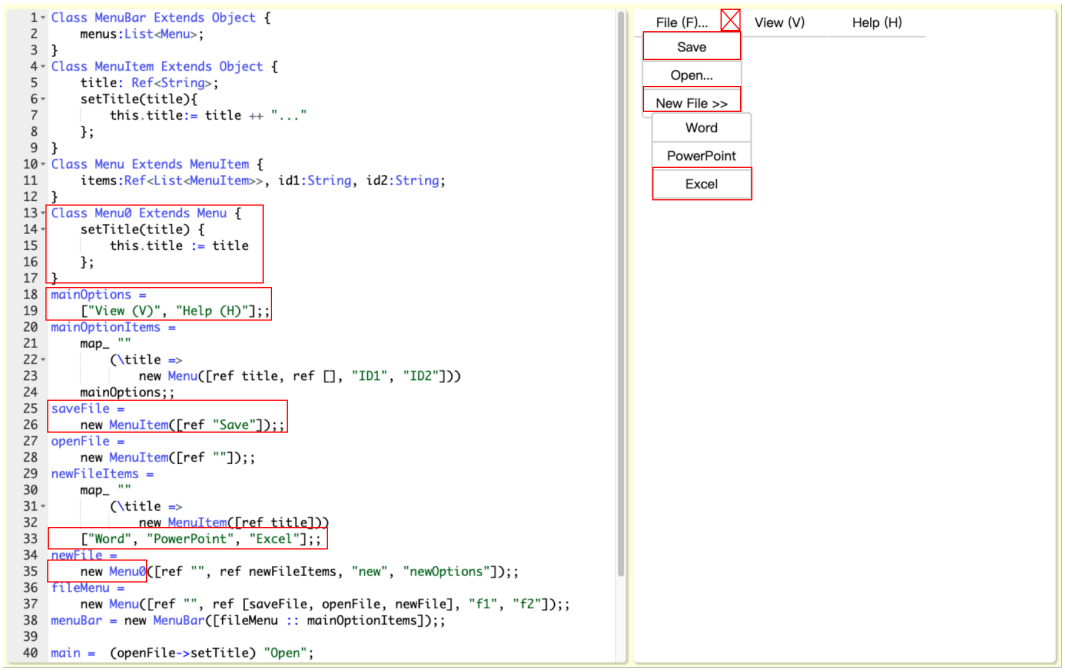

Fig. 10. Updated Program and Menu Bar

3 BIFJ: A BIDIRECTIONAL OBJECT-ORIENTED LANGUAGE

The core of BiOOP is a bidirectional object-oriented programming language BiFJ, whose syntax is adapted from the Featherweight Java (FJ) language with references, which are both defined in TAPL [Pierce 2002]. In this section, we give a formal definition of BiFJ. We define the syntax in Section 3.1. We present a bidirectional semantics for BiFJ and focus on the two language features: objects in Section 3.2 and references in Section 3.3. For other language features, such as conditionals and arithmetic expressions, their semantics is similar to that proposed in LittleLeo [Mayer et al. 2018], so they are omitted. Lastly, we illustrate that the bidirectional evaluation semantics satisfies the round-tripping properties in Section 3.4.

3.1 Syntax

The syntax of BiFJ is defined in Figure 11. BiFJ follows the syntax of FJ in TAPL [Pierce 2002] except for the type system. The meta-variable CL ranges over class declarations; M ranges over method declarations. The declaration $\texttt{class}\ C_1\ \texttt{extends}\ C_2\ \{\overline{f}; \overline{M}\}$ defines a class C_1 that extends class C_2. C_1 inherits all the fields and methods of C_2. Besides, C_1 can also define some local fields $\overline{f} \equiv \{f_1, ..., f_n\}$ and methods $\overline{M} \equiv \{M_1, ..., M_n\}$ and can override the methods in C_2. The method declaration M introduces a method named m with the parameter list p. The method body is the single statement $\texttt{return}\ t$ where the variable *this* referring to the object itself is bound in t. The class declaration table (class table for short) CT maps class names onto class declarations.

Terms mainly include standard terms, reference extensions, and FJ expressions. Standard terms include constants c, variables x, sequences $t_1; t_2$, list constructions $t_1 :: t_2$, tuples (t_1, t_2), primitive operations $t_1 \otimes t_2$, let-bindings $\texttt{let}\ x\ t_1\ t_2$, letrec-bindings (for recursions) $\texttt{letrec}\ x\ t_1\ t_2$, conditionals $\texttt{if}\ t_1\ t_2\ t_3$, and case expressions $\texttt{case}\ e\ (p_1, t_1) \cdots$. Reference extensions include reference

$$
\begin{array}{rcl}
\text{Class Decl. } CL & ::= & \texttt{class}\ C_1\ \texttt{extends}\ C_2\ \{\overline{f};\ \overline{M}\} \\
\text{Method Decl. } M & ::= & m(p)\ \{\texttt{return}\ t;\} \\
\text{Class Decl. Tables } CT & ::= & \emptyset \mid CT,\ C \mapsto CL \\
\\
\text{Terms } t & ::= & c \mid x \mid t_1 :: t_2 \mid (t_1,\ t_2) \mid t_1 \otimes t_2 \mid t_1; t_2 \\
& \mid & \texttt{let}\ p\ t_1\ t_2 \mid \texttt{letrec}\ p\ t_1\ t_2 \mid \texttt{if}\ t_1\ t_2\ t_3 \mid \texttt{case}\ t\ (p_1,\ t_1)\cdots \\
& \mid & \texttt{ref}\ t \mid !t \mid t_1 := t_2 \mid l \mid \texttt{unit} \mid t.f \mid t_1.m(t_2) \mid \texttt{new}\ C(t) \\
\text{Constants } c & ::= & n \mid b \mid s \mid [] \\
\text{Operators } \otimes & ::= & (+) \mid (*) \mid (<) \mid (\&\&) \mid \cdots \\
\text{Patterns } p & ::= & c \mid x \mid p_1 :: p_2 \mid (p_1,\ p_2) \\
\\
\text{States } S & ::= & \emptyset \mid S,\ l \mapsto v \\
\text{Environments } E & ::= & \emptyset \mid E,\ x \mapsto v \\
\text{Values } v & ::= & c \mid v_1 :: v_2 \mid (v_1,\ v_2) \mid l \mid \texttt{unit} \mid \texttt{new}\ C(v)
\end{array}
$$

Fig. 11. BiFJ Syntax

creations $\texttt{ref}\ t$, dereferences $!t$, assignments $t_1 := t_2$, and locations l. FJ expressions include the field access $t.f$, the method invocation $t_1.m(t_2)$, and the object creation $\texttt{new}\ C(t)$, where f is a field name, m is a method name, and the metavariable C is a class name.

Constants include numbers n, booleans b, strings s, empty list $[]$, and primitive operators. Values include constants, lists $v_1 :: v_2$, tuples (t_1, t_2), locations l, $\texttt{unit}$, and object creations $\texttt{new}\ C(v)$. Environments E bind the free variables in terms. States S map locations to values.

3.2 Bidirectional Semantics: Objects

In this section, we propose a bidirectional semantics for objects. We address the challenge of ambiguity in updating class tables (mentioned in Section 1) in two steps: in the backward evaluation, we adopt a straightforward modification to the class table called *subclassing* that always creates a sub-class and overrides the method; then, we optimize the class table through an algorithm called *Class Structure Refactoring*. In this section, the bidirectional evaluation semantics for objects is presented first, followed by *Class Structure Refactoring*.

3.2.1 Bidirectional Evaluation. The forward evaluation semantics for Featherweight Java (FJ) without side effects is presented in TAPL [Pierce 2002]. We expand the forward evaluation semantics with backward updating semantics for FJ with references, as shown in Figure 12. The big-step evaluation rules prefixed "F-" are the semantics for forward evaluation, which is a standard evaluator with a global state S and a class declaration table CT. The forward evaluation output includes not only a value v but also a state S_1, denoted as "$v; S_1$". The backward evaluation rules prefixed "B-" are novel and take the form $E; S; CT \vdash t \Leftarrow v'; S_1' \rightsquigarrow E'; S'; CT' \vdash t'$, which indicates that "when the output is altered to $v'; S_1'$, the program $E; S; CT \vdash t$ is updated to $E'; S'; CT' \vdash t'$". Following the statement in LittleLeo [Mayer et al. 2018], "push $v'; S'$ back to t" refers to the fact that "the original term t is updated by an altered output $v'; S'$".

Forward Evaluation $E; S; CT \vdash t \Rightarrow v; S_1$

Backward Evaluation $E; S; CT \vdash t \Leftarrow v'; S_1' \rightsquigarrow E'; S'; CT' \vdash t'$

$$\textbf{F-Const}\ \frac{}{E; S; CT \vdash c \Rightarrow c; S} \qquad \textbf{B-Const}\ \frac{}{E; S; CT \vdash c \Leftarrow c'; S' \rightsquigarrow E; S'; CT \vdash c'}$$

$$\textbf{F-Var}\ \frac{v = E(x)}{E; S; CT \vdash x \Rightarrow v; S} \qquad \textbf{B-Var}\ \frac{E' = [x \mapsto v']E}{E; S; CT \vdash x \Leftarrow v'; S' \rightsquigarrow E'; S'; CT \vdash x}$$

$$\textbf{F-Obj}\ \frac{E; S; CT \vdash t \Rightarrow v; S_1}{E; S; CT \vdash new\ C(t) \Rightarrow new\ C(v); S_1} \qquad \textbf{B-Obj}\ \frac{E; S; CT \vdash t \Leftarrow v'; S_1' \rightsquigarrow E'; S'; CT' \vdash t'}{E; S; CT \vdash new\ C(t) \Leftarrow new\ C'(v'); S_1' \rightsquigarrow E'; S'; CT' \vdash new\ C'(t')}$$

$$\textbf{F-Proj}\ \frac{E; S; CT \vdash t \Rightarrow new\ C(v); S_1 \quad n = findField\ CT\ C\ f_n \quad v_n = nth\ n\ v}{E; S; CT \vdash t.f_n \Rightarrow v_n; S_1} \qquad \textbf{B-Proj}\ \frac{E; S; CT \vdash t \Rightarrow new\ C(v); S_1 \quad n = findField\ CT\ C\ f_n \quad v' = updateNth\ n\ v_n'\ v \quad E; S; CT \vdash t \Leftarrow new\ C(v'); S_1' \rightsquigarrow E'; S'; CT' \vdash t'}{E; S; CT \vdash t.f_n \Leftarrow v_n'; S_1' \rightsquigarrow E'; S'; CT' \vdash t'.f_n}$$

$$\textbf{F-Invk}\ \frac{E; S; CT \vdash t_1 \Rightarrow new\ C(v_1); S_1 \qquad E; S_1; CT \vdash t_2 \Rightarrow v_2; S_2 \qquad (x, t_m) = mbody\ CT\ C\ m \qquad E, x \mapsto v_2, this \mapsto new\ C(v_1); S_2; CT \vdash t_m \Rightarrow v; S_3}{E; S; CT \vdash t_1.m(t_2) \Rightarrow v; S_3}$$

$$\textbf{B-Invk}\ \frac{\begin{array}{c} E; S; CT \vdash t_1 \Rightarrow new\ C(v_1); S_1 \quad E; S_1; CT \vdash t_2 \Rightarrow v_2; S_2 \quad (x, t_m) = mbody\ CT\ C\ m \\ E, x \mapsto v_2, this \mapsto new\ C(v_1); S_2; CT \vdash t_m \Leftarrow v'; S_3' \rightsquigarrow E', x \mapsto v_2', this \mapsto new\ C'(v_1'); S_2'; CT' \vdash t_m' \\ subclass = class\ C'_{sub}\ extends\ C'\ \{m(x)\{return\ t_m';\}\} \qquad E; S_1; CT \vdash t_2 \Leftarrow v_2'; S_2' \rightsquigarrow E_2; S_1'; CT_2 \vdash t_2' \\ E; S; CT \vdash t_1 \Leftarrow new\ C'_{sub}(v_1'); S_1' \rightsquigarrow E_1; S'; CT_1 \vdash t_1' \qquad CT_M = \{subclass\} \cup CT_1 \cup CT_2 \cup CT' \qquad E_M = E_1{}^{t_1} \oplus_E^{t_2} E_2 \end{array}}{E; S; CT \vdash t_1.m(t_2) \Leftarrow v'; S_3' \rightsquigarrow E_M; S'; CT_M \vdash t_1'.m(t_2')}$$

Fig. 12. Bidirectional Evaluation Semantics for Objects

We adopt the same classification of backward evaluation rules in LittleLeo [Mayer et al. 2018]—replacement rules overwrite terms in the program, while propagation rules propagate the altered output into sub-terms.

Replacement Rules. There are two rules for rewriting terms in the program, which are also responsible for propagating the updates of the global state. The rule B-Const states that when the output $c; S$ is altered to $c'; S'$, the term c is updated to c', and the state S is updated to S', respectively, while the environment E and class table CT remain unchanged.

The rule B-Obj is new to bidirectional programming, which is dedicated to object-oriented programming and extends what developers can manipulate on the output—not only values but also types of objects. The rule B-Obj first pushes the altered output $v'; S_1'$ back to t, producing an updated parameter t', as well as the updated environment E', state S', and class table CT'. One interesting point is that we allow modifying the class name of object creation, i.e., from C to C'. Combining the updated class C' with the parameter t', we get a new object creation term $\mathsf{new}\ C'(t')$.

In practice, updating class names of object creations facilitates the direct manipulation of component types in the output, such as changing a menu to a menu item in Section 2. Note that if the class C is updated to its sub-classes (with default values for the missing fields), then the program execution must not encounter problems because sub-class objects can call super-class methods. On the contrary, if the class C is updated to its super-class, the local fields of C must be removed, and

the program execution will fail when invoking the local methods of C. In summary, updating class names requires careful consideration of class hierarchy.

Propagation Rules. Propagation rules allow changes to values at the leaves of the program to flow throughout the program. We follow the core formulation of updating variables and variable binding forms in LittleLeo [Mayer et al. 2018], while our innovation is the backward semantics for field accesses and method invocations. To simplify the presentation, the evaluation and update rules for method invocations assume only variable patterns rather than arbitrary ones, as in our implementation.

Variables. When the output is altered to $v'; S'$, the rule B-Var updates the environment E to E', where x is bound to the new value v', and the rest remains unchanged. Then, the rule B-Var keeps the term x unchanged while updating the state S to S'. Note that $E(x)$ represents the value bound to the key x in the dictionary E and $[x \mapsto v']E$ represents replacing the binding of x with v'.

Field Accesses. For simplicity, we represent the field sequence in object creations by a list. Like TAPL [Pierce 2002], BiFJ initializes the inherited fields and the fields declared in the current class in turn. In the rule F-Proj, the auxiliary function *findField* first lists all the field names declared in C's super-classes and itself and then finds the index of the field f_n in that field list. The auxiliary function *nth* returns the n-th element of the field list v, which corresponds to the field f_n.

In the rule B-Proj, the first premise evaluates the term t to an object creation $new\ C(v)$, where v is a field list. Then, the auxiliary function *updateNth* replaces the n-th element of v with v'_n and returns v'. Finally, B-Proj pushes the altered output $new\ C(v'); S'_1$ back to the term t, resulting in an updated term t' and the updated context (short for the combination of the environment E', the state S', and the class table CT').

Method Invocations. As shown in Table 1, the rule B-Invk describes the backward evaluation semantics for method invocations that can be divided into five intuitive steps. $P_1 - P_9$ represent nine premises of B-Invk that are explained explicitly in the second "Detail" column. We take the example Case A in Section 2.2.1 to illustrate B-Invk in the third "Example" column. (Due to space, we only show crucial parts and omit the rest.)

Although B-Invk appears complicated, the core idea is straightforward. Similar to the rule in LittleLeo [Mayer et al. 2018] that updates function calls, B-Invk updates the method body first (P_4) and then updates the parameters (P_6) and receiver (P_7).

Nevertheless, a significant difference from LittleLeo [Mayer et al. 2018] is that B-Invk also updates the class table. Our updating strategy for class tables is called *subclassing*. It creates a sub-class that extends the receiver's class with the overriding method (P_5). Then *subclassing* appends the new sub-class to the class table (P_8). We decide to adopt this simple strategy because we hope that the backward semantics of BiFJ is clean and easy to reason. This strategy will yield many new sub-classes in the class table, which may be redundant. Consequently, we must refine the class table to reduce the class redundancy by *Class Structure Refactoring* (see Section 3.2.2).

In P_9, we adopt the merge operation "$^{t_1}\oplus_E{}^{t_2}$" that is defined in LittleLeo [Mayer et al. 2018] to merge two structurally equivalent environments E_1 and E_2. There are two implementations for the merge operation: "$^{t_1}\oplus^{t_2}$" is the *conservative two-way merge* operator where t_1 and t_2 were updated when E_1 and E_2 were produced; "$\oplus_E$" is the *optimistic three-way merge* operator where E is the original environment. We take Example 3.1 to give an intuitive explanation for two operators (see LittleLeo [Mayer et al. 2018] for detailed definitions). The two-way merge compares the two values and fails as soon as it encounters an inconsistent update; the three-way merge compares the

Table 1. Detail Explanation of B-Invk with An Example

Step	Detail	Example
Forward Evaluations of t_1 and t_2	P_1 evaluates the term t_1 to $new\ C(v_1); S_1$.	$\ldots \vdash newFile \Rightarrow new\ Menu(\ldots); \ldots$
	P_2 evaluates the term t_2 to $v_2; S_2$.	$\ldots \vdash$ "*New File*" $\Rightarrow$ "*New File*"; ...
	P_3 uses the auxiliary function *mbody* to find the formal parameter x and the method body t_m of m in CT, which may come from the definition of class C itself or its super-classes.	$(t, t +$ "..."$) =$ *mbody CT Menu setTitle*
Method Body Updating	P_4 pushes the altered output $v'; S'_3$ back to t_m under the environment $E, x \mapsto v_2, this \mapsto new\ C(v_1)$, the state S_2, and the class table CT, producing t'_m as well as the environment $E', x \mapsto v'_2, this \mapsto new\ C'(v'_1)$, an updated state S'_2, and an updated class table CT'.	$\{\ldots, t \mapsto$ "*New File*", *this* $\mapsto$ *new Menu*(...)$\}; \ldots$ $\vdash t +$ "..." $\Leftarrow$ "*New File* >> "; ... $\rightsquigarrow \ldots \vdash t +$ " >> "
Subclassing	P_5 creates a new class C'_{sub} extending the class C' and overriding the method m defined by t'_m. Note that the class name C'_{sub} is unique and does not conflict with other class names.	*subclass* = *class Menu0 extends Menu* { *setTitle*$(t)\{$*return* $t +$ " >> "; $\}\}$
t_1 and t_2 Updating	P_6 pushes the altered output $v'_2; S'_2$ back to t_2, producing an updated term t'_2 as well as E_2, S'_1, and CT_2.	$\ldots \vdash$ "*New File*" $\Leftarrow$ "*New File*"; ... $\rightsquigarrow \ldots \vdash$ "*New File*"
	P_7 pushes the altered output value $new\ C'_{sub}(v'_1)$ that combines v'_1 from $P4$ and C'_{sub} from P_5, as well as the updated state S'_1, back to t_1, resulting in an updated term t'_1 with E_1, S' and CT_1. Putting these pieces together, the final updated term is $t'_1.m(t'_2)$.	$\ldots \vdash newFile$ $\Leftarrow$ *new Menu0*$(\ldots); \ldots$ $\rightsquigarrow \ldots \vdash newFile$
Class Table and Env Merge	P_8 unions $\{subclass\}, CT_1, CT_2$ with CT'. In the backward evaluation, we simplify the updated class tables to only record the newly added classes and union multiple class tables directly because *subclassing* ensures the newly added classes are different.	$CT_M = \{subclass\} \cup \emptyset \cup \emptyset \cup \emptyset$ $= \{subclass\}$
	P_9 reconciles environments E_1 and E_2 with the original E. The symbol "$^{t_1}\oplus_E{}^{t_2}$" is the merge operator of environments discussed later.	$E_M = E\ {}^{t_1}\oplus_E{}^{t_2} E = E$

two values with the original one and heuristically prefers the value that differs from the original. We adopt the three-way merge for implementation because it is more powerful and can cover all situations where the two-way merge can do.

Example 3.1. Consider the program `let x=1 in [x,x]`. If the output value is altered to $[2, 3]$, the two-way merge fails because the two occurrences of the same source constant 1 are not updated consistently. However, in that case, the three-way merge may choose 2 when comparing 2 and 3 with the original 1. Therefore, the updated program is `let x=2 in [x,x]` using the three-way merge.

One more issue to note is that when an object creation is updated by two newly added sub-class objects generated by *subclassing*, the two sub-class definitions need to be merged in the class table, as shown in Example 3.2. Specifically, methods with different names can be merged directly, while methods with the same name but different definitions needs to be merged using the three-way

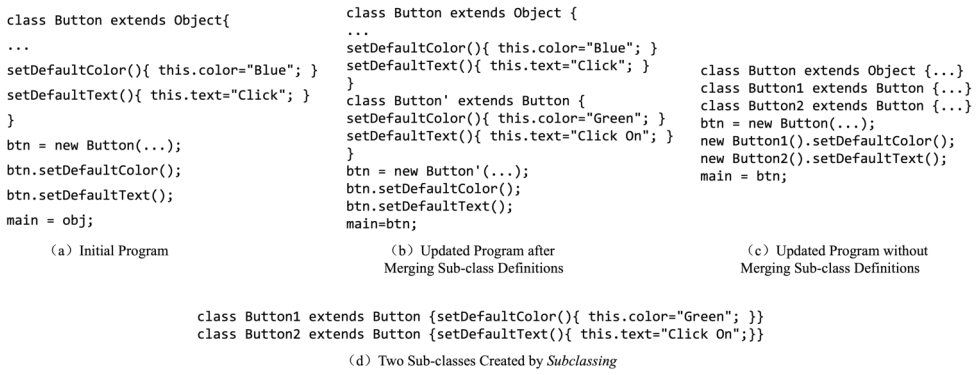

Fig. 13. An Example for Merging Sub-class Definitions

merge. In our implementation, we merge the definitions of the two sub-classes when a variable is updated as objects of two different sub-classes in the environment merge (P_9).

Example 3.2. Consider the program in Figure 13(a). The class Button owns two methods, *setDefaultColor* and *setDefaultText*. Suppose that the color of the output button is changed to "Green" and the text is changed to "Click On". There are two steps in the backward updating: (1) Two new sub-classes Button1 and Button2 of Button are temporarily created, as shown in Figure 13(d); (2) Another new sub-class Button' of Button is created to merge Button1 and Button2, and *btn* is updated to the class Button'. The final updated program is shown in Figure 13(b).

We explain below why we merge the two sub-classes instead of replacing the receivers of the two calls with objects from two different sub-classes. For one thing, using Example 3.2 as an example, due to the imperative feature of our language, changing the receiver of the call to an object of a newly created sub-class would make the updated program unreasonable, as shown in Figure 13(c). For another, to reduce the ambiguity of program updates, we update the constants and classes of objects in the program only, as mentioned in the Replacement Rules section.

3.2.2 Class Structure Refactoring. For the problem of class redundancy caused by *subclassing* in backward evaluation, we adopt a heuristic strategy, *Class Structure Refactoring*, to optimize the class table. *Class Structure Refactoring* aims to minimize the class table changes as much as possible without changing the program behavior. The optimized class table is better for developers in terms of comprehensibility because the number of newly added classes and new overriding methods is reduced.

Intuitively, *Class Structure Refactoring* is a process that combines two popular code refactoring operations, i.e., *Pull Up Method* to move methods from sub-classes up to the super-class and *Collapse Hierarchy* to remove empty sub-classes. Of course, we adapt these refactoring operations to the scenario of bidirectional evaluation: for *Pull Up Method*, we move only the updated methods in the backward evaluation, i.e., the methods that are overridden in the newly added classes; for *Collapse Hierarchy*, we delete the empty classes that are newly added to the class table by the backward evaluation (see B-Invk) and change the class name of the corresponding object creations in the program to their parent classes. It is worth noting that *Class Structure Refactoring* does not delete originally existed classes, which is an important prerequisite for the correctness (Theorem 3.3) to hold.

$$\frac{}{c : \emptyset \triangleleft t \Uparrow c : \emptyset} \qquad \frac{\begin{array}{c} T_1 \triangleleft t \Uparrow c_1 : [T'_{11}, \dots] \quad \dots \quad T_k \triangleleft t \Uparrow c_k : [T'_{k1}, \dots] \\ M = \{m_{t_i} | \exists i \,.\, c_i.m^{\uparrow}_{t_i},\ c \not\downarrow_m t,\ \forall j \in [1,k] \,.\, c_j.m^{\uparrow}_{t_i} \vee (c_j : [T'_{j1}, \dots]) \models_m t\} \\ c' = c \oplus M \qquad c'_1 = c_1 \ominus M \quad \dots \quad c'_k = c_k \ominus M \end{array}}{c : [T_1, \dots, T_k] \triangleleft t \Uparrow c' : [(c'_1 : [T'_{11}, \dots]), \dots, (c'_k : [T'_{k1}, \dots])]}$$

Fig. 14. Class Structure Refactoring: $T \triangleleft t \Uparrow T'$

$$\frac{c.m \vee c \not\downarrow_m t}{c : \emptyset \models_m t} \qquad \frac{c.m \vee (c \not\downarrow_m t \wedge T_1 \models_m t \wedge \dots \wedge T_k \models_m t)}{c : [T_1, ..., T_k] \models_m t}$$

Fig. 15. T's Calls to m in the Forward Evaluation of t are Independent of T's Ancestors: $T \models_m t$

Algorithm. The formal definition of *Class Structure Refactoring* is given in Figure 14. We model the class table as a class tree $c : [T_1, \dots, T_k]$ built according to class inheritance relationships, where c is the root class and $[T_1, \dots, T_k]$ is the sequence of sub-trees. Note that c denotes a class definition that contains a class name C and method definitions. We use "$c.m$" to access the definition of the method m. The symbol "$\Uparrow$" is the refactoring operator that takes two arguments, a class tree T and a term t that T is defined for, and returns the refactored class tree T'. As shown in Figure 14, the base case is that a single class without children remains unchanged after refactoring.

For a class tree $c : [T_1, \dots T_k]$, the first step is to refactor sub-trees $T_1, \dots, T_k$ and obtain $c_1 : [T'_{11}, \dots], \dots, c_k : [T'_{k1}, \dots]$. The second step is to find all methods M in $[c_1, \cdots, c_k]$ that can be moved to the parent class c. The third step is to add methods in M to the class c indicated by the symbol "$\oplus$" and remove methods in M from $[c_1, ..., c_k]$ indicated by the symbol "$\ominus$".

There are three conditions that need to be satisfied when moving the method m_{t_i} from sub-classes to their parent class. The first condition $\exists i \,.\, c_i.m^{\uparrow}_{t_i}$ states that there is at least one sub-class c_i having the method m with the body t_i that can be moved. Note that m_{t_i} represents the signature of the method is m, the method body is t_i, and the symbol "$\uparrow$" at the top-right corner represents that the method is obtained by *subclassing* and does not originally exist. The second condition $c \not\downarrow_m t$ states that, in the forward evaluation of t, the objects of class c do not call the method m. The third condition $\forall j \in [1,k] \,.\, c_j.m^{\uparrow}_{t_i} \vee (c_j : [T'_{j1}, \dots]) \models_m t$ states that for all sub-classes $[c_1, \cdots, c_k]$, either the class c_j has the method m with the same body as t_i that can be moved or, in the forward evaluation of t, the calls of the sub-tree rooted as c_j to the method m is independent of its ancestors, represented as $(c_j : [T'_{j1}, \dots]) \models_m t$.

The judgment $T \models_m t$ is defined in Figure 15. If the class c initially has the unmovable method m (i.e., $c.m$) or $c \not\downarrow_m t$, then $c : \emptyset \models_m t$ holds. For a class tree $c : [T_1, \cdots, T_k]$ whose depth is at least 1, the premise of this judgment is either $c.m$ or $c \not\downarrow_m t \vee T_1 \models_m t \wedge, \cdots, \wedge T_k \models_m t$.

Correctness. We now present the correctness that *Class Structure Refactoring* does not change the execution result of the program, as described in Theorem 3.3.

Theorem 3.3. *If* $T \triangleleft t \Uparrow T'$, $T \vdash t \Rightarrow v_1$, $T' \vdash t \Rightarrow v_2$, *then* $v_1 = v_2$.

If the class table T is refactored to T' under the term t, t evaluates to v_1 under the class table T, and t evaluates to v_2 under the refactored class table T', then v_1 equals v_2. For simplicity, we omit environments and states in the forward evaluation and use the original term t instead of the term that replaces the class names deleted from refactoring with the ones of their parent classes. This is because a class that does not have any new methods or fields is equivalent to its parent

$$\textbf{F-Ref}\ \frac{E;S;CT \vdash t \Rightarrow v;S_1 \quad l \notin dom(S_1)}{E;S;CT \vdash \texttt{ref}\ t \Rightarrow l;S_1 \cup \{l \mapsto v\}} \qquad \textbf{B-Ref}\ \frac{v' = S_1''(l) \quad S_1' = S_1''/\{l\} \quad E;S;CT \vdash t \Leftarrow v';S_1' \rightsquigarrow E';S';CT' \vdash t'}{E;S;CT \vdash \texttt{ref}\ t \Leftarrow l;S_1'' \rightsquigarrow E';S';CT' \vdash \texttt{ref}\ t'}$$

$$\textbf{F-Deref}\ \frac{E;S;CT \vdash t \Rightarrow l;S_1 \quad v = S_1(l)}{E;S;CT \vdash !t \Rightarrow v;S_1} \qquad \textbf{B-Deref}\ \frac{E;S;CT \vdash t \Rightarrow l;S_1 \quad S_1' = [l \mapsto v' \oplus_{S_1(l)} S_1''(l)]S_1'' \quad E;S;CT \vdash t \Leftarrow l;S_1' \rightsquigarrow E';S';CT' \vdash t'}{E;S;CT \vdash !t \Leftarrow v';S_1'' \rightsquigarrow E';S';CT' \vdash !t'}$$

$$\textbf{F-Assign}\ \frac{E;S;CT \vdash t_1 \Rightarrow l;S_1 \quad E;S_1;CT \vdash t_2 \Rightarrow v_2;S_2}{E;S;CT \vdash t_1 := t_2 \Rightarrow unit;[l \mapsto v_2]S_2}$$

$$\textbf{B-Assign}\ \frac{\begin{array}{c} v_2' = S_2''(l) \quad S_2' = [l \mapsto S_2(l)]S_2'' \\ E;S;CT \vdash t_1 \Rightarrow l;S_1 \quad E;S_1;CT \vdash t_2 \Leftarrow v_2';S_2' \rightsquigarrow E_2;S_1';CT_2 \vdash t_2' \quad CT_M = CT_1 \cup CT_2 \\ E;S_1;CT \vdash t_2 \Rightarrow v_2;S_2 \quad E;S;CT \vdash t_1 \Leftarrow l;S_1' \rightsquigarrow E_1;S';CT_1 \vdash t_1' \quad E_M = E_1{}^{t_1} \oplus_E {}^{t_2} E_2 \end{array}}{E;S;CT \vdash t_1 := t_2 \Leftarrow unit;S_2'' \rightsquigarrow E_M;S';CT_M \vdash t_1' := t_2'}$$

Fig. 16. Bidirectional Evaluation Semantics for Referencs

class. Detailed proof is given in Appendix A. Note that we do not discuss the termination of *Class Structure Refactoring* because, in the domain of bidirectional evaluation, backward evaluation only considers terminated programs.

3.3 Bidirectional Semantics: References

This section presents the bidirectional semantics' design principles and concrete evaluation rules for references, as shown in Figure 16. The description of forward and backward judgments is explained in Section 3.2.

3.3.1 Design Principles. Unlike functional environments, the forward evaluator's access to the state (that records values pointed by references) is serial and ordered. That is precisely why the backward evaluation should be in the exact opposite order as the forward evaluation, and the state can be handled linearly without merging. The opposite order of P_1, P_2, and P_6, P_7 in Table 1 can illustrate that point.

In addition, each reference assignment overwrites the effect of the previous one. For example, the program `a:=1;a:=2;!a` outputs 2 because the second assignment `a:=2` overwrites the effect of `a:=1`. Assignments executed early in the forward evaluation are updated later in the backward evaluation, and assignments executed later in the forward evaluation are updated early. Therefore, in the backward evaluation, we need to roll back the reference value to the one computed by the early assignment after updating the late assignment. In the above example, after updating $a := 2$, we need to roll back the value of a to 1 (computed by $a := 1$). The concrete semantics is described in B-Assign.

In summary, each backward evaluation rule for updating references $E;S;CT \vdash t \Leftarrow v';S_1' \rightsquigarrow E';S';CT' \vdash t'$ should guarantee that: (1) the updated state S' is structurally equivalent to S, i.e., both have the same domain; (2) the effects caused by the term t in the updated state S_1' should be rolled back while ensuring the other updates are forwarded to the rest backward evaluation; (3) when forward evaluating the term t', the state S' must be re-evaluated to the modified state S_1' after the backward evaluation (i.e., PUTGET property discussed in Section 3.4). Note that although

Table 2. Detail Explanation of B-Assign with An Example

Step	Detail	Example
Forward Evaluations of t_1 and t_2	P_3 evaluates the term t_1 to $l; S_1$.	$\{a \mapsto l\}; \{l \mapsto 0\}$ $\vdash a \Rightarrow l; \{l \mapsto 0\}$
	P_6 evaluates the term t_2 to $v_2; S_2$.	$\{a \mapsto l\}; \{l \mapsto 0\}$ $\vdash !a + 1 \Rightarrow 1; \{l \mapsto 0\}$
State Rollback	P_1 obtains v_2' from S_2'' according to the location l.	$2 = \{l \mapsto 2\}(l)$
	P_2 restores the value of location l in S_2'' to $S_2(l)$.	$\{l \mapsto 0\} =$ $[l \mapsto \{l \mapsto 0\}(l)]\,\{l \mapsto 2\}$
t_1 and t_2 Updating	P_4 pushes $v_2'; S_2'$ back to the term t_2 under E, CT, and S_1 obtained in P_3, producing the updated term t_2' and the updated context (i.e., E_2, S_1', and CT_2).	$\{a \mapsto l\}; \{l \mapsto 0\} \vdash !a + 1$ $\Leftarrow 2; \{l \mapsto 0\}$ $\rightsquigarrow \{a \mapsto l\}; \{l \mapsto 0\} \vdash !a + 2$
	P_7 pushes $l; S_1'$ back to the term t_1, producing the updated term t_1' and the updated context (i.e., E_1, S', and CT_1). Putting these pieces together, the final updated term is $t_1' := t_2'$.	$\{a \mapsto l\}; \{l \mapsto 0\} \vdash a$ $\Leftarrow l; \{l \mapsto 0\}$ $\rightsquigarrow \{a \mapsto l\}; \{l \mapsto 0\} \vdash a$
Class Table and Env Merge	P_5 unions CT_1 with CT_2, similar to B-Invk.	Omitted.
	P_8 reconciles environments E_1 and E_2 with the original E.	$\{a \mapsto l\} =$ $\{a \mapsto l\}^{t_1} \oplus_E {}^{t_2} \{a \mapsto l\}$

references are an important feature of object-oriented programs, their bidirectional semantics is independent of the object part, so we omit the description of class tables in this section.

3.3.2 Reference Creations. The rule B-Ref first obtains a value v' from the updated state S_1'' according to the location l. Note that the term t' should evaluate to the value v' by F-Ref. Compared with the structure of S_1, there is one more location l in S_1'', which should be removed to keep the same as S_1. In the second premise, the notation "D/K" denotes the removal of elements from a dictionary D that belong to the domain K. Then, B-Ref pushes the altered result $v'; S_1'$ back to the term t, producing the updated term t' and the updated context (i.e., E', S', and CT'). The new term is a reference creation of t', i.e., *ref* t'.

Example 3.4. Consider the term `ref 1` under an empty state, which evaluates to a location l with the state $\{l \mapsto 1\}$. Since we do not allow changes to the location, l must keep unchanged, and the state may be modified to $\{l \mapsto 2\}$. B-Ref pushes the altered value 2 pointed by l as well as the empty state $\emptyset$ (i.e., S_1') back to the term 1, producing an updated term 2 by B-Const. The final updated program is `ref 2` with the empty state S'. □

3.3.3 Dereferences. The rule B-DeRef first evaluates the term t to a location value l with the state S_1. The second premise then replaces the element pointed by the location l with the merged value $v' \oplus_{S_1(l)} S_1''(l)$ in the updated state S_1'', producing the updated state S_1'. The third premise pushes the altered output $l; S_1'$ back to the term t, producing the updated term t' and the updated context (i.e., E', S', and CT'). The new term is a dereference of t', i.e., $!t'$.

Because we allow both the value and the state in the output to be modified, the operator "$\oplus_{S_1(l)}$" is used to merge the two values v' and $S_1''(l)$ that may be updated. Similar to environment merge, the operator $\oplus_v$ has two variations: one is the two-way value merge that updating succeeds if the

two values are consistent or fails if not; the other is the three-way merge that recursively compare the two values with the original value and choose the different one.

Example 3.5. Consider the term $!x$ under an environment $\{x \mapsto l\}$ and a state $\{l \mapsto 1\}$, which evaluates to $1; \{l \mapsto 1\}$. Suppose the output is altered to $2; \{l \mapsto 2\}$. The second premise replaces 1 pointed by l with 2 in the state. The third premise pushes the altered output $l; \{l \mapsto 2\}$ back to x, resulting in the updated state $\{l \mapsto 2\}$, as well as the unchanged term and environment. □

3.3.4 Assignments. The backward semantics of B-Assign is split into four parts, as shown in Table 2. $P_1 - P_8$ are eight premises of B-Assign (from top to bottom, left to right). We take an example to demonstrate each premise shown in the third "Example" column. Consider the term `a:=!a+1` under the environment $\{a \mapsto l\}$ and the state $\{l \mapsto 0\}$. Suppose the output is altered to `unit`; $\{l \mapsto 2\}$. Updating semantics for additions in P_4 chooses to update the right operand, which is user-customizable in the implementation.

The effect rollback in P_2 is necessary. Suppose that we do not roll back the value of l in p_2, then the final updated program is `a:=!a+2` under the environment $\{a \mapsto l\}$ and the state $\{l \mapsto 2\}$, which evaluates to `unit`; $\{l \mapsto 4\}$.

3.4 Correctness

The forward evaluation and the backward evaluation form a lens [Foster et al. 2007] that maintains consistency between an object-oriented program (*source*) and the output (*view*). To ensure the well-behavedness of our system, the bidirectional evaluation should satisfy round-tripping properties, i.e., GetPut and PutGet [Foster et al. 2007], which are defined as follows.

Theorem 3.6 (GetPut). If $E; S; CT \vdash t \Rightarrow v; S_1$, then $E; S; CT \vdash t \Leftarrow v; S_1 \rightsquigarrow E; S; CT \vdash t$.

Theorem 3.7 (PutGet). If $E; S; CT \vdash t \Leftarrow v'; S_1' \rightsquigarrow E'; S'; CT' \vdash t'$, then $E'; S'; CT' \vdash t' \Rightarrow v'; S_1'$.

Intuitively, GetPut states that if we get an output o from a program p and immediately push o (with no modifications) back to p, we must get back exactly p. PutGet, on the other hand, demands that the backward evaluation must capture all of the information contained in the output: if pushing an output o back to a program p yields an updated program p', then the output obtained from p' is exactly o.

We can prove that the bidirectional semantics of BiFJ satisfy GetPut and PutGet if the developer modifies the output consistently. If the output is modified inconsistently, like in Example 3.1, this means that the modified output exceeds the program's output range, where only the constants are allowed to be modified. Therefore, in the correctness proof, we only consider that the output is modified consistently like LittleLeo [Mayer et al. 2018], which means that using the two-way merge always succeeds in the backward evaluation.

The formal proof of satisfaction of GetPut and PutGet using the traditional two-way merge is available in the anonymous code library. We use bidirectional evaluation rules of assignments as an example to demonstrate the informal proof in Appendix B and Appendix C. It is added that, in practice, we use the three-way merge to handle the possible inconsistent output to avoid updating failures (mentioned in Section 3.2.1). However, using the three-way merge may cause PutGet to be unsatisfied because it loses output information.

4 IMPLEMENTATION

To validate the practicality of our approach to bidirectional object-oriented programming, we implemented a prototype programming environment called BiOOP to support our language BiFJ for developing HTML webpages, which is available at https://github.com/xingzhang-pku/BiOOP. We contributed nearly 9000 lines of Elm code in our implementation.

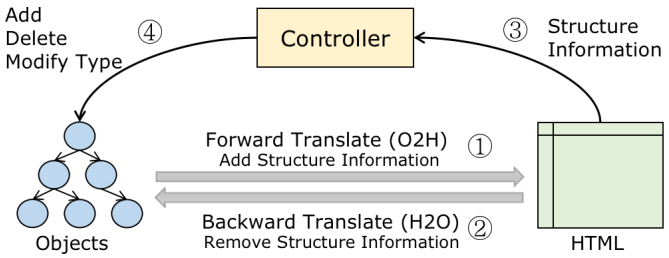

Fig. 17. The BX between Objects and HTML

4.1 Framework

The framework of BiOOP is shown in Figure 4. BiOOP adopts a two-stage bidirectionalization: one is between the object-oriented program and the object structure; the other is between the object structure and the HTML code. The first bidirectional transformation (BX for short) is the core of our tool discussed in Section 3. The second one is shown in Figure 17. Note that the second mapping (called O-bx-H) is not difficult and has been studied widely; many MVC frameworks (e.g., Vue, Angular, and React) can be directly used.

Our key insight is that updating the object structure through the direct manipulation of outputs requires two-stage bidirectionalization. Assume that the source object-oriented program is evaluated to the HTML code directly and developers can make arbitrary modifications to the HTML code. Although it is very flexible, a practical problem arises: because of the lack of correspondence between the HTML code and the object structure in the program, the developers' arbitrary modifications to the output may not be propagated to the source program due to the lack of restrictions. For example, developers may delete a "button" element even though it may be part of an object in the program or even delete the "id" attribute of an element that could be an object field. To handle this problem, we limit the developer's modification of the HTML code by translating the HTML code into an object structure before updating the source program.

4.2 BX between Objects and HTML

As shown in Figure 17, we give a simple implementation of O-bx-H to illustrate the connection and difference with the existing MVC frameworks. O-bx-H contains two parts: a bidirectional translation and an object structure controller.

The bidirectional translation is similar to sugaring and desugaring. The forward translation from objects to HTML (Arrow 1) is briefly described as follows: we recursively match objects with object templates to obtain the field bindings and then replace the variables in the HTML templates with the binding values. The structure information of objects is added to the HTML during translation. The backward translation from HTML to objects (Arrow 2) is similar to the forward translation, which removes the structure information.

The object structure controller extracts the structure information from the GUI component that is directly manipulated (Arrow 3) and modifies the object structure (such as modifying the types of objects, adding an object, or deleting an object) obtained from the program (Arrow 4). Through the forward translation from objects to HTML, developers can observe the structural modification result in HTML.

As a whole, O-bx-H is a kind of MVC framework, but it differs a bit from existing ones. Take Vue (a JavaScript framework) [You 2014] as an example, developers need to implement a listener for each type of direct manipulation to reflect the updates on the HTML code to objects. O-bx-H adds

the structure information to the HTML code to automatically achieve the modifications to the object structure.

4.3 Implementation Details of O-bx-H

In the implementation of O-bx-H, there are three significant issues: (1) what kind of structural information needs to be injected in HTML to enable the direct manipulations of the object structure; (2) how to handle the information loss during the object-to-HTML translation and inconsistent updates of the HTML code during the HTML-to-object translation; (3) how to handle references among objects.

4.3.1 Object Structure Information. When translating an object into an HTML element, we must embed a trace link back to the object (i.e., the object ID) and the information about what kinds of manipulations are allowed into the HTML element. Regarding object IDs, we apply the depth-first traversal to the objects, and the ID of an object is computed by combining the order in which the object is visited with the object's class name. Then, we store the object ID to the *id* attribute of the corresponding HTML element.

To embed the kinds of allowed manipulations, we add the value "Object" to the *class* attribute of the HTML element, which implies that the object type modification and the object deletion are enabled for this element. Moreover, suppose the HTML element is translated from an object in an object list. In that case, we also append the value "Add" to the *class* attribute, which means that the object insertion after this element is enabled.

4.3.2 Translation Complements. The object-to-HTML translation may be an information-losing transformation. For example, a field of an object may not be finally serialized to the HTML. On the other hand, the same field may appear multiple times in HTML and is modified inconsistently. Our solution to the above two issues is to keep an object-to-environment dictionary in the object-to-HTML translation, representing the information discarded (i.e., *complements* [Foster et al. 2012]). Besides, the three-way merge is used to combine multiple updates.

4.3.3 References in Objects. Developers usually set fields to reference types to facilitate class methods to modify class fields. Therefore, we allow object fields in the object structure computed by the program to be references. However, because the location values cannot be translated directly into HTML, we replace them with the corresponding values in the global state. Therefore, when developers modify the HTML fragment corresponding to the object pointed by a reference, they actually modify the global state.

5 EVALUATIONS

The evaluation aims to analyze the expressiveness of our language (BiFJ) and the efficiency of our tool (BiOOP) to produce a single HTML webpage, from the perspective of programming language researchers in the context of object-oriented GUI programming. We mainly focus on the following two research questions:

(1) **RQ.1** Can BiFJ be used to develop typical GUI programs?
Rationale. BiFJ is intended to cover the scenario of GUI design. Hence, we must evaluate whether BiFJ is expressive enough to specify typical GUI programs. In Section 5.1, we collect 11 benchmark programs of Java Swing [Eckstein et al. 1998], a classical GUI framework in Java, and reproduce them using BiFJ in BiOOP.
(2) **RQ.2** Can our tool efficiently respond to the developers' edits by propagating the edits bidirectionally within reasonable time cost when developing a single HTML webpage?
Rationale. We hope our tool can support interactive GUI design. If our tool cannot propagate

Table 3. Benchmarks

ID	Example	LOC	#Cls	Depth of CT	Width of CT	#Obj
1	File Explorer	188	9	4	4	21
2	Source Getter	189	9	4	3	5
3	IP Finder	201	9	4	3	5
4	Number Puzzle	222	7	4	2	12
5	Picture Puzzle	238	7	4	2	13
6	Online Exam	245	10	5	2	54
7	Word Counter	274	12	4	3	9
8	Tic Tac Toe	287	8	4	2	13
9	NotePad	331	11	4	3	37
10	Calculator	371	11	5	3	29
11	Data Sync	631	22	5	10	97

LOC: Lines of code in BiFJ; **#Cls**: Numbers of classes; **Depth of CT**: Length of the longest inheritance chain in the class table; **Width of CT**: Maximum number of sub-classes of the same class; **#Obj**: Numbers of objects;

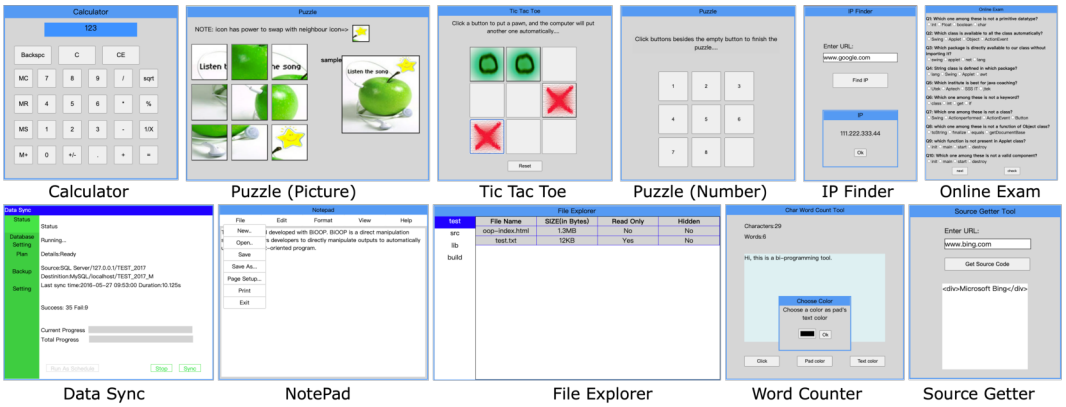

Fig. 18. Screenshots of Applications

a developer's edit efficiently, it may be put into practice. In Section 5.2, we conduct an experiment to show that the efficiency of our tool is acceptable.

5.1 Examples

To demonstrate the expressiveness of BiFJ, we reproduce 11 real-world Java Swing programs with BiFJ. The first 10 programs are collected from a popular Java Swing tutorial available at JavaTpoint[3]; the last program, called Data Sync, is the largest program collected from a GitHub project[4] with 790 stars. Note that we mainly focus on the GUI part and the class table definitions of these programs.

We use BiFJ to implement the 11 benchmark programs. The overall information on the implemented programs is listed in Table 3. The "LOC" column shows the lines of BiFJ code of these programs; the "#Cls" column shows the numbers of classes; the "Depth of CT" column shows the lengths of the longest inheritance chain in the class table; the "Width of CT" column shows the

[3]https://www.javatpoint.com/java-swing.
[4]https://github.com/rememberber/WeSync.

Table 4. Performance Experiment Results

ID	#DMP	Fwd	F:Tot	Fastest			Slowest			Avg		
				Bwd	CSR	B:Tot	Bwd	CSR	B:Tot	Bwd	CSR	B:Tot
1	10	1	655	10	1	626	11	1	799	11	1	671±26
2	9	1	143	5	1	114	5	1	161	5	0	145±29
3	10	1	133	5	1	118	5	1	165	5	1	151±13
4	10	3	313	29	1	357	255	1	581	38	1	406±73
5	9	3	378	34	1	447	152	1	630	32	1	495±14
6	10	3	1394	38	1	1540	255	1	1946	197	1	1698±152
7	11	3	329	20	1	329	19	2	397	21	1	377±26
8	10	6	354	2017	2	2642	2030	2	2706	2013	2	2684±31
9	11	2	1008	12	0	912	12	1	1242	11	1	1124±117
10	9	3	1015	26	2	1147	677	4	1891	26	2	1189±22
11	10	2	3855	499	1	4533	514	2	4802	502	2	4679±162

#DMP: Numbers of Direct Manipulations; **Fwd**: Time to forward evaluation **in milliseconds** for initial programs; **F:Tot**: Total time between clicking the "Eval" button and getting the output; **Bwd**: Time to backward evaluation; **CSR**: Time to class table lifting; **B:Tot**: Total time between clicking the "Update" button and getting the updated program. The data 0 in the table is due to less than 0.5 before retaining integers.

maximum numbers of sub-classes of the same class; the "#Obj" column shows the number of objects. The largest program, i.e., Data Sync, has 631 lines of code and 97 objects. There are 7–12 classes in the ten classic applications and 22 in Data Sync. All class tables have an inheritance chain whose length is at least 4, and the class JComponent in Data Sync has 10 sub-classes. Figure 18 illustrates the screenshots of the 11 programs running in our tool. All the GUI programs work as expected.

Based on Table 3 and Figure 18, our answer to **RQ.1** is **yes**—*BiFJ has sufficient expressiveness to develop GUI programs*. Because we successfully apply BiFJ to the development of these benchmark programs, we think that BiFJ has covered the major features needed in object-oriented GUI programming and can be adapted to handle real-world object-oriented programs.

5.2 Performance Experiment

We conduct a performance experiment to evaluate whether our tool is efficient enough to give the developers quick feedback after directly manipulating the output. We instrument a timer mechanism in our implementation and measure the execution time for the 11 benchmark programs. For each program, we perform 9-11 direct manipulations on its execution output, documented in Appendix D. The selection of direct manipulations covers value manipulations and structure manipulations (including adding objects, deleting objects, and modifying types). We randomly select 1-5 test cases for each kind of direct manipulation and randomly cover different types of objects. We repeat each manipulation 10 times to calculate the average time cost. The experiment is conducted on a laptop with Windows 10, Intel® Core™ i7-11800H @ 2.30GHz, and 16 GB RAM.

The results of the performance experiment are shown in Table 4. The "#DMP" column shows the number of direct manipulations. The "Fwd" column shows the time costs of the forward evaluation. The "F:Tot" column shows the total forward running time between clicking the "Eval" button and getting the output, including the running time of the parsing, forward evaluation, translation from objects to HTML, and printing. The "Bwd" column shows the running time of the backward evaluation. The "B:Tot" column shows the total backward running time between clicking the "Update" button and getting the updated program, including the running time of the parsing, translation from HTML to objects, backward evaluation, class table lifting, and printing. The "CSR"

column shows the running time of the Class Structure Refactoring. All the time is measured in **milliseconds**.

According to Table 4, the forward evaluation takes 1 ms ∼ 6 ms to compute the program output. Considering all the time costs of the forward transformation, our tool consumes 133 ms ∼ 3855 ms. Except for Data Sync, the most time-consuming program, our tool can return the forward results in 1.5 s.

Regarding the backward evaluation, on average, our tool can propagate each direct manipulation backward within 5 ms ∼ 2013 ms. Backward evaluation needs more time that forward evaluation because it is more computationally complicated. The increase in the time spend during backward evaluation of benchmarks 4, 5, 6, 8, 10, 11 is caused by large data structures (mainly strings and lists), which can affect the backward evaluation time to a much greater extent than other metrics. The time cost of class table lifting is generally <5 ms, which is insignificant during backward transformation. For the total time costs of the backward transformation, our tool can finish the computation within 145 ms ∼ 4679 ms. Data Sync is also the most time-consuming program (4679 ms on average).

Based on Table 4, our answer to **RQ.2** is **yes**—our tool can both the forward and the backward transformation within an acceptable time cost. Since the maximum response time in Table 4 is less than 5 seconds, we regard the performance of our tool as acceptable. There are three detailed observations.

1) *Quick Backward Evaluation.* The running time of the backward evaluation (the "Bwd" column) is less than 1 second on 10/11 examples.
2) *Quick Class Structure Refactoring.* The running time of the Class Structure Refactoring varies from 1 ms to 4 ms.
3) *Acceptable Backward Response Time.* The response time of the system feedback after direct manipulations (the "B:Tot" column) is less than 2 seconds on 9/11 applications.

It is worthwhile indicating that the total running time is mainly dominated by the conversion of objects and HTML (except for the backward transformation of Tic Tac Toe), which is not carefully optimized in our tool implementation.

5.3 Threats to Validity

Regarding *internal* validity, the design of direct manipulations for each example affects the response time. For example, adding objects or changing a small object to a large one takes longer time than other manipulations in the translation from HTML to objects. To mitigate this threat, we test both value manipulations and structure manipulations (including adding objects and modifying types) in each example and measure the fastest and slowest response times for different manipulations. From Table 4, the largest difference between the fastest and slowest response times, 744 ms, is acceptable. Of course, in practice, some extreme cases, like copying entire pages, may significantly increase the total backward running time. In the future, we will complement this extreme case testing and investigate acceleration methods.

Regarding *external* validity, the first threat is that we only evaluate our approach based on Swing programs. To mitigate this issue, we should use GUI programs developed based on different languages and frameworks. It will be our future work to enhance our evaluation based on more GUI programs. However, Java Swing is a representative GUI framework containing all the main features shared by existing mature GUI frameworks, such as AWT, SWT, and JFace. Therefore, we believe that our conclusions (for RQ1 and RQ2) are generalizable even though other GUI programs based on different languages and frameworks are used.

The second threat to external validity is that the benchmark programs' sizes, the class table sizes, and the number of objects may be too small to represent real-life applications. Consequently, our answer to RQ2 may not be generalized to more complex programs. Ideally, we should select more complex GUI programs. However, our approach mainly applies to the design of a single webpage. In fact, according to the principle of modularity, complex GUIs should be decomposed into some simple sub-GUIs for design and implementation. Besides, the benchmarks come from the #1 ranked Java Swing tutorial on Google and a project with 790 stars on GitHub, which are all representative of Java Swing programs. Therefore, we believe that the benchmark programs fit the expected application scenario, and this threat does not compromise the validity of our conclusions for RQ2.

6 RELATED WORK

Our work on bidirectional object-oriented programming is much related to bidirectional programming, direct manipulation programming, and visual IDE for object-oriented programming.

6.1 Bidirectional Programming

Lenses [Foster et al. 2007] is a domain-specific bidirectional language to solve the *view update problem* for tree-structured data. They define a well-behavedness stipulating how the *get* (taking a source and returning a view) and *putback* (taking the original source and an updated view and returning an updated source) should behave. *Lenses* are studied for the synchronization of various data structures, including relational data [Bohannon et al. 2006], semi-structured data [Foster et al. 2007; Kawanaka and Hosoya 2006], strings [Barbosa et al. 2010; Bohannon et al. 2008], and models/graphs [He and Hu 2018; He et al. 2022; Hidaka et al. 2010]. However, all these bidirectional programming languages are functional. It is this paper that presents the first bidirectional object-oriented language with side effects. It can be effectively used for the bidirectional transformation between object-oriented programs and their outputs.

Among the work mentioned above, UnCAL [Hidaka et al. 2010] carefully refines the existing forward evaluation of structural recursion to produce sufficient trace information for later backward evaluation. This trace idea has inspired our work on the bidirectional transformation between objects and HTML in Section 4. It would also make it possible for us to extend the bidirectional transformation to deal with graph-structured object outputs in the future.

6.2 Direct Manipulation Programming

Direct Manipulation Programming (DMP), first proposed by [Chugh 2016], is to construct software systems that tightly couple programmatic and direct manipulation. DMP is studied in various domains: LittleLeo [Mayer et al. 2018] for web development; Sketch-n-Sketch [Hempel and Chugh 2016; Hempel et al. 2019] for SVG graphics design; CapStudio [Fukahori et al. 2014] for game application design; and FormsEdit [Avrahami et al. 1989] for graphical user interfaces. Our tool, BiOOP, can be considered a direct manipulation system for both dynamic web development and SVG graphics design.

Our work is based on LittleLeo [Mayer et al. 2018], which presents a *backward evaluation* algorithm for a functional programming language. The heart of the algorithm lies in a well-defined strategy to update variables and variable bindings, which inspired our backward evaluation on method invocations, where the method body is updated first, then the updates are propagated to the receiver and the passed parameters. There is a sharp difference from our work: they present the bidirectional evaluation for a functional language, whereas we present the bidirectional evaluation for an object-oriented language with class hierarchies and references.

Sketch-n-Sketch [Hempel and Chugh 2016; Hempel et al. 2019] provides a graphical editor for drawing shapes, relating shapes to each other, and grouping shapes together. For one thing, their approach is designed for functional programs, while our approach supports object-oriented programs. For another, their approach only supports certain kinds of direct manipulations on SVG graphics, while our approach can handle more general output, such as HTML pages. Of course, their work suggests that we can introduce more kinds of direct manipulations, such as grouping components, to our system by defining object-oriented code templates.

FormsEdit [Avrahami et al. 1989] provides two editable views, allowing developers to use text editing and direct graphics editing in any combination, rapidly switching from one view to the other and seeing the results in both. The language FormsVBT takes the form of symbolic expressions (s-expressions), while our work is based on a general object-oriented language. Their implementation maintains a parse tree as a shared data structure. The transformation between the Parse Tree and the Graphics View is similar to the second mapping (between the objects and HTML) in our two-stage bidirectionalization mentioned in Section 4. The main difference is that they implemented the transformation in a one-to-one mapping, but we allow users to customize the transformation templates, which is more flexible.

CapStudio [Fukahori et al. 2014] is a development environment for a visual application with an interactive screencast using Processing [Fry et al. 2001]. A screencast is a movie player-like output window with code editing functionality. They record the function calls and their parameters in the forward editing, which traces the corresponding position of the source program to modify in the backward editing. They only permit modifications to the function calls' parameters, unlike our approach, which permits modifications to arbitrary terms.

6.3 Visual IDE for Object-Oriented Programming

Many object-oriented language IDEs have visual design interface features, such as *Qt Creator* [Nord and Chambe-Eng 1995] for C++; Visualiser [Visualiser 2004] in Eclipse; JFormDesigner [JFormDesigner 2003] for Java Swing. These tools can update the code to a certain extent by directly manipulating the components on the visual interface. They only permit developers to alter a static screenshot that depends on fixed templates related to object-oriented programs, unlike our approach, which allows developers to manipulate the execution output directly to update the program.

Morphic [Maloney 1995] is the user interface framework for Self [Ungar and Smith 1987]. A primary goal of morphic is to make it easy to construct and edit interactive graphical objects, both by direct manipulation and from within programs. Morphic is relevant to the second stage of bidirectional transformations in our system, both of which establish a mapping between the object structure and the actual representation. In future work, we can even replace O-bx-H with a Morphic-like visual IDE for richer direct manipulations.

7 CONCLUSION

This paper presents the first bidirectional object-oriented language BiFJ, which supports developers not only to do text-based programming but also to manipulate objects in the output directly. Although we focuses on GUI design, our approach is domain-independent. Based on the bidirectional evaluation for pure functional languages, we present a bidirectional semantics for object-oriented programming with references. We tackle the challenges of bidirectional evaluation in the complex class inheritance hierarchies and the global read-and-write state. To deal with objects, we adopt *subclassing* when updating method invocations and optimize the class table to reduce class redundancy. To deal with references, we focus on the bidirectional semantics for reference creations, dereferences, and assignments and emphasize the updating order and effect rollback is crucial for

references. We prove that our bidirectional object-oriented evaluations satisfy the round-tripping properties when the output is altered consistently. To allow developers to manipulate the object structure directly, we design the two-stage bidirectionalization and develop the second bidirectional transformation between the object structure and the mapped HTML to guarantee successful reflection to the source program. Our experimental results show the expressiveness of our language and efficiency of our tool.

Our language is limited in its lack of polymorphism compared to practical object-oriented languages. In the future, we will incorporate a type system to investigate the impact of polymorphism on our existing bidirectional evaluation semantics.

ACKNOWLEDGEMENTS

The authors would like to thank anonymous reviewers for many helpful suggestions. This work was partly supported by the Beijing Natural Science Foundation (NO. 4192036) and the National Key Research and Development Program of China (No. 2021ZD0110202).

DATA-AVAILABILITY STATEMENT

Our tool BiOOP is an open-source project[Zhang et al. 2023], which contains all the source code and the benchmark programs.

REFERENCES

Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. 1989. A Two-View Approach to Constructing User Interfaces. *SIGGRAPH Comput. Graph.* 23, 3 (jul 1989), 137–146. https://doi.org/10.1145/74334.74347

Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. 2010. Matching Lenses: Alignment and View Update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/1863543.1863572

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 407–419. https://doi.org/10.1145/1328438.1328487

Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. 2006. Relational Lenses: A Language for Updatable Views. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Chicago, IL, USA) *(PODS '06)*. Association for Computing Machinery, New York, NY, USA, 338–347. https://doi.org/10.1145/1142351.1142399

Ravi Chugh. 2016. Prodirect Manipulation: Bidirectional Programming for the Masses *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 781–784. https://doi.org/10.1145/2889160.2889210

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

Robert Eckstein, Marc Loy, and Dave Wood. 1998. *Java Swing*. O'Reilly & Associates, Inc., USA.

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

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. 2012. *Three Complementary Approaches to Bidirectional Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–46. https://doi.org/10.1007/978-3-642-32202-0_1

Ben Fry, Casey Reas, Andres Colubri, Elie Zananiri, Samuel Pottinger, and Dan Shiffman. 2001. *Processing*. https://processing.org/

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

Xiao He and Zhenjiang Hu. 2018. Putback-Based Bidirectional Model Transformations. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*

(Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 434–444. https://doi.org/10.1145/3236024.3236070

Xiao He, Zhenjiang Hu, and Na Meng. 2022. A theoretic framework of bidirectional transformation between systems and models. *Sci China Inf Sci* 65, 202103 (2022). https://doi.org/10.1007/s11432-020-3276-5

Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Oct 2016). 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

Miško Hevery and Adam Abrons. 2010. *Angular*. https://angular.io/

Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. 2010. Bidirectionalizing Graph Transformations. *Sigplan Notices - SIGPLAN* 45, 205–216. https://doi.org/10.1145/1932681.1863573

Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. 2004. A Programmable Editor for Developing Structured Documents Based on Bidirectional Transformations *(PEPM '04)*. Association for Computing Machinery, New York, NY, USA, 178–189. https://doi.org/10.1145/1014007.1014025

JFormDesigner. 2003. *Eclipse, IntelliJ IDEA, NetBeans and JDeveloper*. https://www.formdev.com/jformdesigner/

Shinya Kawanaka and Haruo Hosoya. 2006. BiXid: A Bidirectional Transformation Language for XML. *SIGPLAN Not.* 41, 9 (sep 2006), 201–214. https://doi.org/10.1145/1160074.1159830

John Maloney. 1995. Morphic: The Self User Interface Framework. Mountain View, CA 94043 USA. https://ftp.squeak.org/docs/Self-4.0-UI-Framework.pdf

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

Keisuke Nakano, Zhenjiang Hu, and Masato Takeichi. 2008. Consistent Web site updating based on bidirectional transformation. In *2008 10th International Symposium on Web Site Evolution*. 45–54. https://doi.org/10.1109/WSE.2008.4655395

Haavard Nord and Eirik Chambe-Eng. 1995. *Qt Creator*. https://www.qt.io/product/development-tools

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. 2014. Lenses for Web Data. (01 2014). https://doi.org/10.14279/tuj.eceasst.57.879.869

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '87)*. Association for Computing Machinery, New York, NY, USA, 227–242. https://doi.org/10.1145/38765.38828

Visualiser. 2004. *Eclipse*. http://www.eclipse.org/ajdt/visualiser/

Evan You. 2014. *Vue.js*. https://vuejs.org/

Xing Zhang, Guanchen Guo, Xiao He, and Zhenjiang Hu. 2023. *Bidirectional Object-Oriented Programming: Towards Programmatic and Direct Manipulation of Objects(Artifact)*. https://doi.org/10.5281/zenodo.7698353

Received 2022-10-28; accepted 2023-02-25