Towards Bidirectional Live Programming for Incomplete Programs

Xing Zhang Zhenjiang Hu*
Key Laboratory of High Confidence Software Technologies, MoE
School of Computer Science, Peking University
zhangstar@stu.pku.edu.cn huzj@pku.edu.cn

ABSTRACT

Bidirectional live programming not only allows software developers to see continuous feedback on the output as they write the program, but also allows them to modify the program by directly manipulating the output, so that the modified program can get the output that was directly manipulated. Despite the appealing of existing bidirectional live programming systems, there is a big limitation: they cannot deal with incomplete programs where code blanks exist in the source programs.

In this paper, we propose a framework to support bidirectional live programming for incomplete programs, by extending the output value structure, introducing hole binding, and formally defining bidirectional evaluators that are well-behaved. To illustrate the usefulness of the framework, we realize the core bidirectional evaluations of incomplete programs in a tool called Bidirectional Preview. Our experimental results show that our extended backward evaluation for incomplete programs is as efficient as that for complete programs in that it is only 21*ms* slower on a program with 10 holes than that on its full program, and our extended forward evaluation makes no difference. Furthermore, we use *quick sort* and *student grades*, two nontrivial examples of incomplete programs, to demonstrate its usefulness in algorithm teaching and program debugging.

KEYWORDS

live programming, bidirectional evaluation, direct manipulation, hole bindings, hole closures

ACM Reference Format:
Xing Zhang Zhenjiang Hu*. 2022. Towards Bidirectional Live Programming for Incomplete Programs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3510003.3510195

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510195

1 INTRODUCTION

Software developers have psychological expectations[1] on the output when programming in many scenarios, particularly in the design of systems such as dynamic web pages, graphical user interfaces (GUIs), slide-based presentations, and data visualizations. To combine the intuitiveness of direct manipulation on the output with the abstractness and repeatability of text-based programming, researchers have developed many useful bidirectional live programming systems, such as Sketch-n-Sketch [11], Capstudio [7], and Carbide Alpha [9], which can not only allow developers to see continuous feedback on the output when they write programs, but also allow them to modify the program by directly manipulating the output so that the modified program can get the output that was directly manipulated.

Despite the appeal of the existing bidirectional live programming systems, there is a big limitation: they cannot deal with incomplete programs where code blanks exist in the source programs [13]. In practice, software developers tend to program in a way where they skip some parts by leaving some code blank here and there in the program during programming. Therefore, it would be practically useful if, even when the program is incomplete, developers could still directly manipulate the output and automatically synchronize the program with the updated output.

Fortunately, Omar et al. [14] have made a nice progress in this direction, showing that it is possible to do (unidirectional) live programming for incomplete programs. They model incomplete programs as expressions with holes, which denote missing expressions. Rather than aborting the evaluation when a hole is encountered, they track the evaluation state (also known as closure), i.e., variables with their bindings that can be accessed by the hole instances, allowing developers to observe this information in editor services.

With the technique of live programming for incomplete programs, what we need to do is to make this technique bidirectional to achieve the goal of bidirectional live programming for incomplete programs. Different from complete programs, the output of incomplete programs may consist of the output value and closures of holes, both of which should allow direct manipulation (modification).

To this end, we are facing three challenges: (1) In forward evaluation, since the output of incomplete programs is more complex than that of complete programs and it needs to be reflected back later, the output value structure needs to be carefully designed; (2) In backward evaluation, holes in the program are special and

[1]Psychological expectations refer to users' expectations of what the output looks like and what content the output displays. For example, in web development, developers often design prototypes in advance and have expectations about what components are included and where they are located.

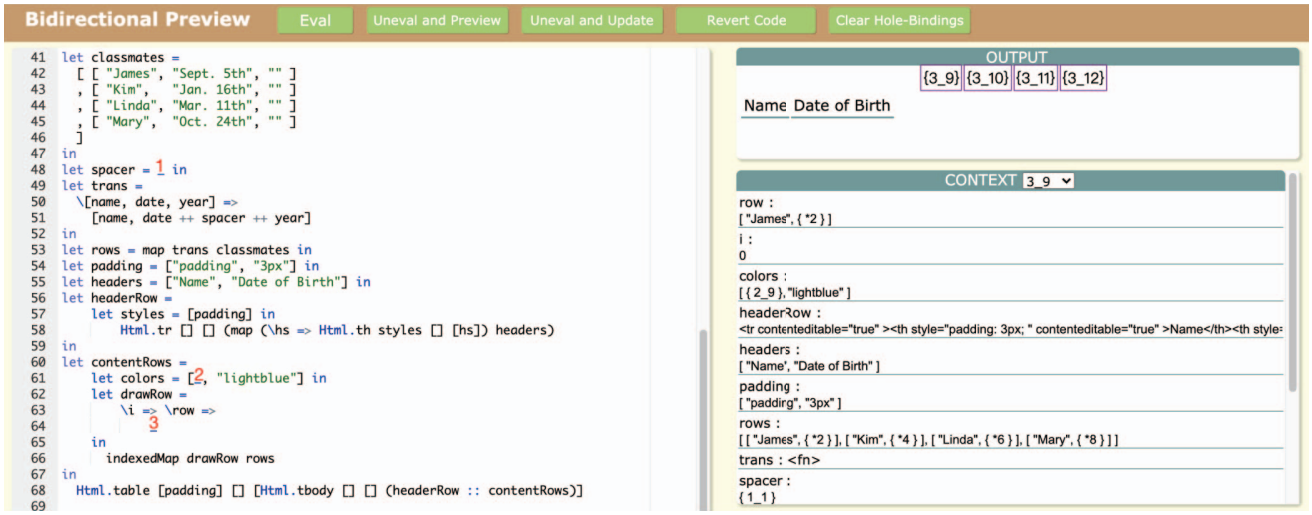

Figure 1: An incomplete program written in Bidirectional Preview generates an incomplete HTML table of classmate names and their date of birth. The language used is defined in Figure 9.

should not simply be treated as an undetermined value, because one hole may be referred many times in the forward evaluation and thus each hole may be associated with multiple values; (3) To guarantee the stability of bidirectional evaluation for incomplete programs, it is important to securely concatenate upstream and downstream backward evaluations that are hole-partitioned and satisfy the round-tripping properties.

To address challenge (1), we extend the definition of values with three types of holes, including instantiations of the source hole expressions, temporary holes generated when evaluation cannot continue, and sub holes generated after holes decomposition. To address challenge (2), we propose the notion of hole bindings, which record the value of a hole under a certain evaluation state. As part of the forward evaluation input, it will be updated in backward evaluation. To address challenge (3), we carefully design a pair of forward and backward evaluations and prove that they satisfy the round-tripping property. This implies that the local consistency can guarantee global consistency.

In this paper, we propose a new framework for bidirectional live programming for incomplete programs. The framework supports developers not only to write incomplete programs and observe the output with hole closures in editor services, but also to directly manipulate them to synchronize the program with the output. Our main technical contributions can be summarized as follows:

- We design a programming framework to support bidirectional live programming for incomplete programs so that developers can directly manipulate the output to synchronize the program with the updated output even when the program is not finished (Section 3). As far as we are aware, this is the first framework that can support bidirectional live programming for incomplete programs.
- We formalize a bidirectional evaluation for incomplete programs (Section 4), which successfully solves the three challenges of defining operable output values for incomplete programs, updating holes with constants, and updating programs with hole values. Besides, the round-tripping properties [6, 12] can be guaranteed (Section 4.4).
- We give an efficient implementation of the bidirectional live programming framework as a concrete tool called Bidirectional Preview, which is available at the public repository[1]. We use two nontrivial examples (Section 5) to demonstrate practical usefulness of our system in algorithm teaching and program debugging. Besides, our experimental results show that compared to bidirectional evaluation for complete programs, our backward evaluation is only 21*ms* slower on an incomplete program with 10 holes than on its complete program, and the forward evaluation is almost no different (Section 6).

2 OVERVIEW

In this section, we shall demonstrate how developers go through bidirectional live programming (forward text-based programming and backward direct manipulation) to accomplish a web development task. Consider the task of implementing an HTML table that displays each of your classmates along with their date of birth. Note that the HTML example is adapted from the baseline work on the complete program in Sketch-n-Sketch [11]. With our tool Bidirectional Preview, the developer may start with an incomplete program that generates an incomplete prototype.

2.1 Initial Incomplete Program

Figure 1 shows a screenshot of the system, which consists of an incomplete program (the auxiliary functions are omitted) on the left and the incomplete table it generates on the right. Lines 41 through 46 define the classmate data; each element is a three-element list,

[1] https://github.com/xingzhang-pku/BidirectionalPreview

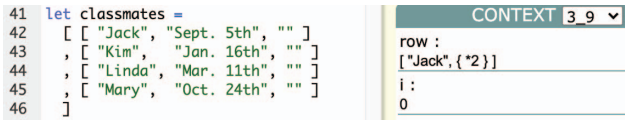

Figure 2: When modifying "James" to "Jack" in the first line of the Context, the classmate data in line 42 of the source program changes correspondingly.

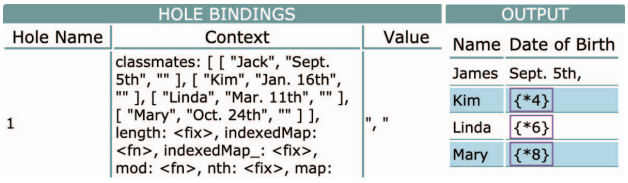

Figure 3: When modifying hole ∗2 in the first line of the HTML table to "Sept. 5th, ", the output value is propagated to the source program and h_1 is inferred to be ", ".

consisting of the name, the month and day of birth, and the year of birth (for now, the data is missing).

The main program consists of two parts: lines 48 through 53 define the data process, using a *spacer* to connect two parts of the date; and lines 54 through 68 define the HTML rendering of table headers and content. The program contains three holes (red numbers): the first one h_1 is the expression assigned to variable *spacer*; the second one h_2 is an uninitialized string-type constant in list *colors*, and the last one h_3 is the body of data rendering function *drawRow*.

Due to the missing code in *drawRow*, the output HTML table has its header but lacks content. On the top of the OUTPUT window, the four purple boxes denote four unknown HTML elements. Hole values are indicated by the hole name written in curly braces. In particular, holes 3_9 through 3_12 are four instantiations of h_3, and their closures (free variables and their values) can be switched via the selector in the CONTEXT window. The current CONTEXT window shows a partial closure of hole 3_9, including *row* being ["*James*", {∗2}], *i* being 0, etc, where {∗2} stands for a temporary hole value generated by string concatenation with h_1.

2.2 Direct Manipulation on Context

Although the program is unfinished, the developer can perform direct manipulation on the output and the context. The modification includes updating a value to another value and updating a hole to a value, or vice versa.

Figure 2 shows that when modifying the first element of *row* from "James" to "Jack" in the first line of the Context, the classmate data in Line 42 of the program is updated correspondingly. HTML values are presented as HTML strings in the Context like *headerRow* in Figure 1 and can also be modified.

2.3 Direct Manipulation on Output

The functionality of the missing code at h_3 is to color the even-numbered lines of the table lightblue and the odd-numbered lines

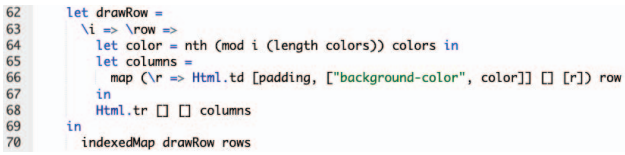

Figure 4: Fill in the missing code in h_3.

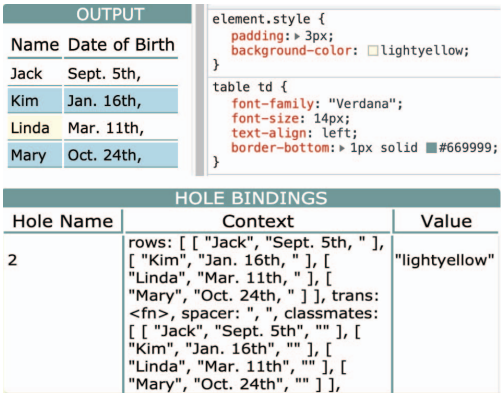

Figure 5: When using DOM Inspector to initialize the uncertain color of the cell with "Linda" on it, "lightyellow" is propagated to h_2 and insert a new piece of record to hole bindings.

the missing color at h_2. After the developer fills in h_3 as shown in Figure 4, the shape of the table is complete but the "Date of Birth" column is still incomplete because of h_1, where holes ∗2, ∗4, ∗6, and ∗8 display in purple boxes.

2.3.1 Modify Text in Output. As shown in the right of Figure 3, the developer updates hole ∗2 in the first row to "Sept. 5th, " which is in his/her expected date format. The modified HTML table is propagated to the source program and the system infers that the value of h_1 is ", ".

Without replacing h_1 locally, a new piece of record is added to the Hole Bindings as shown in the left of Figure 3, which means h_1 should be evaluated to ", ", under the context (not shown in full) including *classmates* and so on.

2.3.2 Modify Output Using DOM Inspector. Besides directly modify the text in output, developers can conveniently modify the output using developer tools (e.g. DOM Inspector) in the browser. As shown in Figure 5, when the developer is not sure of the desired color, he/she set the first element in *colors* to h_1. After the forward evaluation, due to the uninitialized color for odd-numbered lines of the HTML, the background color is default white. Then the developer uses the DOM inspector to select the cell "Linda" in the third row of the table, and sets its background-color property to "lightyellow" in the Styles Box. Through backward evaluation, the color value is propagated to h_2 and a new piece of record about h_2 is added to hole bindings as shown in the bottom of Figure 5.

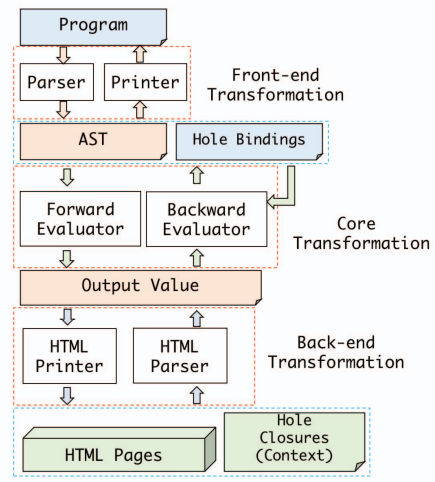

Figure 6: System Framework

```
letrec qsort =
    \ls =>
        case ls of
            [] => []
            | pivot :: xs =>
                let (smaller, bigger) = partition (\x=>x<pivot) xs in
                let (r_smaller, r_bigger) = (qsort smaller, qsort bigger) in
                r_smaller ++ _
in
    qsort [4,2,5,3,1]
```

Figure 7: Quick Sort

```
let students =
    [ [88.0, 89.0, 87.0]
    , [76.0, 93.0, 95.0]
    , [93.0, 79.0, 84.0]
    ]
in
let (hwr, midr, finr) =
    (30.0, 0.3, 0.4)
in
let weighted_average =
    \[hw, midterm, final] =>
        hwr * hw + midr * midterm + finr * final
in
    map weighted_average students
```

Figure 8: Student Grades

Summary

After the above series of manipulations, the developer accomplished the task of finishing the program with h_1 being " " and h_2 being "lightyellow", and h_3 being defined in Figure 4, which runs to get the expected HTML table. In summary, our tool is a direct manipulation system of incomplete programs with friendly interactions, being effective and productive.

3 SYSTEM DESIGN

Our framework for supporting bidirectional live programming for incomplete programs is shown in Figure 6. The down arrows represent the forward process of normal live programming, and the up arrows represent the backward process of modifying the program by directly manipulating the output. We use four colors to mark the four parts of the input, output, intermediate results, and the core implementation respectively.

The user input is a source program, which is written in a simple functional language (used in [11]) with the hole extension. The language used in our system is defined in Figure 9. The output that developers can direct manipulate consists of two parts: one is the visual object designed by the developer (we use HTML pages as an example, but it can also be slide-based presentations, data visualizations, etc.); the other is the closures related to hole values in visual objects, also known as the Context in Bidirectional Preview. Developers can modify the value of each variable in a hole closure in the Context, and if the value is a hole value, they can jump to its closure and modify it. The intermediate results in the system are the internal representation of the program-Abstract Syntax Tree (AST) and the internal representation of the output values.

The core part of the system is the content in white boxes, including three bidirectional transformations [4] denoted with red dotted boxes. The front-end bidirectional transformation maintains the consistency between the program and its AST, including the parser which parses the program to the AST, and the printer which prints the updated AST back to the program in the original format with the same white spaces. The back-end transformation maintains the consistency between the output value and the HTML pages with hole closures, including the HTML printer which prints the HTML pages and collects the hole closures, and the HTML parser which parses the HTML pages with hole closures to the output values.

The core bidirectional transformation is the most critical bidirectional evaluators, including the forward evaluation and backward evaluation. The forward evaluation takes the AST and hole bindings as input and returns the output values. The backward evaluation takes the updated output values and the original AST with hole bindings as input and returns the updated AST with hole bindings. The core bidirectional transformation maintains the consistency between the AST with hole bindings and the output values.

It should be noted that the parser and printer in the front-end and back-end bidirectional transformations are not special. The framework has been implemented in Bidirectional Preview , basically following the structure of Sketch-n-Sketch [11]. The most important contribution of this paper is the crucial bidirectional evaluation part, which will be explained in detail in Section 4.

4 THE CORE BIDIRECTIONAL EVALUATORS

In this section, we show how to tackle the most challenging part of the framework. As discussed in the introduction, we need to address three issues: defining operable output values for incomplete programs, updating holes with constants, and updating programs with hole values. In the following, we shall address these issues, by explaining the (source) programs to be developed, and defining evaluation rules for the forward evaluator and the backward evaluator in the framework of bidirectional live programming for incomplete programs.

4.1 Source Program

The source program is written in a simple functional language, almost the same as that in Sketch-n-Sketch [11] with additional holes. It is a functional language with holes to denote blank codes.

$$
\begin{aligned}
\text{Expressions e} &::= (||)^{u} \mid c \mid x \mid \lambda p.e \mid e_1\ e_2 \mid e_1 :: e_2 \mid (e_1,\ e_2) \\
&\quad \mid \text{let p } e_1\ e_2 \mid \text{letrec p } e_1\ e_2 \mid \\
&\quad \mid \text{if } e_1\ e_2\ e_3 \mid \text{case e } (p_1,\ e_1) \cdots \\
\text{Constants c} &::= n \mid b \mid s \mid [\,] \mid (+) \mid (*) \mid (\&\&) \mid not \\
\text{Patterns p} &::= c \mid x \mid p_1 :: p_2 \mid (p_1,\ p_2) \\
\text{Environment E} &::= \emptyset \mid E,\ x \mapsto v \\
\text{Values v} &::= (||)^{\text{u}}_{\text{E}} \mid \text{dv} \\
\text{Determinate Values dv} &::= \text{c} \mid (\text{E},\ \lambda\text{p.e}) \mid \text{v}_1 :: \text{v}_2 \mid (\text{v}_1,\ \text{v}_2) \\
\text{Hole Name u} &::= \text{n} \mid *\text{n} \mid \text{u\_n} \mid \text{u.n}
\end{aligned}
$$

Figure 9: Syntax

Figure 9 gives the syntax of the language. In particular, we define the output of an incomplete program suitable for manipulation, which addresses the first issue (i.e., the definition of operable output values) we mentioned.

In our language, expressions (i.e. programs) include hole expressions $(||)^{u}$ (u is the hole name), constants c, variable x, function application $e_1\ e_2$ (apply function e_1 to arguments e_2), list construction $e_1 :: e_2$ (append the list e_2 with the head element e_1), tuple $(e_1,\ e_2)$, let-bindings `let` $p\ e_1\ e_2$, letrec-bindings `letrec` $p\ e_1\ e_2$, conditionals `if` $e_1\ e_2\ e_3$, and case expressions `case` $e\ (p_1,\ e_1) \cdots$. The definition of constant includes numbers n, booleans b, strings s, the empty list `[]`, and primitive operators.

Simply put, a source program is an expression that has let-bindings combined. Each expression can only access variables defined in the outer let-bindings. We have seen an example in the overview, and there are two more example programs, i.e. *quick sort* shown in Figure 7 and *student grades* shown in Figure 8.

4.2 Forward Evaluator

The forward evaluator computes the value of an expression, which explains how the output we defined in the previous subsection is obtained. Figure 10 defines the evaluation rules (whose names are prefixed with "E-") of the forward evaluator, which is standard and similar to that in Sketch-n-Sketch [11], except for the evaluation rules involving holes. In this section, we mainly explain the evaluation of holes in hole expressions, function calls, primitive operations, conditionals, etc, where the three different types of holes are generated. The forward evaluation judgment $\Sigma; E \vdash e \Longrightarrow v$ states that "the expression e evaluates to v under the environment E and the hole bindings Σ."

4.2.1 Two Bindings. There are two bindings, environment E and hole bindings Σ, used in the forward evaluator. As shown in Figure 9, environment E is a variable-value mapping that denotes the evaluation state. $E, x \mapsto v$ denotes inserting the binding of x with v to E. Hole-bindings Σ maps pairs of hole names u and environments E to determinate values and its notation is the same as environments.

Example 4.1. Consider the expression `x+`$(||)^1$ under the environment $\{x \mapsto 1\}$ (E) and the hole bindings $\{(1, \{x \mapsto 1\}) \mapsto 2\}$ (Σ). The variable x evaluates to 1 according to E and $(||)^1$ under E evaluates to 2 according to Σ. □

In addition, supplementary definitions of values and hole names are as follows. Values v include determinate values dv and hole values $(||)^{u}_{E}$ where u means the hole name and E means the closure. The definition of hole closures is the same as the environment, which is a variable-value mapping. Determinate values include constants c, function closure $(E, \lambda p.e)$ where E binds free variables in the body of the function λp.e, list values $v_1 :: v_2$, and tuples (v_1, v_2). Hole names u includes numbers n representing source hole expressions, numbers beginning with an asterisk $*n$ representing intermediate temporary hole values, numbers joined with an underscore u_n representing hole instantiations, and numbers joined with a dot $u.n$ representing sub holes.

4.2.2 Hole instantiations. The hole instantiations are generated when evaluating the source hole expressions. E-Hole-2 is the same as the evaluation rule of hole expressions defined by Omar et al. [14]. It says that when the pair consisting of u and E is not found as a key in hole bindings Σ, the hole expression $(||)^{u}$ evaluates to a hole instantiation $(||)^{u\_n}_{E}$, where n means the nth instantiation of the hole expression u and the environment E as a closure attaches to it. There is no hole binding when the developer has just finished the initial incomplete program, so all hole expressions evaluate to hole instantiations.

E-Hole-1 is different from the rule in previous work [14]. It says that if hole expression name u binds a determinate value dv in hole bindings Σ under the environment E, it will evaluate to dv. For example, Figure 3 shows that h_1 binds the value ", " under the environment (containing *classmates*, etc). Therefore, h_1 assigned to *spacer* evaluates to ", ", and the date of birth of Kim evaluates to "Jan. 16th, ", rather than the hole value $*4$.

4.2.3 Sub Holes. Sub holes are new in our evaluation, and they are generated when hole values do pattern matching. Before a detailed explanation of that, we show an example first, where a hole decomposes to a list construction and matches with a list pattern to produce two sub holes.

Example 4.2. Consider the function call `(\[x].x)` $(||)^1$. The argument $(||)^1$ evaluates to $(||)^{1\_1}_{\emptyset}$ which decomposes to $(||)^{1\_1.1}_{\emptyset} :: (||)^{1\_1.2}_{\emptyset}$. In particular, $(||)^{1\_1.1}_{\emptyset}$ and $(||)^{1\_1.2}_{\emptyset}$ are sub holes of $(||)^{1}_{\emptyset}$, and match with $[x]$ (i.e., $x :: [\,]$) to returns $\{x \mapsto (||)^{1\_1.1}_{\emptyset}\}$, according to M-Cons (explained later). Therefore, the final result is $(||)^{1\_1.1}_{\emptyset}$. □

Pattern matching is the process of matching values with structural patterns and binding values with corresponding variables in the pattern. The two key axioms in matching rules are defined as follows. The matching judgement $match(p, v) = E$ states that "value v matches with pattern p and the result is the matched bindings E." The rule M-Const says that a hole value $(||)^{u}_{E}$ matches with any constant pattern c and no binding is generated. The rule M-Cons is used when the value v matches with the list pattern $p_1 :: p_2$. If the decomposition of v is $v_1 :: v_2$, v_1 matches with p_1 to produce E_1 and v_2 matches with p_2 to produce E_2, respectively. The matched bindings are the concatenation of E_1 and E_2.

$$\text{E-Const}\ \frac{}{\Sigma; E \vdash c \Rightarrow c} \qquad \text{E-Var}\ \frac{E(x) = v}{\Sigma; E \vdash x \Rightarrow v}$$

$$\text{E-Hole-1}\ \frac{\Sigma(u, E) = dv}{\Sigma; E \vdash (||)^u \Rightarrow dv} \qquad \text{E-Hole-2}\ \frac{(u, E) \notin dom(\Sigma) \quad n\ is\ fresh}{\Sigma; E \vdash (||)^u \Rightarrow (||)_E^{u\_n}}$$

$$\text{E-Fun}\ \frac{}{\Sigma; E \vdash \lambda p.e \Rightarrow (E, \lambda p.e)} \qquad \text{E-App}\ \frac{\Sigma; E \vdash e_1 \Rightarrow (E_f, \lambda p.e_f) \quad \Sigma; E \vdash e_2 \Rightarrow v_2 \quad E_m = match(p, v_2) \quad \Sigma; E_m \circ E_f \vdash e_f \Rightarrow v}{\Sigma; E \vdash e_1\ e_2 \Rightarrow v}$$

$$\text{E-Case-1}\ \frac{\Sigma; E \vdash x \Rightarrow (||)_E^u \quad n\ is\ fresh}{\Sigma; E \vdash case\ x\ (p_i \to e_i)^{i=1..n} \Rightarrow (||)_E^{*n}} \qquad \text{E-Case-2}\ \frac{\Sigma; E \vdash x \Rightarrow dv \quad \exists j. E_m = match(p_j, dv) \quad \Sigma; E_m \circ E \vdash e_j \Rightarrow v}{\Sigma; E \vdash case\ x\ (p_i \to e_i)^{i=1..n} \Rightarrow v}$$

$$\text{E-Fix}\ \frac{\Sigma; E \vdash e\ (fix\ e) \Rightarrow v}{\Sigma; E \vdash fix\ e \Rightarrow v} \qquad \text{E-Plus}\ \frac{\Sigma; E \vdash e_1 \Rightarrow (||)_E^{u_1} \quad or \quad \Sigma; E \vdash e_2 \Rightarrow (||)_E^{u_2} \quad n\ is\ fresh}{\Sigma; E \vdash e_1 + e_2 \Rightarrow (||)_E^{*n}}$$

Figure 10: Evaluation Rules of Forward Evaluator

$$\text{M-Const}\ \frac{}{match(c, (||)_E^u) = \emptyset} \qquad \text{M-Cons}\ \frac{v \blacktriangleright v_1 :: v_2 \quad match(p_1, v_1) = E_1 \quad match(p_2, v_2) = E_2}{match(p_1 :: p_2, v) = E_1 \circ E_2}$$

The decomposition relation is defined using symbol $\blacktriangleright$, and $v_1 \blacktriangleright v_2$ states that "v_1 decomposes to v_2." The determinate values decompose to themselves, such as $[1, 2] \blacktriangleright [1, 2]$. The hole value decomposes to a list construction with two sub holes, written as $(||)_E^u \blacktriangleright (||)_E^{u.1} :: (||)_E^{u.2}$.

4.2.4 Temporary Holes. To make the forward evaluator continue the subsequent evaluation when it's stuck because of holes, we use temporary holes as result, rather than proceeding around holes [14]. In particular, temporary holes are generated in the evaluation of case expressions (E-Case-1) and primitive expressions (E-Plus). It is important to note that guard expressions in case expressions only consider the situation of a single variable; more complex situations can be achieved through syntactic sugar of function calls.

The rule E-Case-1 says that the case expression evaluates to a temporary hole value $(||)_E^{*n}$ when x evaluates to a hole value. The closure of the temporary hole value is the environment E, and n does not conflict with the names of the source hole expressions. The rule E-Plus is similar to E-Case-1, which stops the forward evaluation and returns a temporary hole value when encounters that e_1 or e_2 evaluates to a hole value.

4.3 Backward Evaluator

In this subsection, we explain how to address the second and third issues we mentioned at the beginning of this section, i.e., how to update hole expressions, and how to update incomplete programs with hole values in the backward evaluator.

The backward evaluator is the most important part of our approach. It takes the original internal-representation AST of the program with the hole bindings and the updated output values as the input and returns the updated AST and hole bindings. The backward evaluation judgment $\Sigma; E \vdash e \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash e'$ states that "when the output value updates to v', the program e updates to e', the environment E updates to E', and the hole bindings Σ updates to Σ'".

The evaluation rules (whose names are prefixed with "U-") of the backward evaluator are defined in Figure 11. The backward evaluation rules come in three categories: replacement rules overwrite values (base constants, function closures) in the program with new ones and update bindings of hole expressions; propagation rules, as the opposite of their corresponding forward rules, propagate the updated output to the whole evaluation process (through variables, applications, conditionals, etc); primitive rules define how to update operations on values, and the evaluation policies are heuristic and can be customized by domain experts.

4.3.1 Replacement Rules. Replacement rules define what can be overwritten in the source program and how to update the hole bindings. There are three axioms for holes, constants, and function closures, respectively.

Hole Expressions. The rule U-Hole-1 defines how a determinate value updates the hole expression, like Example 4.3.

Example 4.3. Consider the program `let a = 1 in` $(||)^1$. When the output value $(||)_{\{a \mapsto 1\}}^{1\_1}$ updates to 2, rather than rewriting the program to `let a = 1 in 2`, the hole binding which binds pair $(1, a \mapsto 1)$ with 2 is added to the hole bindings while the program remains unchanged. The hole binding means that the hole expression $(||)^1$ evaluates to 2 when variable a evaluates to 1. In the forward evaluation with the updated hole bindings, the program evaluates to 2. □

The rule U-Hole-1 says that, if a determinate value dv updates a hole expression u, then the binding of pair (u, E) with dv is inserted into the hole bindings, or the original binding of (u, E) is updated, while the program remains unchanged. The rule U-Hole-2 defines how a hole value updates the hole expression, like Example 4.4

Example 4.4. Consider the program in Example 4.3. If the output value is a hole value $(||)_{\{a \mapsto 2\}}^{1\_1}$, the updated hole closure $\{a \mapsto 2\}$ will propagate to the variable a through U-Fun. Therefore, the program updates to `let a = 2 in` $(||)^1$. □

The rule U-Hole-2 says that, if the output value is a hole value $(||)_{E'}^{u'}$, the domain of the updated hole closure is checked to see if

$$\text{U-Hole-1}\ \frac{\Sigma' = \Sigma, (u, E) \mapsto dv}{\Sigma; E \vdash (||)^u \Leftarrow dv \rightsquigarrow \Sigma'; E \vdash (||)^u} \qquad \text{U-Hole-2}\ \frac{dom(E) = dom(E')}{\Sigma; E \vdash (||)^u \Leftarrow (||)^{u'}_{E'} \rightsquigarrow \Sigma; E' \vdash (||)^u}$$

$$\text{U-Const-1}\ \frac{}{\Sigma; E \vdash c \Leftarrow c' \rightsquigarrow \Sigma; E \vdash c'} \qquad \text{U-Const-2}\ \frac{dom(E) = dom(E') \quad u_2\ is\ fresh}{\Sigma; E \vdash c \Leftarrow (||)^{u_1}_{E'} \rightsquigarrow \Sigma; E \vdash (||)^{u_2}}$$

$$\text{U-Plus-1}\ \frac{\Sigma; E \vdash e_1 \Rightarrow (||)^u_E \quad \Sigma; E \vdash e_2 \Rightarrow dv_2 \quad \Sigma; E \vdash e_1 \Leftarrow dv' - dv_2 \rightsquigarrow \Sigma_1; E_1 \vdash e_1'}{\Sigma; E \vdash e_1 + e_2 \Leftarrow dv' \rightsquigarrow \Sigma_1; E_1 \vdash e_1' + e_2} \qquad \text{U-Plus-2}\ \frac{dom(E) = dom(E') \quad \Sigma; E \vdash e_1 \Rightarrow (||)^{u_1}_E \quad or \quad \Sigma; E \vdash e_2 \Rightarrow (||)^{u_2}_E}{\Sigma; E \vdash e_1 + e_2 \Leftarrow (||)^u_{E'} \rightsquigarrow \Sigma; E' \vdash e_1 + e_2}$$

$$\text{U-Cons}\ \frac{v' \blacktriangleright v_1' :: v_2' \quad \Sigma; E \vdash e_1 \Leftarrow v_1' \rightsquigarrow \Sigma_1; E_1 \vdash e_1' \quad E' = E_1 \oplus_E E_2 \quad \Sigma; E \vdash e_2 \Leftarrow v_2' \rightsquigarrow \Sigma_2; E_2 \vdash e_2' \quad \Sigma' = \Sigma_1 \oplus_\Sigma \Sigma_2}{\Sigma; E \vdash e_1 :: e_2 \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash e_1' :: e_2'} \qquad \text{U-Fix}\ \frac{\Sigma; E \vdash e\ (fix\ e) \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash e_1\ (fix\ e_2) \quad e' = e_1 \oplus_e e_2}{\Sigma; E \vdash fix\ e \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash fix\ e'}$$

$$\text{U-Case}\ \frac{\Sigma; E \vdash x \Rightarrow (||)^u_E \quad \exists j\,.\, E_m = match(p_j, (||)^u_E) \quad \Sigma; E_m \circ E \vdash e_j \Leftarrow v' \rightsquigarrow \Sigma'; E_m' \circ E' \vdash e_j' \quad \emptyset; E_m' \vdash p_j \Rightarrow v_j \quad E'' = E', x \mapsto v_j}{\Sigma; E \vdash case\ x\ (p_i \to e_i)^{i=1..n} \Leftarrow v' \rightsquigarrow \Sigma'; E'' \vdash case\ x\ (p_i \to e_i)^{i \neq j} | (p_j \to e_j')}$$

$$\text{U-App}\ \frac{\Sigma; E \vdash e_1 \Rightarrow (E_f, \lambda p.e_f) \quad \Sigma; E \vdash e_2 \Rightarrow v_2 \quad E_m = match(p, v_2) \quad \Sigma; E_m \circ E_f \vdash e_f \Leftarrow v' \rightsquigarrow \Sigma_f; E_m' \circ E_f' \vdash e_f' \quad \Sigma; E \vdash e_1 \Leftarrow (E_f', \lambda p.e_f') \rightsquigarrow \Sigma; E_1 \vdash e_1' \quad \emptyset; E_m' \vdash p \Rightarrow v_2' \quad \Sigma; E \vdash e_2 \Leftarrow v_2' \rightsquigarrow \Sigma_2; E_2 \vdash e_2' \quad E' = E_1 \oplus_E E_2 \quad \Sigma' = \Sigma_2 \cup \Sigma_f}{\Sigma; E \vdash e_1\ e_2 \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash e_1'\ e_2'}$$

Figure 11: Evaluation Rules of Backward Evaluator

it's equal to the domain of the environment E. If the domains are equal, U-Hole-2 replaces E with E' directly. This is a key step for the updated hole closure to affect updating the source program. As shown in Example 4.4, the updated hole closure has the same domain as that of the original closure, which is $\{a\}$. In fact, domain consistency is an overly strict condition. If the domain of E' covers E, there is no bad effect on backward evaluation. However, if the variables in E are missing in E', it will cause a breakdown in backward evaluation. To simplify the problem in Bidirectional Preview, there is no interface for adding or deleting bindings in Context, so the domain of the environment is always consistent.

Constants. The backward evaluation rules of constants need to consider the types of output values. The rule U-Const-1 says that, when the output value c updates to c', the expression c also updates to c'. For example, when the output of the expression 1 updates to 2, the expression updates to 2. The rule U-Const-2 defines how a hole value updates the constant expression, like Example 4.5.

Example 4.5. Consider the program `let a = 1 in a`. If the output value 1 updates to a hole value $(||)^{1\_1}_{\emptyset}$, the program will update to `let a =` $(||)^1$ `in a`. □

The rule U-Const-2 says that when a hole value $(||)^{u_1}_{E'}$ updates the constant expression c, c will be replaced with a hole expression u_2 while the environment E remains unchanged. The hole name u_1 and hole closure E' are ignored because the hole value is generated in the output without defining a closure. u_2 is a fresh hole name for the new hole expression which is different from the names of existing holes. The rule U-Const-2 is the only way to add new hole expressions to the program by modifying the output. When a hole value updates the determinate value in the output, the hole value will flow through the entire evaluation process and only change the final constant in the program, like Example 4.5.

Function Closures. The function rule U-Fun (not shown) says that when the output value $(E', \lambda p.e')$ updates the expression $\lambda p.e$ under E and Σ, $\lambda p.e$ updates to $\lambda p.e'$ and E updates to E' while Σ remains unchanged. U-Fun is important to propagate the changes in the environment to sub derivations.

4.3.2 Propagation Rules. Propagation rules define how the output changes flow throughout the whole derivation. There are three axioms for variables, function calls, and conditionals, respectively.

Function Calls. The rule U-App defines how to update function calls, which follows the main idea in Sketch-n-Sketch [11] except for the hole bindings.

There are four steps in U-App: (1) e_1 evaluates to the function closure $(E_f, \lambda p.e_f)$ while e_2 evaluates to v_2, then the pattern matching between v_2 and p returns the matched bindings E_m; (2) The output value v' updates the function-body expression e_f under the concatenation of E_m and E_f with the original hole bindings Σ, then e_f updates to e_f', Σ updates to Σ_f, and $E_m \circ E_f$ updates to $E_m' \circ E_f'$; (3) The function closure $(E_f', \lambda p.e_f')$ updates e_1 under E and Σ, then e_1 updates to e_1', E updates to E_1, and Σ remains unchanged, while the updated output value v_2' updates e_2 and the result is that e_2 updates to e_2', E updates to E_2 and Σ updates to Σ_2; (4) Merging E_1 and E_2 returns E', while merging Σ_2 and Σ_f returns Σ'. Example 4.6 shows the four steps in practice.

Example 4.6. Consider the function call `(\x. [x, a]) a` with the environment $\{a \mapsto 1\}$ and the hole bindings $\emptyset$. Its output is $[1, 1]$ and then updates to $[2, 1]$. In step (1), e_1 evaluates to $(\{a \mapsto 1\}, \lambda x.[x, a])$ while e_2 evaluates to 1 and E_m is $\{x \mapsto 1\}$. In step (2), e_f' is the same as e_f, i.e., `[x,a]`. $E_m' \circ E_f'$ is $\{x \mapsto 2, a \mapsto 1\}$ and Σ_f is empty. In step (3), e_1 remains unchanged. The output value 2 updates $\Sigma; E \vdash e_2$ to $\emptyset; \{a \mapsto 2\} \vdash a$. Step (4) reconciles E_1

($\{a \mapsto 1\}$) and E_2 ($\{a \mapsto 2\}$) with the original E ($\{a \mapsto 1\}$) and returns $\{a \mapsto 2\}$ because a in E_2 is different with the one in E. □

Environments merge and hole bindings merge are two key steps to reconcile conflicts between two sub derivations of e_1 and e_2. As for environments merge, if a variable appears free in both expressions, the traditional two-way merge requires that it is updated with the same new value in both environments. Since two-way merge is too restrictive to make the backward evaluation work in most scenarios, we adapt the three-way merge instead, which is explained in detail in Sketch-n-Sketch [11]. The three-way merge is able to handle conflicts between E_1 and E_2 smoothly because it selects the value of the variable from E_1 or E_2 that is different from that in E like the environments merge in Example 4.6. Abstractly, the three-way merge implies that the updated value has a higher priority to be used than the original value.

As for hole bindings merge, it simply unions two hole-bindings directly, as shown in Example 4.7. The reason is that the updated or inserted hole bindings in Σ_2 and Σ_f must be different because closures of holes in e_2 and e_f must be different. The updated closures of holes in e_f have the same domain as that of $E_m \circ E_f$, however, in e_2 the updated closures of holes have the same domain as that of E.

Example 4.7. Consider the function call `(\x.[x, (||)¹ ]) (||)²`. Its output is $[\ (||)^{2\_1}_{\emptyset},\ (||)^{1\_1}_{\{x \mapsto (||)^{2\_1}_{\emptyset}\}}\ ]$ and then updates to $[1, 2]$. In step (2), $E'_m \circ E'_f$ is $\{x \mapsto 1\}$ and Σ_f is $\{(1, \{x \mapsto (||)^{2\_1}_{\emptyset}\}) \mapsto 2\}$. In step (3), Σ_2 is $\{(2, \emptyset) \mapsto 1\}$. According to the hole bindings merge, Σ' is the union of Σ_f and Σ_2. □

Case Expressions. The rule U-Case defines how to update a case expression of which the guard expression evaluates to a hole value, like Example 4.8.

Example 4.8. Consider the expression `case a of [x]->1|[]->2` under the environment $\{a \mapsto (||)^{1\_1}_{\emptyset}\}$. Its output is $(||)^{*1}$ (through E-Case-1) and then updates to 3. At first, a evaluates to $(||)^{1\_1}_{\emptyset}$ and tries to match with the first branch, i.e., `[x]->1`. Then the output value 3 successfully updates the branch expression 1 to 3 (through U-Const-1). Finally, the expression updates to `case a of [x]->3|[]->2` with the environment and hole bindings unchanged. Otherwise, if the output value fails to update the first branch, the backward evaluator will try the second branch `[]->2`. □

The rule U-Case says that when x evaluates to a hole value, the hole value tries to match with each pattern and the output value tries to update the corresponding branch. If the update success, the updated value v_j replaces the original binding of x in E'', and the updated branch expression e'_j replaces e_j.

Variables. The rule U-Var is omitted and it just replaces the original binding in the environment with the output value and the expression remains unchanged. For example, when the output value updates to 2, $\emptyset; \{x \mapsto 1\} \vdash x$ updates to $\emptyset; \{x \mapsto 2\} \vdash x$.

4.3.3 Primitive Rules. There are many strategies to update primitive operations, which can be tailored to different domains and problems. Suppose the strategy is that the effect of backward evaluation on the source program is as small as possible. Here we use plus operation and list construction as examples.

Plus. In the rule U-Plus-1, instead of updating e_2 that originally evaluates to a determinate value dv_2, the difference $dv' - dv_2$ as the output value updates e_1, which originally evaluates to a hole value.

Example 4.9. Consider the plus expression $(||)^1$+1. Its output value is $(||)^{*1}_{\emptyset}$ (through E-Plus) and then updates to 2. According to U-Plus-1, e_2 remains unchanged and the difference $2 - 1$ updates e_1. Through U-Hole-1, the backward evaluator returns $\{(1, \emptyset) \mapsto 1\}; \emptyset \vdash (||)^1$. □

The rule U-Plus-2 says that when the output value is a hole value $(||)^u_{E'}$ and e_1 or e_2 evaluates to a hole, the environment E is replaced by the hole closure E'.

There are many backward evaluation rules for other situations (such as both e_1 and e_2 evaluate to hole values) and the update strategies are somehow subjective, so they are omitted. In our implementation, each situation is handled only by a unique backward evaluation rule, so there is only a unique solution for the whole backward evaluation and no ambiguity. It is a better choice to provide multiple options like Sketch-n-Sketch [11], however since this is not what we focus on, we didn't implement it in our tool.

List Construction. The rule U-Cons defines how to update a list construction, like Example 4.10.

Example 4.10. Consider the expression `[a,a]` under the environment $\{a \mapsto (||)^{1\_1}_{\emptyset}\}$ and the hole bindings $\{(1, \emptyset) \mapsto 1\}$. Its output is $[1, 1]$ (through E-Hole-1) and then updates to $[1, 2]$. The head of the output value 1 updates the head of the list construction a to $\{(1, \emptyset) \mapsto 1\}; \{a \mapsto (||)^{1\_1}_{\emptyset}\} \vdash a$. The tail of the output value $[2]$ updates the tail of the list construction $[a]$ to $\{(1, \emptyset) \mapsto 2\}; \{a \mapsto (||)^{1\_1}_{\emptyset}\} \vdash [a]$. According to the three-way merge, the hole bindings update to $\{(1, \emptyset) \mapsto 2\}$. □

The rule U-Cons says that the output value v' should decompose to two sub-values, which update the head expression and tail expression respectively. The environments merge is the same as that in U-App, while the hole bindings merge is different. This is because there may be conflicts in Σ_1 and Σ_2. We also adopt the principle that the updated value has a higher priority, so the rule U-Cons apply three-way merge on hole bindings. The values bound to holes in Σ_1 or Σ_2 that is different from that in Σ is selected to be inserted into the updated hole bindings Σ'.

4.3.4 Recursion. Since recursion is not discussed in Sketch-n-Sketch [11], here is an explanation of our approach. Although recursion is not relevant with holes, it is essential to the expressiveness of a language.

The U-Fix rule propagates the output value v' to the expanded expression $e\ (fix\ e)$ which updates to $e_1\ (fix\ e_2)$. If e_1 is different from e, then e_1 is selected, otherwise, e_2 is selected.

4.4 Round-Tripping Properties

The forward evaluation and the backward evaluation form a lens [6] to maintain consistency between *source* (AST of programs with hole bindings) and *view* (output with hole closures). To ensure the stability of the system, the relationship of the forward evaluation and the backward evaluation should satisfy round-tripping properties, i.e., GetPut and WeakPutGet [12] defined as follows.

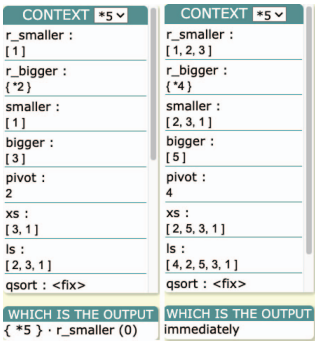

Figure 12: The student modifies *r_smaller* in hole $*3$ to $[1]$ and *r_smaller* in hole $*5$ closure to $[1, 2, 3]$.

Lemma 4.11 (GetPut). If $\Sigma; E \vdash e \Rightarrow v$ and $\Sigma; E \vdash e \Leftarrow v \rightsquigarrow \Sigma'; E \vdash e$, then $\Sigma'; E \vdash e \Rightarrow v$ and $\Sigma \subseteq \Sigma'$.

Lemma 4.12 (WeakPutGet). If $\Sigma; E \vdash e \Leftarrow v' \rightsquigarrow \Sigma'; E' \vdash e'$ and $\Sigma'; E' \vdash e' \Rightarrow v''$, then $\Sigma; E \vdash e \Leftarrow v'' \rightsquigarrow \Sigma'; E' \vdash e'$.

The GetPut Lemma requires that whenever the output has not been updated, the same program and hole bindings as the original one are obtained by backward evaluation. However, things are a little different for the bidirectional evaluators in our approach. A hole with a data structure decomposes to sub holes when it does pattern matching in the forward evaluation, and it binds with the data structure in hole bindings in the backward evaluation. Therefore, Σ is a sub-set of the updated hole bindings Σ'. For example, when hole $(||)^u_E$ matches with $x :: xs$, the binding of $(||)^u_E$ with $(||)^{u.1}_E :: (||)^{u.2}_E$ is inserted into hole bindings. It's easy to prove that the inserted hole bindings does not affect the forward evaluation, i.e., expression e under E and Σ' still evaluates to v.

The WeakPutGet Lemma requires that: (1) the output value v' updates the source program $\Sigma; E \vdash e$ to $\Sigma'; E' \vdash e'$; (2) e' under E' and Σ' evaluates to v''; (3) v'' updates $\Sigma; E \vdash e$ to the same updated program $\Sigma'; E' \vdash e'$. This property is important because it ensures that v' and v'' update the program to get the same result. We prove that the bidirectional evaluations with the three-way merge satisfies WeakPutGet when considering that there are no primitive rules and the control flow remains unchanged in the backward evaluation.

5 TWO APPLICATIONS

In this section, we use *quick sort* and *student grades*, two nontrivial examples, to illustrate the usefulness of our system in algorithm teaching and program debugging.

5.1 Quick Sort

Suppose that we are teaching the quick sort algorithm. An incomplete program is given in Figure 7, where we define a recursive function *qsort*; the auxiliary function *partition* is used to screen out *smaller* and *bigger* with pivot as the boundary, and *qsort* is recursively applied to *smaller* and *bigger* and get *r_smaller* and *b_smaller*, respectively. Now assume that we are not sure how to produce the final result and we add a hole in Line 23.

HOLE BINDINGS

Hole Name	Context	Value
1	r_smaller: [] , r_bigger: [] , smaller: [] , bigger: [] , pivot: 1 , xs: [] , ls: [1] , qsort: <fix> , smaller: [1] , bigger: [3] , pivot: 2 , xs: [3, 1] , ls: [2, 3, 1] , qsort: <fix> , smaller: [2, 3, 1] , bigger: [5] , pivot: 4 , xs: [2, 5, 3, 1] , ls: [4, 2, 5, 3, 1] , qsort: <fix> , qsort: <fix> , partition: <fix>	[1]
1	r_smaller: [1] , r_bigger: { *1 } , smaller: [1] , bigger: [3] , pivot: 2 , xs: [3, 1] , ls: [2, 3, 1] , qsort: <fix> , smaller: [2, 3, 1] , bigger: [5] , pivot: 4 , xs: [2, 5, 3, 1] , ls: [4, 2, 5, 3, 1] , qsort: <fix> , qsort: <fix> , partition: <fix>	[2, 3]

Figure 13: The bindings of hole 1 in quick sort.

Our system provides two convenient ways for students to understand the behavior of programs. First, they can "see". Students can observe the context of each recursive call of quick sort through the Context window. Second, they can "edit". Students can edit the "Context", changing the unknown value to the one they expect. For example, when students tell the system that sorting the list $[1]$ should yield $[1]$ (which is initially unknown), then the system will show what the value should be for the hole in the context, which helps them to guess what to fill in the hole.

Suppose the student runs the incomplete quick sort program and gets the Context table on the right in Figure 12. Students may wish to view the nested hole (e.g. *3) of the outer hole (e.g. *5), so they could click the subitem and jump to its context. At this time, the Context table displays the content on the left in Figure 12. "WHICH IS THE OUTPUT" in UI shows the click path "{*5}·r_smaller(0)". Note that the selector in the "Context" header is designed to show the hole names in the "Output" of the program, so both table headers in Figure 12 show *5. Next, it is easy for students to understand that the function *qsort* applied to *smaller* list $[1]$ results in $[1]$, so he/she modifies *r_smaller* to $[1]$ in the Context like Figure 12. In the same way back to the closure of the outermost hole $\{*5\}$, students know that *r_smaller* should be $[1, 2, 3]$ and modify the Context again like the right context in Figure 12.

After students fill in the hole in the Context with the results they expected, the hole-bindings in Figure 13 are generated through backward evaluation. With the second hole-binding, students know that when *pivot* is 2 and *r_smaller* is $[1]$, the value of hole expression 1 should be $[2, 3]$. Through observing the context of the second hole binding, he/she might have reasoned that the hole should actually be the connection of $[pivot]$ and *r_bigger*.

It is remarked that, for a hole in the program, our system provides developers only with the value of the hole and its context (as part of the output for one to modify); it does not infer the code for the hole, though developers may benefit from the value and its context to guess a possible code shape.

5.2 Student Grades

Consider that a teacher computes the students' final grades at the end of the semester. Suppose there are only three students, and the teacher wants to score according to the weighted average of homework, midterm, and final. He/she writes the program in Figure 8, where the function *weighted_average* applies to each student's score in Line 21.

Suppose that the program runs to get an incorrect output, say $[2701.5, 2345.9, 847.3]$. The teacher does not know which part of the expression in Line 19 was wrong, and he/she doubts the value at $hwr * hw$. Therefore, he/she changes the expression in Line 19 to $\_ + midr * midterm + finr * final$ and propagates the wrong

HOLE BINDINGS

Hole Name	Context	Value
1	hw: 93 , midterm: 79 , final: 84 , hwr: 30 , midr: 0.3 , finr: 0.4 , students: [[88, 89, 87], [76, 93, 95], [93, 79, 84]] , map: <fix>	2790
1	hw: 76 , midterm: 93 , final: 95 , hwr: 30 , midr: 0.3 , finr: 0.4 , students: [[88, 89, 87], [76, 93, 95], [93, 79, 84]] , map: <fix>	2280
1	hw: 88 , midterm: 89 , final: 87 , hwr: 30 , midr: 0.3 , finr: 0.4 , students: [[88, 89, 87], [76, 93, 95], [93, 79, 84]] , map: <fix>	2640

Figure 14: Hole-bindings of using [2701.5, 2345.9, 2847.3] **to update the program with a hole expression**

output through backward evaluation to get the records in the Hole bindings window (as shown in Figure 14). It is easy to know that the value (2790, 2280, and 2640) of the hole expression 1 (which is $hwr * hw$ originally) is wrong. Then the teacher observes the value of hwr and hw in the context of hole bindings and finds that the value of hwr should be 0.3, rather than 30.

Note that the purpose of turning a suspected expression into a hole is to observe its context and value to be sure that they are what we want for this expression. The example just shows the mechanism of using our work for debugging; in fact, the teacher may choose to convert any expression into a hole, not just $hwr * hw$.

Summary. These two examples and the example in Section 2 demonstrate the potential of our work in three areas: web development, classic algorithms, and program debugging. These examples are nontrivial and cover sufficient language constructs such as conditionals, function calls, and recursions (similar to loops).

6 EFFICIENCY

We have implemented the core bidirectional evaluators based on the framework of bidirectional live programming for incomplete programs in our tool Bidirectional Preview. Our implementation is written mostly in Elm, a functional language, with about 6,000 lines of code.

To evaluate whether our framework is effective enough to give the developers quick feedback after directly manipulating the output, we measured the running time (in milliseconds) of the various parts of the framework on the nontrivial example "table of states" (used in Sketch-n-Sketch [11]). The selected program has 76 lines of codes written in the language we defined. Note that in practice, an incomplete program is usually not long, particularly in the live programming setting. We've instrumented a timing mechanism in our implementation, which can record running times for each part of our implementation (such as times for parsing, forward and backward evaluations) when our system is running.

We use different numbers of hole expressions to indicate the degree of incompleteness of the same program. The more holes there are, the more incomplete the program is. The holes were introduced into some entries of the input table for the selected program, which can be propagated to various places of the program during evaluation. The experiments include four settings with 0, 1, 5, and 10 holes. We test up to 10 holes because there are generally not many holes in an incomplete program at one time in practical programming.

Table 1 shows a summary of our results averaged over 10 trials. We confirm that the fluctuation range of most of the results is within 5%. The "HN" column shows the number of hole expressions in the initial program. The "Env Merge" column shows the running time of the environment merges, while the "HB Merge" column shows the running time of the hole bindings merges. "Env Merge" and "HB Merge" are parts of "Backward Evaluator" and they are explained in Section 4.3.2.

In Table 1, as the number of holes in the program increase from 1 to 10, the time difference of forward evaluation is within 1ms, almost the same as that of the complete program. And it takes around the same amount of time to parse and print code. However, with the increase of holes, the time difference of backward evaluation is greater, and the difference is up to 21ms when there are 10 holes. This is an acceptable increase compared to the time it takes to parse code and HTML. The time to parse HTML code has also increased slightly, and this is due to hole elements in the HTML page, which take up some time.

There is a difference between the complete program and the incomplete program in backward evaluation, and the difference increases with the increase in the number of holes. This is because the time of environments merge and hole-bindings merge increases and accounts for most of the running time of the backward evaluation, as can be seen from Table 1. In both merges, values need to be compared. After adding holes into the definition of values, the comparison becomes more complicated because a hole value is carried with a closure, which can have nested hole values. Therefore, the recursive comparison process is very time-consuming.

7 RELATED WORK

Our work on bidirectional live programming is much related to the work on program sketching, direct manipulation programming system, live programming, and program debugging.

Program Sketching. Program sketching is a useful technique where developers express their high-level insights using an incomplete program as a sketch and leave the low details to the computer to synthesize [15]. Justin et al. [10] propose a *bidirectional evaluation* to propagate input-output examples through partially evaluated sketches. It is a bidirectional evaluation of incomplete programs, but there is a sharp difference from ours: it propagates input-output example constraints to the holes, whereas our method propagates output values back to the program. In particular, as our intention is to directly manipulate the output of incomplete programs, both holes and their closures are editable in our method but not editable in [10].

Direct Manipulation Programming System. Our approach is built upon the bidirectional evaluation in Sketch-n-Sketch [11], with a significant extension from complete programs to incomplete programs. In fact, our framework is the first to perform direct manipulation on incomplete programs, which is in sharp contrast to the traditional direct manipulation systems, such as [1, 3, 7, 16], which can work on only complete programs.

Table 1: Running Time of Bidirectional Evaluations

HN	Parse Code	Forward Eval	Parse HTML	Backward Eval	Print	Env Merge	HB Merge
0	36.3	3.3	31	8.17	1.13	8.4	1
1	37.07	3.43	39.1	13.33	1.17	9.77	1.67
5	36.37	3.7	38.4	21.1	1.07	11.47	5.27
10	35.63	4.27	46.67	29.97	1	16.43	6.53

HN: Number of holes in program; Env Merge: Environment Merge; HB Merge: Hole Bindings Merge
Parse Code refers to the "Parser" in Figure 6; Forward Eval refers to the "Forward Evaluator" described in Section 4.2; Parse HTML refers to "HTML Parser"; Backward Eval refers to "Backward Evaluato" described in Section 4.3; Print refers to "Printer" in Figure 6.

Live Programming. Live programming allows developers to edit the code of a running program and immediately see the effect of the code changes [2]. To deal with incomplete programs, Omar et al. [14] proposed to model incomplete functional programs as expressions with holes. Inspired by this work, we extend it from "unidirectional" to "bidirectional".

Recently, Omar et al. [13] import hole expressions to define a typed livelit calculus, and the system continuously gathers closures associated with the hole that the livelit is filling. Livelit helps developers with some special editing tools, including colors, tabular data, and diagrams, through user-defined GUIs to fill holes in the program online. Essentially, this approach is still unidirectional. Although our work is also to fill in the hole expressions, we do it by editing the output and propagating it back to the program.

Program Debugging. Program debugging [8] is related to backward evaluationbecause when the output of the wrong test case is known, backward evaluation can help to locate the wrong code fragments. For instance, Faddegon and Chitil [5] present an approach to algorithmic debugging that builds a computation tree from the runtime value observations by adding annotations to suspected modules, of which the aim is to limit the scope of debugging. As demonstrated in the paper, our framework can be used for debugging, where the suspicious expression can be set as a hole and the observation in hole bindings and hole closures helps with decision-making. As our future work, we are interested in looking into the deep relationship between algorithmic debugging and backward transformation of incomplete programs.

8 CONCLUSION

We present the first framework to support bidirectional live programming for incomplete programs. Essentially, it can be considered either as bidirectionalization of the technique of live programming for incomplete programs or as an extension of the framework of bidirectional live programming from complete programs to incomplete programs. The challenge lies in the richer output (of incomplete programs) which makes backward evaluation difficult. In particular, the holes need much attention because they can be instantiated many times and decomposed into sub-holes during evaluation. We carefully design the algorithms for efficient bidirectional evaluation and implement the tool Bidirectional Preview that is shown to be useful to develop various HTML applications interactively. In addition, we highlight the usefulness of bidirectional live programming of incomplete programs in algorithm teaching and program debugging. It is worth noting that although only functional programs are considered in this paper, the framework can be adapted to other programs such as imperative programs.

REFERENCES

[1] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. 1989. A Two-View Approach to Constructing User Interfaces. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 137–146. https://doi.org/10.1145/74334.74347
[2] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. *SIGPLAN Not.* 48, 6 (June 2013), 95–104. https://doi.org/10.1145/2499370.2462170
[3] 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
[4] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, Richard F. Paige (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–283.
[5] Maarten Faddegon and Olaf Chitil. 2015. Algorithmic Debugging of Real-World Haskell Programs: Deriving Dependencies from the Cost Centre Stack. *SIGPLAN Not.* 50, 6 (June 2015), 33–42. https://doi.org/10.1145/2813885.2737985
[6] 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. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es. https://doi.org/10.1145/1232420.1232424
[7] Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. 2014. CapStudio: An Interactive Screencast for Visual Application Development. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI EA '14)*. Association for Computing Machinery, New York, NY, USA, 1453–1458. https://doi.org/10.1145/2559206.2581138
[8] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *Softw. Test. Verification Reliab.* 26, 1 (2016), 40–68. https://doi.org/10.1002/stvr.1574
[9] K. kwok and G. Webster. 2016. *Carbide Alpha.* https://alpha.trycarbide.com/
[10] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug 2020), 1–29. https://doi.org/10.1145/3408991
[11] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional evaluation with direct manipulation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct 2018), 1–28. https://doi.org/10.1145/3276497
[12] Keisuke Nakano. 2019. Towards a Complete Picture of Lens Laws. arXiv:1910.10421 [cs.PL]
[13] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059
[14] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2018. Live Functional Programming with Typed Holes. arXiv:1805.00155 [cs.PL]
[15] Armando Solar-Lezama. 2013. Program Sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5–6 (Oct. 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7
[16] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 16, 11 pages. https://doi.org/10.1145/2393596.2393614