ISSN 1884-0760

GRACE TECHNICAL REPORTS

Bidirectional Transformation on Ordered Graphs

Fei Yang Soichiro Hidaka

GRACE-TR 2015–08 December 2015

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Bidirectional Transformation on Ordered Graphs*

Fei Yang[1] and Soichiro Hidaka[2]

[1] Eindhoven University of Technology, The Netherlands
f.yang@tue.nl
[2] National Institute of Informatics, Japan
hidaka@nii.ac.jp

Abstract. The linguistic (language-based) approach plays an important role in the development of well-behaved structural bidirectional transformations. It has been successfully applied to solve the challenging problem of bidirectional transformation on graphs by establishing a clear bidirectional semantics based on a bulk semantics of the structural recursion. However, this result was limited to unordered graphs, where the order between outgoing edges of nodes is disregarded, and it was not clear how to treat ordered ones such as XML documents with pointers. λ_{FG} is a language for querying ordered graphs, in which, a bulk semantics of structural recursion, extended with children rearrangement capability, is provided in a unidirectional setting. In this paper, we show that (the first-order subset of) λ_{FG} can be bidirectionalized by a three-stage procedure. The forward evaluation generates a view graph with comprehensive order-aware trace information. The traceable view enables us to reflect the edits on the view to the updates in the source by backward evaluation. We adopt a classical notion of ϵ-edges to represent the unobservable short cuts between nodes, which are fully utilized in bidirectionalization to keep the original shape of the input graphs. Thus our bidirectionalization is completed by a novel order-aware bidirectional procedure to eliminate ϵ-edges.

1 Introduction

Bidirectional transformations [9,8] provide a graceful mechanism for the synchronization and maintenance of the structures and contents in transformations. They are pervasive and have many potential applications, including the synchronization of replicated data in different formats [9], presentation-oriented structured document development, interactive user interface design [19], coupled software transformation, and the well-known *view updating* mechanism which has been intensively studied in the database community [3,10].

The linguistic (language-based) approach [9] gives us a promising way for the development of structured and well-behaved bidirectional transformation, in which every expression simultaneously specifies both a forward and the corresponding (correct) backward transformation, and every composition of expressions defines a structural gluing of smaller bidirectional transformations to a bigger one. Despite its usefulness for

* LNCS style file is used to format this article.

bidirectional transformation on lists and trees [9,5,18,16,21], it is a challenge to implement it on graphs. Firstly, unlike lists and trees, there is no unique way of representing, constructing, or decomposing a general graph, and this requires a more precise definition of *equivalence* between two graphs. Secondly, graphs have *shared nodes and cycles*, which makes both forward and backward evaluation non-trivial; naïve recursive evaluation for tree structures would visit the same nodes possibly infinitely many times.

In our previous work [14], we tackled the problem by showing that the linguistic approach can be applied to bidirectional transformation on graphs, where a clear bidirectional semantics is given for UnCAL, a graph algebra of the known graph query language UnQL [6]. The key to this success is that the bulk semantics of the structural recursion function is evaluated by first processing *in parallel* on all edges of the input graph and then combining the results. This bulk semantics relies on ϵ-edges (short cuts between nodes like those in automata) to graphs, providing a smart way of treating shared nodes and cycles in graphs and of tracing back from the view to the source.

However, the graphs that can be dealt with in this manner must be unordered, which means the order of outgoing branches of nodes are disregarded. It has not yet been known, in a bidirectional setting, how to treat ordered graphs such as XML graphs like that in Figure 1 that represents book data with ordered sections and mutually cyclic references, where we would like to extract the dotted part to see outlines of the book, by selecting first sections (first branches are rendered in thick arrows) while removing references, and further reflect the updates on the outline to the original book data. One might consider to encode the ordered graphs in terms of unordered ones by introducing some special edge labels, but this would make it difficult to keep consistency of the labels as discussed in [11]. In [11], a transformation language λ_{FG} for ordered graphs is provided, where a bulk semantics to transform ordered graphs is also given in unidirectional setting. Thus, a natural question is how we can bidirectionalize transformation on ordered graphs via λ_{FG}. However, combination of [11] and [14] is not straightforward, since λ_{FG} not only respects the order of siblings during transformation, but also rearranges them, like the example above that extracts only the first sections. The order-aware ϵ-edge elimination should also be bidirectionalized. It is required not only from user interface point of view where users usually do not directly operate on graphs with ϵ-edges, but also from theoretical point of view because ϵ-edges should be eliminated before every structural recursion in ordered settings to make structural recursion well-defined [11].

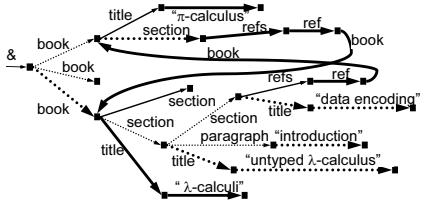

Fig. 1. Ordered Graph Representation of Books

In this paper, we show that the transformation on ordered graphs can be bidirectionalized. An earlier attempt [12] with $UnCAL^O$ also extended *UnCAL*. However, $UnCAL^O$ cannot rearrange siblings as we do in this paper. Our work is based on a first-order subset of λ_{FG} defined in [11]. As a result, we are able to bidirectionalize all the examples appeared in [11] that use bulk semantics, as well as the example men-

tioned in this section. The first-order restriction comes from the fact that we do not have clear semantics of updates of function-type views. Nevertheless, we lose no expressive power relative to [14]. Rather, we have refined the condition in which well-behavedness is guaranteed in the presense of interference between variable bindings, as well as the strategies of edge-deletion reflections. The main technical contributions are three folds. First, a novel three-stage bidirectional transformation strategy is designed for ordered graphs. Next, we establish a bidirectional evaluation semantics for λ_{FG}. Finally, we give an order-aware bidirectionalized procedure to eliminate ϵ-edges.

Organization of the Paper We shall start from a preliminary on the ordered graph model and the ordered graph transformation language λ_{FG} in Section 2. In Section 3, we present an overview of our three-stage bidirectionalization, and the properties which specify the goal of the design. The semantics of forward evaluation enriched with trace information, and backward evaluation to reflect view updates, are discussed in Sections 4 and 5. Then in Section 6, we present a bidirectionalized procedure for eliminating ϵ-edges. Finally, we discuss the related work and make a conclusion in Sections 7 and 8. The completed semantics, proofs and other materials are provided in the Appendix.

2 Preliminaries

2.1 Ordered Graphs

The Graph Model We start with an introduction of the formal definition for ordered graphs, which is inherited from [11].

We presuppose a finite set $\mathcal{L}$ of *labels* and an abbreviation $\mathcal{L}_\epsilon$ for $\mathcal{L} \cup \{\epsilon\}$. Let X and Y be finite sets of input and output markers, and we add the prefix & for markers like &x. An *ordered graph* G is defined by a triple (V, B, I), where V is a finite set of *nodes*, $B : V \to \mathit{List}(\mathcal{L}_\epsilon \times V + Y)$ is the total function mapping a node to a list of *branches*: a branch in $\mathcal{L}_\epsilon \times V + Y$ is either a *labeled edge* $\mathsf{Edge}(l, v)$ to node v with label l, or an *output marker* $\mathsf{Outm}(\&y)$, and $I : X \to V$ is the total function which determines the *input nodes* (roots) of the graph.

Example 1. The ordered graph in Figure 2(a), is represented as (V, B, I), where

$$
\begin{aligned}
V &= \{1, 2, 3, 4, 5, 6\} \\
B &= \{1 \mapsto [\mathsf{Edge}(\epsilon, 2), \mathsf{Edge}(a, 3), \mathsf{Edge}(\epsilon, 4)],\ 2 \mapsto [\mathsf{Edge}(\epsilon, 5)],\ 3 \mapsto [], \\
&\quad\ 4 \mapsto [\mathsf{Edge}(a, 3), \mathsf{Edge}(b, 6)],\ 5 \mapsto [\mathsf{Edge}(a, 6)],\ 6 \mapsto [\mathsf{Outm}(\&y)]\} \\
I(\&) &= 1
\end{aligned}
$$

We use G^X_Y to denote a graph G with an input marker set X and an output marker set Y. Moreover, we write $\mathcal{G}^X_Y$ to represent the set of such graphs. Although the graph data model in [11] also theoretically consider infiniteness of branches, it is sufficient for us to consider only finite cases.

For input markers, we allow a graph to have multiple roots. When the graph is single-rooted, we often use & as the *default marker* to indicate the root and use $\mathcal{G}_Y$ to denote $G^{\{\&\}}_Y$.

Proper Branches and Bisimilarity In the definition of ordered graphs, ϵ-edges are introduced to represent shortcuts between nodes. However, the sequence of ϵ-edges makes some observable branches of a node which are reachable from shortcuts implicit, i.e. we can not get every immediate observable branch of a node without searching through such sequence. We handle this difficulty by defining the notion of *proper branch*, which is a path from a node v, going through zero or more ϵ-edges until it reaches a non-ϵ-edge or an output marker. This notion relates the source and destination of an observable edge or an output marker.

Let us consider Figure 2(a) in which the list of branches of nodes are drawn in counter-clockwise order with the first branches marked by thicker lines. Node 1 and 6 are connected by two proper branches ($1 \xrightarrow{\epsilon} 2 \xrightarrow{\epsilon} 5 \xrightarrow{a} 6$ and $1 \xrightarrow{\epsilon} 4 \xrightarrow{b} 6$). However, such connectivity information does not appear explicitly in the branches before ϵ-elimination.

Let $G = (V, B, I)$ and $v \in V$. The path starting from v

$$v(= v_0) \xrightarrow{\epsilon}_{i_0} v_1 \ldots \xrightarrow{\epsilon}_{i_{n-1}} v_n \rightarrow_{i_n}$$

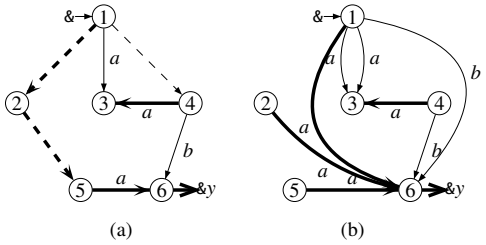

Fig. 2. Two Equivalent Graphs

($v \xrightarrow{l}_i$ denotes the i-th edge branch (labeled l) of node v) is called a *proper branch* of v if the i_n-th branch $B(v_n).i_n$ is not an ϵ-edge (either a non-ϵ edge or an output marker), and all the previous steps are ϵ-edges. The set of all proper branches of v in G is denoted by $Pb(G, v)$.

We impose a total order on proper branches. Let us consider two proper branches in an arbitrary graph, $p = (v \xrightarrow{\epsilon}_{i_0} v_1 \ldots \xrightarrow{\epsilon}_{i_{n-1}} v_n \rightarrow_{i_n})$, and $p' = (v \xrightarrow{\epsilon}_{i'_0} v'_1 \ldots \xrightarrow{\epsilon}_{i'_{n'-1}} v'_{n'} \rightarrow_{i'_n})$. Let their branch index sequences be $\tilde{p} \stackrel{def}{=} (i_0, \ldots, i_{n-1}, i_n)$ and $\tilde{p'} \stackrel{def}{=} (i'_0, \ldots, i'_{n'-1}, i'_{n'})$. We define $p \leq_{\mathrm{Pb}} p' \stackrel{def}{\Longleftrightarrow} \tilde{p} \leq_l \tilde{p'}$ where $\leq_l$ is the lexicographical order between branch index sequences. This order helps us to define a bisimilarity and ϵ-elimination as natural extensions of those in unordered setting, as well as to retain the correspondence of branches within the procedure of bidirectional elimination of ϵ-edges.

For a given proper branch p, we use $p.last$ to denote the last step of the branch sequence, which is either a non-ϵ edge or an output marker.

For the soundness of the graph model, we need an equivalence relation that can judge whether two arbitrary graphs are the same. Here we use the bisimilarity on ordered graphs defined in [11]. For example, the two graphs in Figure 2 are bisimilar, but not isomorphic. Note that we assume that node 1 is the root node. Thus the bisimilarity equivalence would only respect to the behavior of the parts reachable from node 1 through proper branches. The equivalence relation relates the pair of nodes which has an order isomorphism on their proper branch index sequence, satisfying that their reachable nodes through correspondent proper branches with identical edge label or output marker, respectively, are also equivalent.

2.2 The Core λ_{FG}

λ_{FG} is a language for transforming (querying) ordered graphs. It is an extension of typed λ-calculus with graph constructors and list functions. It has a powerful feature of structural recursion, and is capable of manipulating the sibling branches of nodes, which considerably extends the expressiveness of the language. The syntax of λ_{FG} is given as below, in which we only focus on a graph-related first-order subset.

$$
\begin{array}{ll}
e ::= \$x \mid \lambda\$x.e\ e \mid (e,e) \mid \mathbf{pr_l}\ e \mid \mathbf{pr_r}\ e & \{ \text{ terms of lambda calculus } \} \\
\mid \mathbf{nil} \mid \mathbf{cons}(e,e) \mid \mathbf{foldr}(e,e) \mid \ldots & \{ \text{ functions for lists } \} \\
\mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{isEmpty}(e) & \{ \text{ conditional and emptiness} \} \\
\mid a \mid e = e & \{ \text{ labels } (a \in \mathcal{L}) \text{ and label equality } \} \\
\mid \underline{[]} \mid e \mathbin{\underline{+\!\!+}} e \mid \underline{[}e : e\underline{]} \mid \underline{[}\&y\underline{]} \mid \&x := e & \\
\mid () \mid e \oplus e \mid e@e \mid \mathbf{cycle}(e) & \{ \text{ graph constructors } \} \\
\mid \mathbf{srec}(e,e)\ e & \{ \text{ structural recursion functions} \}
\end{array}
$$

$\mathbf{pr_l}$ and $\mathbf{pr_r}$ respectively denote left and right projections of pairs.

The graph constructors are introduced in Appendix A, and the type rules are discussed in Appendix B. The enriched semantics is briefly given in Section 4 and complemented in Appendix C.

Structural Recursion Functions Here we focus on the explanation of *structural recursion*, which provides a mechanism to describe the queries and transformations that guarantees termination of the computation and preserves finiteness.

In general a structural recursion is written as a function $\mathbf{srec}(e,d)$, where $e : \mathbf{Label} \times \mathbf{G}_Y \to \mathbf{G}_Z^Z$ is the body function applied on the labels of labelled edges and the subgraphs reachable from the edges, and $d : \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z + \mathbf{G}_{Z\times Y}^Z) \to \mathbf{G}_{Z\times\alpha+Z\times Y}^Z$ is a rearrangement function manipulating sibling branches obtained from the body functions for each node. The output marker type $Z \times \alpha$ in the domain of d corresponds to the output marker branch produced by the body function, where α is instantiated to node identifier and used to make connections to other fragment of the result of structural recursion in the bulk semantics. Parametricity implied by α means programmer of d do not manipulate this output marker. Another output marker type $Z \times Y$ corresponds to the output marker branches of the input graph of **srec**. Therefore, the output markers in the result of d are disjoint sum of these output marker types. For an example of d, we consider a simple case where $d = \mathbf{foldr}(\odot, \iota_\odot)$ for some monoid $(\odot, \iota_\odot)$ on $\mathbf{G}_{Z\times\alpha}^Z$(graph type with sets of input markers Z and output markers $Z \times \alpha$). We provide a bulk semantics for the evaluation of structural recursion functions, and its traceable forward semantics is given in Section 4.

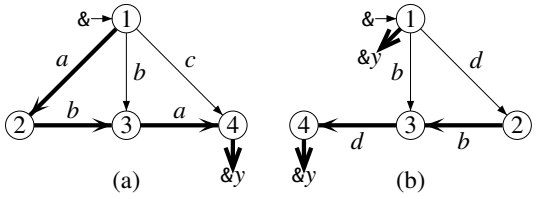

Fig. 3. Source (a) and view (b) of *a2d_xc*

Example 2. This example shows how to manipulate edges of the graph and change its shape by structural recursion. As shown in Figure 3, the following function *a2d_xc* replaces all labels `a` with `d` and contracts `c`-labeled edges, and reverse the order of the branches of each node. Note that the contraction of the edge from Node 1 to Node 4 leads to a shortcut from Node 1 to an output marker branch.

$$
\begin{array}{l}
a2d\_xc = \mathbf{srec}(rc, \mathbf{foldr}(\underline{\hat{+\!\!+}}, \underline{[]})) \\
\quad \text{where} \quad x \,\underline{\hat{+\!\!+}}\, y \;\; = y \,\underline{+\!\!+}\, x \\
\qquad\quad rc(\$l, \$g) = \mathbf{if} \;\; \$l = \mathtt{a} \;\mathbf{then}\; \underline{[}\mathtt{d} : \underline{[\&]}\underline{]} \\
\qquad\qquad\qquad\quad \mathbf{else\ if}\; \$l = \mathtt{c} \;\mathbf{then}\; \underline{[\&]} \\
\qquad\qquad\qquad\quad \mathbf{else} \qquad\qquad\quad \underline{[}\$l : \underline{[\&]}\underline{]}
\end{array}
$$

Example 3. The next example removes the branches in the even positions in the siblings of the root. The d funciton $\mathbf{pr_l} \circ (\mathbf{foldr}(f, (\mathbf{nil}, \mathbf{nil})))$ is used to do the selection using tupling technique [15].

$$
\begin{array}{l}
even\_remove = \mathbf{srec}(id, \mathbf{pr_l}(\mathbf{foldr}(f, (\mathbf{nil}, \mathbf{nil})))) \\
\quad \text{where} \quad id(\$l, \$g) = \underline{[}\$l : \$g\underline{]} \\
\qquad\quad f(\$x, \$xs) = (\mathbf{pr_r}\ \$xs, \mathbf{cons}(\$x, (\mathbf{pr_l}\ \$xs)))
\end{array}
$$

Restrictions on the second argument of **srec** We restrict the form of the second argument d of **srec** into $\mathbf{foldr}(e_1, e_2)$, where e_1 and e_2 should not necessarily form a monoid. Even with this restriction, the function d can do rearrangement (i.e., (1) swapping, (2) selection, (3) replication, (4) constructing new element) of siblings of given nodes by taking list of sibling graphs and returns another graph. Further graph transformations in (first-order subset of) λ_{FG} on these siblings are also possible. When e_1 and e_2 are restricted to monoids, selection of siblings like in Example 3 would not have been possible. **foldr**, instead of unbiased **fold** on join list, is also crucial to achieve order-based selection like selecting the first element, in order to provide unique decomposition of sibling graphs.

Example 4. Figure 1 represents books with cyclic references. The following transformation *outline* extracts outlines of these books by selecting first sections while removing references. The transformation returns the subgraph that is rendered with dotted edges.

$$
\begin{array}{l}
outline = \mathbf{srec}(\lambda(\$l, \$g).\underline{[}\$l : fstsection(sections(cutref(\$g)))\underline{]}, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\
\text{where}\ cutref \;\; = \mathbf{srec}(\lambda(\$l, \$g).\ \mathbf{if}\ \$l = \mathtt{refs}\ \mathbf{then}\ \underline{[]}\ \mathbf{else}\ \underline{[}\$l : \underline{[\&]}\underline{]}, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\
\qquad sections \;\; = \mathbf{srec}(\lambda(\$l, \$g).\ \mathbf{if}\ \$l = \mathtt{section}\ \mathbf{then}\ \underline{[}\$l : \$g\underline{]}\ \mathbf{else}\ \underline{[]}, \mathbf{foldr}(\underline{+\!\!+}, \underline{[]})) \\
\qquad fstsection = \mathbf{srec}(\lambda(\$l, \$g).\underline{[}\$l : \$g\underline{]}, \mathbf{foldr}(\mathbf{pr_l}, \underline{[]}))
\end{array}
$$

Bulk Semantics The structural recursion of λ_{FG} has two equivalent semantics. One is *recursive semantics*. It defines the function behavior and the program transformation/optimization recursively. It appears to be more concise and is useful for reasoning. Another one is *bulk semantics*. By allowing ϵ-edges, we can evaluate a structural recursion in a *bulk* manner. The bulk semantics consider the transformation on the whole. It is useful in parallel computation and the proof of termination and finiteness-preserving property.

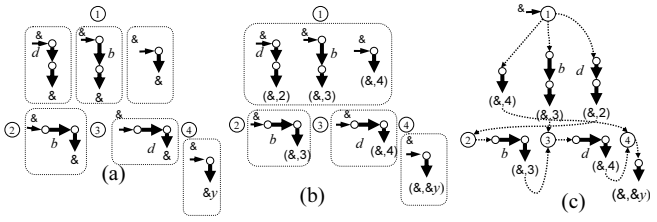

Fig. 4. Illustration of bulk semantics

Let us consider a structural recursion **srec**(e, d). In the bulk semantics, the evaluation on every graph component is applied in parallel. It involves the following three steps (an enriched formal semantics is presented in Section 4). We shall also revisit the transformation *a2d_xc* to transform the graph in Figure 3(a) to Figure 3(b). The corresponding bulk semantics is illustrated in Figure 4.

1. Map computation on edges with e: the function e is applied on each labeled edge and the following subgraphs, yield a set of intermediate result graphs for e. These are surrounded by dotted round squares in Fig. 4(a). Results are bundled into list of graphs for each node of input graphs which are surrounded by dotted round squares in Fig. 4(b), where output markers are associated with the node id to be connected.
2. Map computation on nodes with d: The above list of graphs are merged by the rearrangement measure defined in d, leading to a set of merged graphs for each node. In Figure 4(c), the rearrangement places the output marker branch under node 1 in the first position and edge d in the last position in the siblings.
3. Groups new graphs with ϵ-edges: to get the final result, the graphs for each node need to be grouped together by ϵ-edges. This is depicted in Figure 4(c). The ϵ-edges are added according to the input and output markers of the previous result. After ϵ-elimination, we the graph in Figure 3(b) is obtained.

3 An Overview of Bidirectional Transformation

3.1 Bidirectional Properties

The bidirectionalization of transformation on ordered graphs is approached by providing the bidirectional semantics of λ_{FG}. We aim to obtain a semantics that has bidirectional properties as defined in the previous work for UnCAL [14]. A whole picture of bidirectional transformation is shown in Figure 5. Let $\mathscr{F}[\![e]\!]\rho$ denote a forward evaluation (get) of expression e under environment ρ containing the source graph G_S to produce a view G_V, and $\mathscr{B}[\![e]\!](\rho, G'_V)$ denote a backward evaluation (put) of expression e under original environment ρ to reflect a possible modification on the view G'_V to the source by computing an updated environment ρ', from which the updated source graph G'_S could be extracted. An *environment* ρ is a mapping with a form $\{\$x \mapsto X, \ldots\}$ where X is a graph G or a label l, and $\$x$ is a variable. The transformation needs to satisfy the following two important properties (equality stands for exact equivalence here):

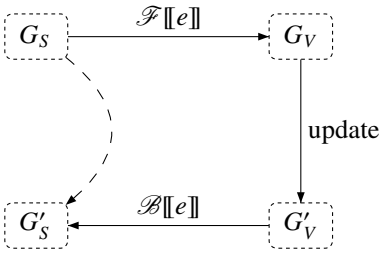

Fig. 5. Schema of Bidirectional Transformation

$$\frac{\mathscr{F}[\![e]\!]\rho = G_V}{\mathscr{B}[\![e]\!](\rho, G_V) = \rho}(GETPUT) \qquad \frac{\mathscr{B}[\![e]\!](\rho, G'_V) = \rho' \; \mathscr{F}[\![e]\!]\rho' = G''_V}{\mathscr{B}[\![e]\!](\rho, G''_V) = \rho'}(WPUTGET)$$

The (*GETPUT*) property states that unchanged view G_V should give no change on the environment ρ in the backward evaluation, while the (*WPUTGET*) property states that the modified view G'_V and the view G''_V obtained by backward evaluation on G'_V immediately followed by forward evaluation may differ, but both of the views have the

same effect ρ' on the original source ρ if backward evaluations are applied on them. We consider a pair of forward and backward evaluation is *well-behaved* if it satisfies (*GETPUT*) and (*WPUTGET*) properties.

In general, forward and backward transformations may not be totally defined. In our language, forward transformation fails if ϵ-elimination fails, and we have an effective decision procedure [11]. Backward transformation fails if the update on the view is not propagable, like update of constant values created in the transformation. See Sections 4 and 5 for more detail on each case of failure.

On our proposed bidirectionalization, we have the following theorem.

Theorem 1 (Well-behavedness). *The proposed forward and backward evaluations on λ_{FG} are well-behaved, provided their evaluations succeed.*

According to the (*WPUTGET*) property, given a certain forward evaluation function, there could be more than one backward evaluation that would satisfy the well-behavedness property. Moreover, the modifications on the view graph could be categorized into three types: in-place updates (edge-renaming), edge deletion and subgraph insertion. We select the rational choice for updating the source graph by three principles: *location correspondence* to reflect modifications on the corresponding part in the source, the *least change* [19] on the set of updating edges, and the *monotonicity* on the modification operations (deletion is reflected to deletion, and so on). Henceforth, such choice could be a prescriptive goal in designing the bidirectional transformation. The goal is to let the updates on the source most possibly meet the intention of the users for the change on the view.

3.2 Three-Stage Bidirectionalization

In UnCAL, a two-stage framework is proposed for bidirectionalizing unordered graph transformation [14], in which during the forward transformation we first apply the evaluation with semantics enriched with trace information, then eliminate the ϵ-edges to produce a usual view.

The most challenging part of bidirectionalization on λ_{FG} comes from the capability of rearrangement of sibling branches in the structural recursion. It requires a three step bulk semantics as discussed in Section 2, which makes it more complicated to retain a consistent trace between the source and view. The original method in [14] is not capable of handling the sibling transformations. Further, there is a requirement that the structural recursion with general d function only accepts input graphs without ϵ-edges to keep the bisimulation genericity (actually, the presence of ϵ-edges breaks the bisimulation genericity, see [11] for a counterexample). Thus, an ϵ-elimination procedure is essential before applying the structural recursion for an arbitrary graph. Also, the ϵ-edge should be eliminated before it is presented to the end-user. Moreover, the order of the branches must be recorded during the bidirectional transformation. Our basic idea is to divide the forward evaluation into three stages, each of which could be bidirectionalized.

- Stage 1: Elimination of ϵ-edges on the source, to produce a simple source on which we can apply bulk semantics.

- Stage 2: Forward evaluation using bulk semantics. This stage may create some new ϵ-edges, so that the output graph will have a similar shape to the input graph. Appropriate trace information might be appended locally on the nodes in the result graph.
- Stage 3: Elimination of ϵ-edges to produce a usual view. The method is the same as that in Stage 1.

In our bidirectional transformation framework, we will propose in Section 6 a general bidirectionalized ϵ-edge elimination procedure for both Stage 1 and Stage 3. For Stage 2, a bidirectionalized semantics for λ_{FG} is provided in Section 4 and 5. Thus, the whole transformation is bidirectionalized by an inductively defined procedure.

4 Traceable View and Reflection of Inplace-Updates

Practically, a λ_{FG} expression usually specifies a forward transformation that maps a source ordered graph (possibly a database or an XML file) to a view graph, and a backward transformation is the procedure to reflect view updates to the source graph. However, the structural information kept by ϵ-edges is implicit in the view graph. Moreover, the newly constructed parts during transformation can not be traced back in the source. In this sense, some appropriate trace information (like provenance traces [7]) should be added to the view, to make the view more informative, in other words, *traceable*. In this section, we will enrich the original semantics of λ_{FG} to produce a traceable view after forward transformation.

4.1 Local Trace Information

In our scenario, a view is obtained by evaluating a λ_{FG} expression with an ordered graph. A node in the view graph could be obtained from the source graph, or constructed by the λ_{FG} expression, or generated by a structural recursion function (in bulk semantics). To record this lineage of computations, we wrap the tracing information up in every node locally in the view graph. We define *TraceID* for every node as

$$
\begin{array}{lll}
\mathit{TraceID} := \mathit{SrcID} & & \mid \mathsf{Code}\ \mathit{Pos}\ \mathit{Marker} \\
\quad \mid \mathsf{RecN}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{Marker} & & \mid \mathsf{RecE}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{TraceID}\ \mathit{Num} \\
\quad \mid \mathsf{RecM}\ \mathit{Pos}\ \mathit{Marker}\ \mathit{TraceID}\ \mathit{Num} & & \mid \mathsf{RecD}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{TraceID}
\end{array}
$$

where *SrcID* ranges over the identifiers uniquely assigned to all nodes in the source graph, *Pos* ranges over code positions in the λ_{FG} expressions, *Marker* ranges over input/output makers, and *Num* ranges over the lists of integers, which represent the position of the branch among its siblings.

Now we briefly explain the information recorded by each *TraceID*. *SrcID* means the node is generated from the corresponding one in the source. Code p &m is the information for the nodes created by the λ_{FG} constructor on the code position p, where &m is used only for constructors involving markers. RecN, RecE, RecM, RecD are four kinds of trace information used in bidirectionalizing structural recursion $\mathbf{srec}(e_1, d)(e_2)$. RecN, RecE, RecM represent the node in the intermediate result after e_1 is applied,

which are generated from the nodes/non-marker branches/output marker branches respectively, and RecD information is added when the rearrangements on the lists of siblings are applied by d. RecN $p\,v\,\&m$ denotes a node originated from node v generated by e_2 for component $\&m$ (each component of mutually recursive functions are encoded by a marker), and it corresponds to nodes ① through ④ in Figure 4. RecE $p\,v\,w\,i$ denotes a node originated from node w generated by e_1, which in turn is generated by i-th branch of node v generated by e_2, and node w corresponds to the nodes drawn with small circles in the round dotted squares except the one on the right bottom of Figure 4(a). RecM $p\,\&m\,v\,i$ denotes a node originated from i-th marker ($\&m$) branch of node v generated by e_2, and corresponds to the nodes drawn with small circles in the dotted squares on the right bottom of Figure 4(a). and RecD $p\,v\,w$ denotes a node originated from node v generated by the function d applied on node w, where the node $w(\in \mathit{TraceID})$ is marked by RecN and identifies the list appeared in that d function. RecD constructor is applied to the nodes drawn with small circles in Figure 4(b) and (c).

4.2 Enriched Bidirectional Semantics and the Reflection of Inplace-updates

Now we proceed to define the enriched forward semantics of λ_{FG} which generates a traceable view. Compared with the original semantics [11] of λ_{FG}, the trace information is recorded whenever a node is created by the forward transformation. For each expression, we define its backward evaluation reflecting the inplace-update. We adopt the notion of code position to specify the expression which creates new graph elements. Let e^p denote a λ_{FG} subexpression e at code position p. We write $\rho(\$x)$ for G when $(\$x \mapsto G) \in \rho$, and $e\rho$ applies the variable substitution using ρ in the expression e. We only give the semantics for some representative expressions here, and complete with the remaining operators in Appendix C D. Moreover, the well-behavedness of the semantics is discussed in Appendix F

In order to merge the updated environment computed by the backward evaluation, we introduce a merge operator [14].

$$\rho_1 \uplus_\rho \rho_2 = \left\{ (x \mapsto \mathbf{mg}(G, G_1, G_2)) \;\middle|\; \begin{array}{l} (x \mapsto G_1) \in \rho_1 \\ (x \mapsto G) \in \rho \\ (x \mapsto G_2) \in \rho_2 \end{array} \right\}$$

$$\text{where } \mathbf{mg}(G, G_1, G_2) = \begin{cases} G_1 & \textbf{if } G_2 = G \vee G_1 = G_2 \\ G_2 & \textbf{if } G_1 = G \\ \mathit{FAIL} \text{ otherwise} \end{cases}$$

The operator $\uplus_\rho$ unifies environments on both hand sides (ρ_1 and ρ_2) updated relative to original environment ρ. For each variable x, if only one binding is updated, or both bindings are updated consistently, then the updated binding is adopted. Otherwise, the unification of inconsistent updates fails.

Graph Constructor Expressions The functions of graph constructor expressions are illustrated in Appendix A. For instance, the enriched forward semantics of single node graph constructor at code position p is (see also Figure 6(a)),

$\mathscr{F}[\![\underline{[]}^p]\!]\rho = (\{\mathsf{Code}\ p\}, \{\mathsf{Code}\ p \mapsto []\}, \{\& \mapsto \mathsf{Code}\ p\})$.

Note that the code position is recorded as a *TraceID*, which infers that it is generated by the expression at position p.

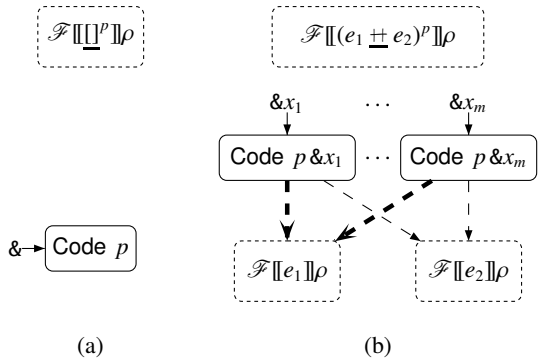

Fig. 6. $\mathscr{F}[\![[]^p]\!]\rho$ and $\mathscr{F}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!]\rho$

As its forward evaluation only generates a constant graph, in the backward evaluation, no modification is accepted on the view. The backward semantics is defined as:

$\mathscr{B}[\![[]^p]\!](\rho, G') = \rho$ if $G' = \mathscr{F}[\![[]^p]\!]\rho$ *FAIL* otherwise

Here, the parameter ρ is the original environment and G' is the updated view. We simply require that G' is equal to the result of forward evaluation on ρ. Thus the well-behavedness property is guaranteed whenever the backward evaluation succeeds.

For another example, the enriched forward semantics of the expression $e_1 \mathbin{\underline{+\!+}} e_2$ is defined as follows (see also Figure 6(b)).

$$\begin{aligned}
&\mathscr{F}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \mathbin{\underline{+\!+}^p} \mathscr{F}[\![e_2]\!]\rho \\
&\quad \text{where } G_1 \mathbin{\underline{+\!+}^p} G_2 = (V_1 \cup V_2 \cup V', B_1 \cup B_2 \cup B', I') \\
&\qquad (V_1, B_1, I_1) = G_1 \\
&\qquad (V_2, B_2, I_2) = G_2 \\
&\qquad M = \mathbf{inMarker}(G_1) = \mathbf{inMarker}(G_2) \\
&\qquad V' = \{\mathsf{Code}\ p\,\&m \mid \&m \in M\} \\
&\qquad B' = \{\mathsf{Code}\ p\,\&m \mapsto [\mathsf{Edge}(\epsilon, I_1(\&m)), \mathsf{Edge}(\epsilon, I_2(\&m))] \mid \&m \in M\} \\
&\qquad I' = \{\&m \mapsto \mathsf{Code}\ p\,\&m \mid \&m \in M\}
\end{aligned}$$

where $\mathbin{\underline{+\!+}^p}$ is a union operator for two graphs concerning position p. We write **inMarker**(G) and **outMarker**(G) to denote the set of input and output markers in a graph G, respectively.

For its backward evaluation, we first decompose the updated graph G' and apply backward transformation on each of the subexpressions e_1 and e_2 using the fragment graphs. We have the following definition, given $\mathsf{decomp}_{G_1 \mathbin{\underline{+\!+}} G_2}(G')$ defined in Appendix D as the decomposition of the graph G', then

$$\begin{aligned}
&\mathscr{B}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G'_1) \uplus_\rho \mathscr{B}[\![e_2]\!](\rho, G'_2) \\
&\quad \text{where } G_1 = \mathscr{F}[\![e_1]\!]\rho, \quad G_2 = \mathscr{F}[\![e_2]\!]\rho \quad (G'_1, G'_2) = \mathsf{decomp}_{G_1 \mathbin{\underline{+\!+}^p} G_2}(G')
\end{aligned}$$

Variables The query of ordered graph via λ_{FG} requires lambda abstraction and variable reference. The forward semantics for variable reference looks up the variable in the environment ρ and returns its binding.

$\mathscr{F}[\![(\$v)^p]\!]\rho = \rho(\$v)$

For an inplace-update, a variable simply updates its bindings as $\mathscr{B}[\![\$v]\!](\rho, G') = \rho[\$v \leftarrow G']$. Here, $\rho[\$v \leftarrow G']$ is an abbreviation for $(\rho \setminus \{\$v \mapsto \_\}) \cup \{\$v \mapsto G'\}$.

Condition The forward semantics of a condition is defined as

$$\mathscr{F}[\![(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)^p]\!]\rho = \begin{cases} \mathscr{F}[\![e_2]\!]\rho \textbf{ if } \mathscr{F}[\![e_1]\!]\rho \\ \mathscr{F}[\![e_3]\!]\rho \text{ otherwise} \end{cases}$$

It first evaluates the conditional expression e_1, and it chooses the right branch to evaluate according to the result of the conditional expression.

And the procedure of its backward evaluation is defined by

$$\mathscr{B}\left[\!\!\left[\begin{array}{r}\textbf{if } e_1 \textbf{ then } e_2 \\ \textbf{else } e_3\end{array}\right]\!\!\right](\rho, G') = \begin{cases} \rho_2' & \textbf{if } \mathscr{F}[\![e_1]\!]\rho \wedge \mathscr{F}[\![e_1]\!]\rho_2'', \textbf{ where } \rho_2' = \mathscr{B}[\![e_2]\!](\rho, G') \\ & \rho_2'' = \mathsf{pr}(\rho_2', \mathsf{utrans}(\mathsf{upd}_{\mathcal{A}}, t_1')) \\ & \quad t_1' = \text{external trace for } e_1 \text{ (see Section F)} \\ \rho_3' & \textbf{if } \neg\mathscr{F}[\![e_1]\!]\rho \wedge \neg\mathscr{F}[\![e_1]\!]\rho_3'', \textbf{ where } \rho_3' = \mathscr{B}[\![e_3]\!](\rho, G') \\ & \rho_3'' = \mathsf{pr}(\rho_3', \mathsf{utrans}(\mathsf{upd}_{\mathcal{A}}, t_2')) \\ & \quad t_2' = \text{external trace for } e_2 \\ \textit{FAIL} & \text{otherwise} \end{cases}$$

It is reduced to the backward evaluation of e_2 if e_1 holds, and to the backward evaluation of e_3 otherwise. To guarantee well-behavedness, we further ensure that the boolean expression e_1 does not change after backward evaluation. Because the result of e_1 may be influenced indirectly by backward evaluation of the bodies e_2 and e_3, if they update variable bindings, and the values of variables in the condition are produced from the updated bindings. It is essential to check the value of the condition with the new binding of variables. To compute the influence, we separately compute, for every expression, the mapping from every edge produced by the expression, to the corresponding edge in the source graph (if the view edge is originated from the edge in the source graph), and use these mappings to reflect (top-level) view updates to the edges bound to free variables in e_1. It is achieved by a mechanism similar to the classification of view edges used in [13]. See equation BIF in Appendix F for the detailed description.

Structural Recursion Figure 7 illustrates an overview of bidirectionalization of structural recursion functions. It applies the transformation in Example 2 to the source graph in Figure 3(a) to get a view graph in Figure 3(b). For simplicity, we omit code positions in these figures. As shown in Figure 7(a), the forward evaluation first applies edge mapping using body expression e_b which is *rc*, and results in a mapping from every node to a list of subgraphs. Two of these subgraphs are shown in Figure 7(b) and (c), which are corresponding to the results of a labelled edge and an output marker, respectively. The lists rearrangement

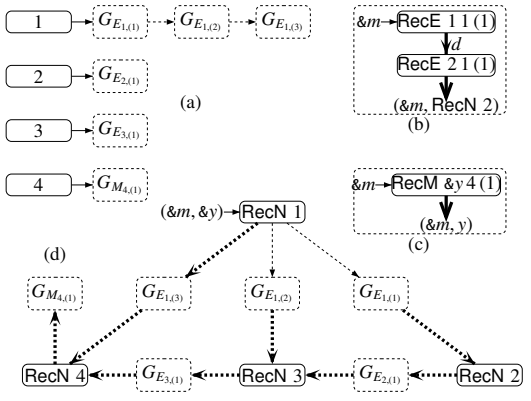

Fig. 7. Forward Evaluation of Bulk Semantics

and merging with ϵ-edges are illustrated in Figure 7(d). Finally, the view graph in Figure 3(b) is obtained by an ϵ-elimination procedure.

An overview of the forward evaluation is given below. Note that the functions EMap, DMap and Merge are defined for edge mapping e_{b}, list mapping d and view construction respectively.

$$
\begin{aligned}
&\mathscr{F}[\![(\mathbf{srec}_Z(e_{\mathrm{b}}, d)(e_{\mathrm{a}}))^p]\!]_\rho = (g') \\
&\quad \text{where } g' = \mathrm{Merge}(g.\mathrm{V}, B_{\mathrm{DMap}}, g.\mathrm{I}) \quad g = \mathscr{F}[\![e_{\mathrm{a}}]\!]_\rho \\
&\qquad \mathrm{EMap}(B) = \{u \mapsto \mathrm{BMap}(x) \mid (u \mapsto x) \in B\} \\
&\qquad \mathrm{BMap}(x) = \left[\begin{cases} \mathrm{ELable}(\mathscr{F}[\![e_{\mathrm{b}}]\!]_{\rho'}, v, u, i) & b = \mathsf{Edge}(l, v) \\ & \rho' = \rho \cup \{\$l \mapsto l,\ \$g \mapsto g|_v\} \\ (\mathrm{MLabel}(\&y, u, i)) & b = \mathsf{Outm}(\&y) \end{cases} \middle| \, (b, i) \in_{\mathrm{L}} x \right] \\
&\qquad \mathrm{DMap}(B) = \{u \mapsto (\lambda g.\mathrm{DLabel}(g, u)).\mathscr{F}[\![d]\!]_{\rho \cup \{\$b \mapsto x\}} \mid (u \mapsto x) \in B\} \\
&\qquad B_{\mathrm{DMap}} = \mathrm{DMap}(\mathrm{EMap}(g.\mathrm{B})) \\
&\quad [e \mid (x, i) \in_{\mathrm{L}} l] \stackrel{\mathrm{def}}{=} [e \mid x = l.i]_{i \in |l|}
\end{aligned}
$$

Backward evaluation of a structural recursion function is defined by the following stepwise procedure. It performs an inverse procedure of the forward evaluation. The most crucial point is to decompose the updated view graphs and to remove the ϵ-edges generated during the forward evaluation using trace information. Further, when we merge the updated environment, we first merge local updates on environments of body expressions, and then merge them to the whole updated environment. Since there might be overlapped parts among them, we let the backward evaluation fail if the overlapped parts are inconsistent.

$$
\begin{aligned}
&\mathscr{B}[\![\mathbf{srec}(e_{\mathrm{b}}, d)(e_{\mathrm{a}})]\!](\rho, G'_{\mathrm{srec}}) = \mathscr{B}[\![e_{\mathrm{a}}]\!](\rho, G') \uplus_\rho \biguplus \{\rho'_{v,i} \setminus \{\$l \mapsto \_\} \setminus \{\$g \mapsto \_\}\} \\
&\quad \text{where } (V_{\mathrm{src}}, B_{\mathrm{src}}, I_{\mathrm{src}}) = \mathscr{F}[\![e_{\mathrm{a}}]\!]_\rho, \quad (V'_{\mathrm{srec}}, B'_{\mathrm{srec}}, I'_{\mathrm{srec}}) = G'_{\mathrm{srec}} \\
&\qquad G' = (V' \cup \bigcup \rho'_{v,i}(\$g).V, B' \cup \bigcup \rho'_{v,i}(\$g).B, I_{\mathrm{src}}) \\
&\qquad V' = \{v \mid (v \mapsto s) \in B'\} \cup \{w \mid \mathsf{Edge}(l, w) \in s, (v \mapsto s) \in B'\} \\
&\qquad B' = \mathrm{EMap}'(B'_{\mathrm{EMap}}) = \mathrm{EMap}'(\mathrm{DMap}'(\mathrm{Merge}'(G'_{\mathrm{srec}}))) \\
&\qquad \mathrm{Merge}'(G') = \{u \mapsto (\mathrm{VMerge}'(G'.\mathrm{V}, u), \mathrm{BMerge}'(G'.\mathrm{B}, u), \mathrm{IMerge}'(G'.\mathrm{B}, u)) \mid u \in V_{\mathrm{src}}\} \\
&\qquad \mathrm{DMap}'(B'_{\mathrm{DMap}}) = \{u \mapsto \mathscr{B}[\![d]\!](\rho_u, \mathrm{DLabel}'(B'_{\mathrm{Dmap}}(u)))(\$b) \mid u \in V\} \\
&\qquad \mathrm{EMap}'(B'_{\mathrm{EMap}}) = \{u \mapsto [\mathrm{emapB}'(u, b, i) | (b, i) \in_{\mathrm{L}} x] \mid (u \mapsto x) \in B'_{\mathrm{EMap}}\} \\
&\qquad \mathrm{emapB}'(u, b, i) = \begin{cases} \mathsf{Edge}(\rho'_{u,i}(\$l), \rho'_{u,i}(\$g).\mathrm{I}(\&)) & (b, i) = (G_{\mathrm{E}_{\mathrm{u,i}}}, i) \\ \quad \text{where } \rho'_{u,i} = \mathscr{B}[\![e_{\mathrm{b}}]\!](\rho_{u,i}, \mathrm{ELabel}'(G_{u,i})) & \\ \mathrm{MLabel}'(G_{\mathrm{M}_{\mathrm{u,i}}}) & (b, i) = (G_{\mathrm{M}_{\mathrm{u,i}}}, i) \end{cases}
\end{aligned}
$$

We put the complete semantics for the bidirectional transformation of structural recursion functions in Appendix CD, where the detailed explanation of every subfunction can be found.

5 Reflection to Deletion and Insertion

A node in the view graph is either originated from a node in the source graph, or generated by the query (graph constructor or structural recursion function). Using the trace information generated by enriched forward evaluation, we are able to figure out its origin. In this section, we categorize backward evaluation algorithms by the types of view

updates: (1) edge-renaming (in-place updates), (2) edge-deletion, and (3) insertion of edges or a subgraph rooted at a node. We already have dealt with the first case in Section 4. In this section, we proceed to reflect the other two types of updates.

5.1 Reflection of Edge Deletion

Similarly to the method in our previous work in [14], we reflect deletion of an edge in the view as deletion of corresponding edge in the source utilizing the information within *TraceID*s. Here we identify an edge in the view by a tuple of its origin node and branch order.

The function corr defined below computes a corresponding edge of an edge in the view as a tuple of origin node ID and its branch order in the source.

$$\mathsf{corr}((u,i)) = (u,i) \quad \text{if } u \in \mathit{SrcID} \quad \mathsf{corr}((\mathsf{RecE}\ p\ u\ v\ i, j)) = \begin{cases} \mathsf{corr}((u,j)) \text{if } \mathsf{corr}((u,j)) \neq \mathit{FAIL} \\ \mathsf{corr}((v,i)) \text{if } \mathsf{corr}((u,j)) = \mathit{FAIL} \end{cases}$$

$$\mathsf{corr}((\mathsf{RecD}\ p\ u\ v, j)) = \mathsf{corr}((u,j)) \quad \mathsf{corr}(\zeta) = \mathit{FAIL} \quad \text{otherwise}$$

FAIL means failure in finding the corresponding edge. Also note that for the RecE-labeled branches, we choose the corresponding edge from the body expression of structural recursion first, rather than from the argument expression, to make the change to the new view created by the next forward transformation smaller (we later discuss as the principle of least change [19]). When the tracing hits the edge created by the query (the *TraceID* must be of the form Code), the evaluation fails. Moreover, the user would be informed of such invalid deletion.

Note that only non-ϵ edges are allowed to be deleted in the view. Thus, a node with trace information RecN or RecM never appears as an argument of corr.

Suppose the user deletes one edge, say (u,i) on the view graph G_V. We can get one corresponding edge on the source $\mathsf{corr}((u,i))$. In fact, multiple edges from the view G_V might have the same value on corr with (u,i). Let $D_{u,i} = \{(v,j) \mid \mathsf{corr}((v,j)) = \mathsf{corr}((u,i))\}$. The set is computed by an enumeration on all the edges in G_V. We adopt $G'_V = G_V - D_{u,i}$ as the updated view.

The failure of the computation $\mathsf{corr}((u,i))$ results in the failure of backward evaluation. Otherwise, we compute the updated source G'_S by removal of $\mathsf{corr}((u,i))$ in G_S. As mentioned in Section 2.1, we use a list of integers as the branch order, which forms a total order. Thus the relative position of the following (sibling) branches will not change if we delete one edge.

Finally, we return $\rho' = \rho[\$g \to G'_S]$, if $\mathscr{F}[\![e]\!]\rho' = G'_V$, or failure otherwise. One of the cases of failure here is that, the deletion of $\mathsf{corr}((u,i))$ in G_S changes the value of boolean expressions during evaluation of some condition clauses. It would change the evaluation into the other branch in such condition clauses, which is invalid according to our assumption that the value of boolean expressions are invariant during the view updates.

Another case of the violation of (*WPUTGET*) can be demonstrated as follows. For the query on the right, suppose we have graph [a, b, c] and remove a on the view. Then we trace back the edge a and delete a on the source. However, another forward transformation will result in an empty graph. Backward transfor-

```
srec((λ($l, $g).
        if $l = a then srec((λ($l, $g).
                        if $l = b then $db
                        else []), id)($db)
        else []), id)($db)
```

mation on that empty view will result in an empty source graph. Therefore, (*WPUT-GET*) is not satisfied. The checking rules out these cases. For the successful case, a query **srec**((λ($l, $g). **if** $l = a **then** $g **else** []), *id*)($*db*) with the same input graph will extract subgraph under edge labeled a, i.e., [b, c]. Then the deletion of c succeeds. The trace-based approach is unable to detect the change of reachability or boolean values caused by edge deletion. Nevertheless, such failures are distinguished from invalid updates on the query (which could be detected by the corr function). A compromise to detect such failures is to use a final check to conclude the failure of evaluation.

Hence, the (*WPUTGET*) property is straightforward (actually we are imposing stronger (*PUTGET*) property) from our final check, which leads to the well-behavedness. A natural question is to avoid the final check and the failure of evaluation. To do so, we provide an alternative solution with more refined semantics as well as a more detailed case study on the user's intention of updates in Appendix E.

5.2 Reflection of Subgraph Insertion

The method used in previous work of handling insertion in unordered graph [14] can be adopted here.

The insertion operation on the view is specified by a triple of a node v on the view and the position i where a graph is inserted, and the inserted graph G_{vins}. Then we first compute the corresponding source node u at which insertion of corresponding subgraph G_{sins} takes place, by the following function tr.

$\text{tr}(SrcID) = SrcID \quad \text{tr}(\text{RecN } \_\ v\ \_) = \text{tr}(v) \quad \text{tr}(\text{Code } \_\_) = FAIL$
$\text{tr}(\text{RecE } \_\ v\ \_\_) = tr(v) \quad \text{tr}(\text{RecD } \_\ v\ \_\_) = \text{tr}(v) \quad \text{tr}(\text{RecM } \_\_\_\_) = FAIL$

We find a graph G_{sins} connected to u and the corresponding position j at which G_{sins} is connected, by inversion computation on G'_V using the Universal Resolving Algorithm (URA) [1], where G'_V is obtained by adding ϵ-edge from v at position i to G_{vins}. Further, we conjecture that well-behavedness of the above insertion reflection can be directly derived from the soundness of URA.

6 Bidirectionalizing ϵ-Elimination Procedure

The idea of the ϵ-elimination procedure is to substitute the shortcuts within proper branches with new labeled edges. As mentioned in Section 2.1, we use a list of integers as a total order over proper branches on a node, so we are able to access one of the branches from the list by a list of integers (denoted $\tilde{p}$ below) rather than a single integer.

For a graph $G = (V, B, I) \in \mathcal{G}_Y^X$, the ϵ-elimination ϵ-elim(G) of G gets a graph $(V, B', I) \in \mathcal{G}_Y^X$. We let $|B'(v)| \stackrel{def}{=} |Pb(G, v)|$, and $B'(v).\tilde{p} \stackrel{def}{=} Pb(G, v).\tilde{p}.last$, where $Pb(G, v).\tilde{p}$ is a proper branch $p = (v \xrightarrow{\epsilon}_{i_0} \ldots v_n \rightarrow_{i_n})$ in $B'(v)$. Note that this procedure could fail. Since if there are some ϵ-edge cycles, it would lead to infinite number of proper branches, as mentioned in [11]. We use the effective procedure in [11] to decide the eliminability of ϵ-edges.

Next we will introduce a mechanism to reflect view updates over $G_V = \epsilon\text{-elim}(G_S)$ to the source graph G_S. We make use of the correspondence between the proper branches in the source graph G_S and the new branches in the view graph G_V.

1. For the in-place updates on an arbitrary edge $B'(v).\tilde{p} \in G_V$, the corresponding labeled edge on the source is $Pb(G_S, v).\tilde{p}.last$. We just apply the same in-place update operation on this edge.
2. For the deletion of an edge $B'(v).\tilde{p} \in G_V$, we update the source graph by deleting the corresponding edge $Pb(G_S, v).\tilde{p}.last$.
3. For the edge insertion, since the set of nodes V is the same on the view and the source, we just need to insert the edge to the same location on the source.

After we have done the update operation on the source, an ϵ-elimination procedure need to be applied again on the updated source. The reason is that, the updated labeled edge may be duplicated in different branches in the view. For example, in Figure 2, (b) is the graph obtained by ϵ-elim from (a). The a-labeled edge from nodes 5 to 6 in (a) is related with three a-labeled edges, respectively from node $1, 2$ and 5 to node 6 in (b). In case one of them is deleted, we need to delete the rest to retain the consistency. Thus, the corresponding edges of these proper branches on the view graph need to be updated to maintain the consistency between the source and the view. Therefore, the whole procedure would satisfy the well-behavedness. Moreover, three principles for rational updates in Section 3.1 are also implied from this constructive bidirectionalizing procedure.

7 Related Work

In the database community, bidirectional transformation has been discussed as view updating problem. Bancilhon and Spyratos proposed a general approach to this problem in [3]. The method was to define a constant complement view, from which the original database can be computed with the correlated information in the user-defined view. This method was applied to relational databases [10,17], as well as tree structures [18]. The constant complement view satisfies a very strong bidirectional properties at the sacrifice of the number of reflectable updates. It was too strong for our purpose, i.e., model transformation in software engineering, despite some particular applications [10]. A lot of linguistic approaches in the area of programming languages was proposed [8,9,5] for strings, trees and relational databases. However, they are difficult to be applied on graphs models due to the cycles and sharing nodes in graphs. In the context of software engineering, there has been several works on bidirectional (graph) transformation, one of the representative methods is the triple graphs grammars [20], which is powerful in graph transformation, however still not implemented on ordered graphs.

Another strongly related concept is structural recursion, which has been studied in the database and functional programming communities. It was adopted by graph transformations in UnCAL [6]. UnCAL enjoys many nice properties, including treating graphs as regular trees via bisimulation equivalence, functional style programming and reasoning, and termination property. Further, UnCAL [6] was separately extended to (1) give bidirectional semantics through traces [14], and to (2) introduce order among outgoing branches of a node (and thus order-aware bisimulation was also introduced), rearranging transformation of these sibling branches within structural recursion, and higher

order function [11], while keeping all the UnCAL features. The present work joins [3] the above two research lines – bidirectionalization from [14] and order introduction from [11]. However, this is not just a simple combination. In [14], we reused the Skolem terms in the bulk semantics for trace information to decompose target graphs, and two-stage bidirectionalization enabled to defer all the ε elimination at the end of the entire forward transformation. However, when sibling transformation is introduced in [11], to keep the structural recursion well-defined, ε have to be eliminated for every structural recursion application, which means we can no longer use two-stage bidirectionalization strategy. Therefore our bidirectional ε elimination is inserted along with every structural recursion application. Also, [14] did not discuss the diversity in the backward transformation reflection strategy as we did with respect to least-change principles [19], in particular, in the edge-deletion handling. Moreover, [14] did not have clear condition to keep WPutGet property in relation to branching behavior change, while we clarified using the classification of edges similarly to that in [13]. With respect to [11], higher order transformation was dropped, because we do not have clear semantics of the updates on functional values. We lose no expressive power relative to[14].

With respect to the notion of traces, apart from general studies like provenance traces [7], self-adjusting computation by Acar et al. [2] also leaves traces during computation, to make the recomputation with slightly modified input efficient, utilizing the locality of the modification. Dynamic dependency graph is used to record the dependencies among data and computations. In the present paper, we also leave trace during forward computation. Although our trace IDs are natural extensions of the Skolem functions, we could also consider them as run-time relation between operations and data, because we associate the code position of the operators with the edges and nodes in the arguments of the Skolem functions. However, we propagate the changes in the opposite direction, from output to input. Therefore we would not be able to reuse the same technique, at least in a direct manner. However, as we mention in the conclusion, with pure trace approach, we could have propagated changes directly through mapping between source and target edges, with branching behavior change checking. Then our approach would become closer to the self-adjusting computation approach in the sense that trace is used to exploit locality. Instead, we currently reverse all the computation regardless of the presence of the modification.

Handling ordered structure in the bidirectional transformation has been studied extensively through lenses [9], later improved to deal with rearranging updates through dictionary lenses [5] with key-based alignments, and the alignment strategies were generalized through matching lenses [4]. Although we support rearranging by the transformation, our update is operation based, so update should always be decomposed into edge renaming, edge deletion and subgraph insertion. Rearranging update cannot be fed to our backward transformation directly.

[3] Earlier attempt can be found in [12] as bidirectional language $UnCAL^O$. However, sibling transformation was not considered.

8 Conclusion

In this paper, we bidirectionalized transformation on ordered graphs, in which a three-stage bidirectionalizing strategy is provided. With trace-enriched semantics, we are able to reflect three kinds of view updates: inplace updates, edge deletion and subgraph insertion. In future work, we plan a practical implementation. Also, we would like to utilize the externalized traces defined in Appendix F that directly maps between source and target edges/nodes for reflecting other than inplace updates.

Acknowledgement. We thank the authors of the paper [12] on which our present work is built. We also thank Prof. Zhenjiang Hu for his valuable comments and suggestions.

References

1. S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In *The Essence of Computation*, pages 269–295, 2002.
2. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, Nov. 2006.
3. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.
4. D. M. J. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *ICFP'10*, pages 193–204. ACM, 2010.
5. A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *POPL '08*, pages 407–419, 2008.
6. P. Buneman, M. Fernandez, and D. Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.
7. J. Cheney, U. A. Acar, and A. Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.
8. K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT'09*, pages 260–283, 2009.
9. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17, 2007.
10. S. J. Hegner. Foundations of canonical update support for closed database views. In *ICDT'90*, pages 422–436. Springer, 1990.
11. S. Hidaka, K. Asada, Z. Hu, H. Kato, and K. Nakano. Structural recursion for querying ordered graphs. In *ICFP'13*, pages 305–318, Sept. 2013.
12. S. Hidaka, K. Asada, H. Kato, K. Nakano, and Z. Hu. Towards bidirectional transformations on ordered graphs. National Institute of Informatics, GRACE-TR-2011-07, Dec. 2011.
13. S. Hidaka, M. Billes, Q. M. Tran, and K. Matsuda. Trace-based approach to editability and correspondence analysis for bidirectional graph tansformations. Technical report, May 2015. `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/tesem.pdf`, extended summary appears in proc. of BX 2015.
14. S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP'10*, pages 205–216, 2010.
15. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. ICFP'97, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.
16. Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

17. J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM Transactions on Database Systems (TODS)*, 28(2):175–208, 2003.
18. K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP'07*, pages 47–58, 2007.
19. L. Meertens. Designing Constraint Maintainers for User Interaction. `http://www.kestrel.edu/home/people/meertens/`, 1998.
20. A. Schürr and F. Klar. 15 years of triple graph grammars. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, pages 411–425. Springer-Verlag, 2008.
21. J. Voigtländer. Bidirectionalization for free!(pearl). In *POPL'09*, pages 165–176, 2009.

A Graph Constructors

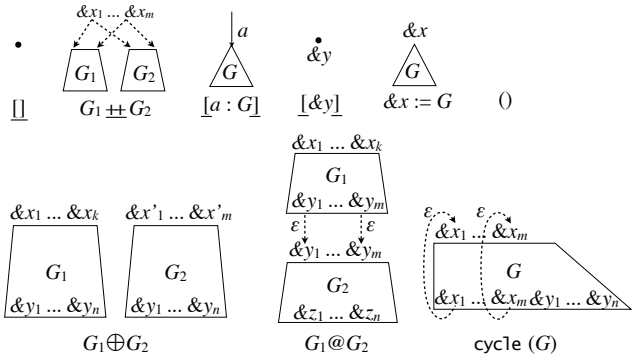

Fig. 8. Graph Constructors

We have nine graph constructors (Figure 8) as the operators to construct ordered graphs inductively.

$$
\begin{array}{lll}
G ::= & \underline{[]} & \{ \text{ single node graph } \} \\
| & G_1 \mathbin{\underline{++}} G_2 & \{ \text{ graph concatenation } \} \\
| & \underline{[}a : G\underline{]} & \{ \text{ an edge pointing to a graph } \} \\
| & \underline{[}\&y\underline{]} & \{ \text{ a node graph with output marker } \} \\
| & \&x := G & \{ \text{ label the root node with an input marker } \} \\
| & () & \{ \text{ empty graph } \} \\
| & G_1 \oplus G_2 & \{ \text{ disjoint graph union } \} \\
| & G_1 \mathbin{@} G_2 & \{ \text{ append of two graphs } \} \\
| & \mathbf{cycle}(G) & \{ \text{ graph with cycles } \}
\end{array}
$$

The single node graph constructor $\underline{[]}$ constructs a root-only graph. $G_1 \mathbin{\underline{++}} G_2$ joins two graphs by connecting every pair of roots from G_1 and G_2 to a new root with ϵ-edges. $\underline{[}a : G\underline{]}$ adds an a labeled edge pointing to the root of G. $\underline{[}\&y\underline{]}$ constructs a single node graph with an output marker $\&y$. $\&x := G$ associates an input marker $\&x$ to the root node of G. () constructs an empty graph with neither a node nor an edge. $G_1 \oplus G_2$ joins two graphs by using a componentwise (V, B, I) union. Note that $\mathbin{\underline{++}}$ unifies input nodes while $\oplus$ does not. $G_1 \mathbin{@} G_2$ joins two graphs by connecting the output marker branches of G_1 with the corresponding input nodes of G_2 with ϵ edges. **cycle**(G) connects the output marker branches of G with the corresponding input nodes with ϵ edges, which would form cycles.

For the set of input nodes of constructed graphs, the set of input markers of G_1 and G_2 in $G_1 \mathbin{\underline{++}} G_2$ should coincide, and matching input nodes are bundled with ϵ edges.

Whereas the set of input markers of G_1 and G_2 in $G_1 \oplus G_2$ are disjoint, and their union becomes the new set of input markers. For $G_1 \mathbin{@} G_2$, the input nodes of the first operand becomes the new input nodes and the input nodes of the second operands are connected with the output marker branches if exists.

B Type Definition and Typing Rules of λ_{FG}

The type definition in λ_{FG} is given as follows:

$$\begin{array}{rll} \sigma ::= \sigma \to \sigma \mid \sigma + \sigma \mid \sigma \times \sigma & \{ \text{ function, coproduct, product types } \} \\ \quad \mid \ \mathbf{List}(\sigma) \mid \mathbf{Bool} & \{ \text{ list and boolean types } \} \\ \quad \mid \ \mathbf{Label} \mid \mathbf{G}_Y^X & \{ \text{ label and graph types } \} \end{array}$$

The typing rules for graph-related expressions in λ_{FG} are given as follows:

$$\frac{(a \in \mathcal{L})}{\vdash a : \mathbf{Label}} \qquad \frac{\vdash e_1 : \mathbf{Label} \quad \vdash e_2 : \mathbf{Label}}{\vdash e_1 = e_2 : \mathbf{Bool}}$$

$$\frac{}{\vdash \underline{[]} : \mathbf{G}_Y} \qquad \frac{\vdash e_1 : \mathbf{G}_Y^X \quad \vdash e_2 : \mathbf{G}_Y^X}{\vdash e_1 \mathbin{\underline{+\!+}} e_2 : \mathbf{G}_Y^X} \qquad \frac{\vdash e_1 : \mathbf{Label} \quad \vdash e_2 : \mathbf{G}_Y}{\vdash \underline{[}e_1 : e_2\underline{]} : \mathbf{G}_Y}$$

$$\frac{(\&y \in Y)}{\vdash \underline{[}\&y\underline{]} : \mathbf{G}_Y} \qquad \frac{\vdash e : \mathbf{G}_Y}{\vdash \&x := e : G_Y^{\{\&x\}}} \qquad \frac{}{\vdash () : \mathbf{G}_Y^{\emptyset}}$$

$$\frac{\vdash e_1 : \mathbf{G}_Y^{X_1} \quad \vdash e_2 : \mathbf{G}_Y^{X_2} (X_1 \cap X_2 = \emptyset)}{\vdash e_1 \oplus e_2 : \mathbf{G}_Y^{X_1 \cup X_2}} \qquad \frac{\vdash e_1 : \mathbf{G}_Y^X \quad \vdash e_2 : \mathbf{G}_Z^Y}{\vdash e_1 @ e_2 : \mathbf{G}_Z^X}$$

$$\frac{\vdash e : \mathbf{G}_{X \cup Y}^X (X \cap Y = \emptyset)}{\vdash \mathbf{cycle}(e) : \mathbf{G}_Y^X} \qquad \frac{\vdash e : \mathbf{G}_Y^X}{\vdash \mathbf{isEmpty}(e) : \mathbf{Bool}}$$

$$\frac{\vdash e : \mathbf{Label} \times \mathbf{G}_Y \to \mathbf{G}_Z^Z \quad \vdash d : \mathbf{List}(\mathbf{G}_{Z \times \alpha}^Z + \mathbf{G}_{Z \times Y}^Z) \to \mathbf{G}_{Z \times \alpha + Z \times Y}^Z}{\vdash \mathbf{srec}(e, d) : \mathbf{G}_Y^X \to \mathbf{G}_{Z \times Y}^{Z \times X}}$$

We have omitted the rules for lambda terms, which are standard. Note that for rules like $e_1 \mathbin{\underline{+\!+}} e_2$, we do not require the output markers of e_1 and e_2 to be identical. Here, we just choose a set Y, which is a superset of the set of output markers of both expressions. Moreover, for rules like $\underline{[]}$, there is exactly the default marker & in the set X, which is why it is not mentioned.

C Formal Semantics of Traceable Forward Evaluation

In this section we present the complete semantics of traceable forward evaluation that generates traceable views, in a similar way to the semantics of $UnCAL^O$[12]. The forward semantics $\mathscr{F}[\![e^p]\!]\rho$ for a λ_{FG} expression e and a variable binding environment ρ is inductively defined on the structure of e. We complete the semantics for λ_{FG} here together with the contents in Section 4.

In addition, we replace the composition of markers in $X \times Z$ by a monoid $(., \&)$, with & as the identity, i.e., $\&.\&x = \&x.\& = \&x$. The associativity of the monoid structure is needed here. In particular, we let $\{\&\}.Z = Z.\{\&\} = Z$ for any Z, and we call "." the Skolem function.

Graph Constructors Expressions

The enriched forward semantics of the expression $e_1 \mathbin{\underline{+\!+}} e_2$ is defined as below. Note that $\underline{+\!+}$ retains the order of the new branches when they link the input nodes.

$$
\begin{aligned}
&\mathscr{F}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \mathbin{\underline{+\!+}^p} \mathscr{F}[\![e_2]\!]\rho \\
&\quad \text{where} \quad G_1 \mathbin{\underline{+\!+}^p} G_2 = (V_1 \cup V_2 \cup V', B_1 \cup B_2 \cup B', I') \\
&\qquad (V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2 \\
&\qquad\qquad M = \mathbf{inMarker}(G_1) = \mathbf{inMarker}(G_2) \\
&\qquad\qquad V' = \{\mathsf{Code}\ p\ \&m \mid \&m \in M\} \\
&\qquad\qquad B' = \{\mathsf{Code}\ p\ \&m \mapsto [\mathsf{Edge}(\epsilon, I_1(\&m)), \mathsf{Edge}(\epsilon, I_2(\&m))] \mid \&m \in M\} \\
&\qquad\qquad I' = \{\&m \mapsto \mathsf{Code}\ p\ \&m \mid \&m \in M\}
\end{aligned}
$$

where $\underline{+\!+}^p$ is a union operator for two graphs concerning position p. We write $\mathbf{inMarker}(G)$ and $\mathbf{outMarker}(G)$ to denote the set of input and output markers in a graph G, respectively.

$\mathbf{e_1} \oplus \mathbf{e_2}$ is a componentwise union operator like $\underline{+\!+}$, except that no ϵ-edges are involved to concatenate two subgraphs.

$$
\begin{aligned}
&\mathscr{F}[\![(e_1 \oplus e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \oplus \mathscr{F}[\![e_2]\!]\rho \\
&\text{where}\, G_1 \oplus G_2 = (V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2) \\
&\quad (V_1, B_1, I_1) = G_1; (V_2, B_2, I_2) = G_2
\end{aligned}
$$

The forward semantics for the **empty graph constructor** and the **constant marker graph** are defined as follows.

$$
\begin{aligned}
&\mathscr{F}[\![()^p]\!]\rho \quad\ = (\{\}, \{\}, \{\}) \\
&\mathscr{F}[\![\underline{[}\&m\underline{]}^p]\!]\rho = (\{\mathsf{Code}\ p\}, \{\mathsf{Code}\ p \mapsto [\mathsf{Outm}(\&m)]\}, \{\& \mapsto \mathsf{Code}\ p\})
\end{aligned}
$$

The next two constructors **prepend labeled edges and markers** with a graph respectively.

$$
\begin{aligned}
&\mathscr{F}[\![\underline{[}l : e\underline{]}^p]\!]\rho = (\{\mathsf{Code}\ p\} \cup V, \{\mathsf{Code}\ p \mapsto [\mathsf{Edge}(l\rho, I(\&))]\} \cup B, \{\& \mapsto \mathsf{Code}\ p\}) \\
&\text{where}\, (V, B, I) = \mathscr{F}[\![e]\!]\rho
\end{aligned}
$$

$$
\begin{aligned}
&\mathscr{F}[\![(\&m := e)^p]\!]\rho = (\&m := \mathscr{F}[\![e]\!]\rho) \\
&\text{where}\, (\&m := G) = (V, B, I') \\
&\qquad (V, B, I) \quad = G \\
&\qquad I' \qquad\quad = \{\&m.\&x \mapsto v \mid (\&x \mapsto v) \in I\}
\end{aligned}
$$

Here "." is the Skolem function introduced before.

$\mathbf{e_1}$ @ $\mathbf{e_2}$ appends two graphs by connecting the output marker branches of the left operand and corresponding input nodes of the right operand with ϵ-edges.

$$
\begin{aligned}
&\mathscr{F}[\![(e_1 \mathbin{@} e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho \mathbin{@^p} \mathscr{F}[\![e_2]\!]\rho \\
&\text{where}\ G_1 \mathbin{@^p} G_2 = (V_1 \cup V_2, B_1' \cup B_2, I_1) \\
&\qquad (V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2 \\
&\qquad\qquad B_1' = \{u \mapsto \left[\begin{array}{l} x.i = \mathsf{Edge}(l, v)\ \Rightarrow x.i \\ \quad = \mathsf{Outm}(\&m) \Rightarrow \mathsf{Edge}(\epsilon, I_2(\&m)) \end{array}\right]_{i \in |x|} \mid (u \mapsto x) \in B_1\}
\end{aligned}
$$

cycle(e) links the output marker branches with the corresponding input nodes of a graphs with ϵ-edges and forms a cycle, and its forward semantics is defined as follows.

$$\mathscr{F}[\![(\mathbf{cycle}(e))^p]\!]\rho = \mathbf{cycle}^p(\mathscr{F}[\![e]\!]\rho)$$
$$\text{where } \mathbf{cycle}^p(G) = (V, B', I)$$
$$(V, B, I) = G$$
$$B' = \{u \mapsto b(x) \mid (u \mapsto x) \in B\}$$
$$b(x) = \left[\begin{array}{l} s = \mathsf{Edge}(l, v) \Rightarrow s \\ \quad = \mathsf{Outm}(\&m) \wedge \&m \in \mathrm{dom}(I) \Rightarrow \mathsf{Edge}(\epsilon, I(\&m)) \\ \quad = \mathsf{Outm}(\&m) \wedge \&m \notin \mathrm{dom}(I) \Rightarrow s \end{array} \middle| \; s \in x \right]$$

Labels In λ_{FG} the types of labels and boolean values are formally declared. The forward semantics for label and label equality are

$$\mathscr{F}[\![a^p]\!]\rho = a\rho \quad (a \in \mathcal{L})$$
$$\mathscr{F}[\![(l_1 = l_2)^p]\!]\rho = l_1\rho = l_2\rho$$

Emptiness Checking The forward semantics of emptiness checking function is defined as

$$\mathscr{F}[\![(\mathbf{isEmpty}(e))^p]\!]\rho = \mathbf{isEmpty}(\mathscr{F}[\![e]\!]\rho)$$

Function on Lists There is a formal method for bidirectional transformation on lists [9], so we will not include the details.

λ Expressions We restrict the occurrence of λ expressions only to applied forms, so the forward semantics of the application is determined by the forward semantics of e_1 with environment extended with the bindings using the result of the forward evaluation of e_2. We allow tuple patterns in the arguments to cope with functions taking multiple arguments like $\underline{\hat{+}} = \lambda(\$x, \$y).\$y \underline{+} \$x$ in Example 2.

$$\mathscr{F}[\![(\lambda(\$x_1, .., \$x_n).e_1\ e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho'$$
$$\text{where } v_1, .., v_n = \mathscr{F}[\![e_2]\!]\rho$$
$$\rho' = \rho \cup \{\$x_1 \mapsto v_1\} \cup ... \cup \{\$x_n \mapsto v_n\}$$

We provide the semantics only for occurrences in applied forms.

Products As mentioned in Section 4, it is also essential to define the forward semantics of products of expressions.

$$\mathscr{F}[\![(e_1, e_2)^p]\!]\rho = (\mathscr{F}[\![e_1]\!]\rho, \mathscr{F}[\![e_2]\!]\rho)$$

Projections $\mathbf{pr_l}$ and **$\mathbf{pr_r}$** are syntactic sugars of $\lambda(\$x, \$y).\$x$ and $\lambda(\$x, \$y).\$y$ and the semantics are provided through that of λ expression.

foldr Forward semantics of **foldr** is standard, while backward semantics decompose the view based on the binary operators $\odot$ to feed the backward evaluation of e and e_a.

$$\odot \in \{\underline{+\!+}, @, \mathbf{cons}, \}$$
$$\mathscr{F}[\![\mathbf{foldr}(\odot, e)(e_\mathrm{a})]\!]_\rho = f\ l$$
$$\text{where } l = \mathscr{F}[\![e_\mathrm{a}]\!]_\rho$$
$$f\ \mathbf{nil} = \mathscr{F}[\![e]\!]_\rho$$
$$f\ \mathbf{cons}(x, xs) = x \odot (f\ xs)$$

Structural Recursion The forward semantics of a structural recursion in λ_{FG} is formally given by *bulk semantics*.

To have a clear insight of the bulk semantics of structural recursion function, we recall the type rules for all component.

$e_\mathrm{b} : \mathbf{Label} \times \mathbf{G}_Y \to \mathbf{G}_Z^Z$, $d : \mathbf{List}(\mathbf{G}_{Z\times\alpha}^Z + \mathbf{G}_{Z\times Y}^Z) \to \mathbf{G}_{Z\times\alpha+Z\times Y}^Z$. $\mathscr{F}[\![(\mathbf{srec}(e_\mathrm{b}, d)(e_\mathrm{a}))^p]\!]\rho = G_\mathrm{srec}$ is a $\mathbf{G}_Y^X \to \mathbf{G}_{Z\times Y}^{Z\times X}$ function

$$\text{where } G_\mathrm{src} = \mathscr{F}[\![e_\mathrm{a}]\!]\rho, \quad Z = \mathbf{inMarker}(e_\mathrm{b})$$
$$(V_\mathrm{src}, B_\mathrm{src}, I_\mathrm{src}) = G_\mathrm{src}\ (G_\mathrm{src} \text{ has no } \epsilon\text{-edges}),$$
$$G|_v = (V_\mathrm{src}, B_\mathrm{src}, \{\& \mapsto v\})$$

It can be formally defined by three stages. For bisimulation genericity, we require that the input graph in this stage has no ϵ-edges (this is guaranteed by an ϵ-elimination procedure stated in Section 6).

The forward evaluation does the following mapping operation: $B_{\text{src}} \to B_{\text{EMap}} \to B_{\text{DMap}} \to G_{\text{srec}}$, which are implemented by EMap, DMap and Merge.

1. First apply edge mappings using e_{b}. The edge mapping function
   EMap : $(\mathbf{V} \to \mathbf{List}(\mathbf{Label} \times \mathbf{V} + \mathbf{Y})) \to (\mathbf{V} \to \mathbf{List}(\mathbf{G}^Z_{Z\times V} + \mathbf{G}^Z_{Z\times Y}))$ is defined by
   $\text{EMap}(B_{\text{src}}) = B_{\text{EMap}}$, **where** :
   $$B_{\text{EMap}} = \{u \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(l, v) \Rightarrow \text{ELabel}(\mathscr{F}[\![e_{\text{b}}]\!]\rho_{u,i}, v, u, i) \\ \rho_{u,i} = \rho \cup \{\$l \mapsto l, \$g \mapsto G|_v\} \\ x.i = \mathsf{Outm}(\&y) \Rightarrow \text{Mlabel}(\&y, u, i) \end{bmatrix}_{i\in|x|} \mid (u \mapsto x) \in B_{\text{src}}\}$$
   B_{EMap} collects the set of branches resulted from the forward evaluation of e_{b} in each sub-environment $\rho_{u,i}$ and annotate each resulted branches with some trace information. The sub-environment $\rho_{u,i}$ is extended by adding the label l and v-rooted subgraph $G|_v$.
   The branches are annotated by either ELabel or Mlabel, for edge or marker branches respectively.
   ELabel : $\mathbf{G}^Z_Z \times \mathbf{V} \times \mathbf{V} \times \mathbf{Num} \to \mathbf{G}^Z_{Z\times V}$ adds the trace information RecE to a subgraph G resulted from e_{b}. Note that a pair (u, i) uniquely identifies an edge in the source graph.
   $\text{ELabel}(G, v, u, i) = G_{\text{E}}$
   where,
   $G = (V, B, I)$, $G_{\text{E}} = (V_{\text{E}}, B_{\text{E}}, I_{\text{E}})$
   $V_{\text{E}} = \{\mathsf{RecE}\ p\, w\, u\, i \mid w \in V\}$
   $$B_{\text{E}} = \{\mathsf{RecE}\ p\, w\, u\, i \mapsto \begin{bmatrix} x'.j = \mathsf{Edge}(l, w') \Rightarrow \mathsf{Edge}(l, \mathsf{RecE}\ p\, w'\, u\, i) \\ x'.j = \mathsf{Outm}(\&m) \Rightarrow \mathsf{Outm}(\&m, v) \end{bmatrix}_{j\in|x|} \mid (w \mapsto x) \in B, \&m \in Z\}$$
   $I_{\text{E}} = \{\&m \mapsto \mathsf{RecE}\ p\, w\, u\, i \mid (\&m \mapsto w) \in I\}$
   Mlabel : $\mathbf{Y} \times \mathbf{V} \times \mathbf{Num} \to \mathbf{G}^Z_{Z\times Y}$ adds the trace information RecM to the marker branch $\&y$ identified by the pair (u, i).
   $\text{Mlabel}(\&y, u, i) = G_{\text{M}}$
   where
   $G_{\text{M}} = (V_{\text{M}}, B_{\text{M}}, I_{\text{M}})$
   $V_{\text{M}} = \{\mathsf{RecM}\ p\, \&m\, u\, i \mid \&m \in Z\}$
   $B_{\text{M}} = \{\mathsf{RecM}\ p\, \&m\, u\, i \mapsto [\mathsf{Outm}(\&m, \&y)] \mid \&m \in Z\}$
   $I_{\text{M}} = \{\&m \mapsto \mathsf{RecM}\ p\, \&m\, u\, i \mid \&m \in Z\}$
   Hence, we get B_{EMap} as the result of the edge mappings in this stage.
2. Then apply list manipulation function d. It is applied by the function Dmap : $(\mathbf{V} \to \mathbf{List}(\mathbf{G}^Z_{Z\times V} + \mathbf{G}^Z_{Z\times Y})) \to (\mathbf{B} \to \mathbf{G}^Z_{Z\times V+Z\times Y})$,
   $\text{Dmap}(B_{\text{EMap}}) = B_{\text{Dmap}}$
   where
   $B_{\text{Dmap}} = \{u \mapsto \left[\text{DLabel}(\mathscr{F}[\![d]\!]\rho_u, u)\rho_u = \rho \cup \{\$b \mapsto B_{\text{EMap}}(u)\}\right] \mid u \in V\}$
   Here $\$b$ is a variable binds the branch B_{EMap} for node u generated in the previous step, and are supposed to be referred to during the evaluation of d for operations on these branches like permutation or selection.

The function DLabel : $\mathbf{G}^{Z}_{Z\times V+Z\times Y} \times \mathbf{V} \to \mathbf{G}^{Z}_{Z\times V+Z\times Y}$ adds the RecD information to every node in the graphs resulted from list manipulation d. It uses the root node u of the list to identify the original list of each components in the graph.
$\text{DLabel}(G, u) = G_{\text{D}}$
Let $G = (V, B, I)$, $G_{\text{D}} = (V_{\text{D}}, B_{\text{D}}, I_{\text{D}})$
where,
$V_{\text{D}} = \{\mathsf{RecD}\ p\, v\, u \mid v \in V\}$
$B_{\text{D}} = \{\mathsf{RecD}\ p\, v\, u \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(l, w) \Rightarrow \mathsf{Edge}(l, \mathsf{RecD}\ p\, w\, u) \\ x.i = \mathsf{Outm}(\&m, m') \Rightarrow x.i \end{bmatrix}_{i\in|x|} \mid (v \mapsto x) \in B\}$
$I_{\text{D}} = \{\&m \mapsto \mathsf{RecD}\ p\, v\, u \mid (\&m \mapsto v) \in I\}$

3. Finally we merge all the nodes and subgraphs in B_{Dmap} with ϵ-edges, using the Merge : $\mathbf{V} \to \mathbf{G}^{Z}_{Z\times V+Z\times Y} \to \mathbf{G}^{Z\times X}_{Z\times Y}$.
$\text{Merge}(B_{\text{Dmap}}) = G_{\text{srec}}$
Let $G_{\text{srec}} = (V_{\text{srec}}, B_{\text{srec}}, I_{\text{srec}})$,
we now explain the way of generating each components.
The set of nodes $V_{\text{srec}} = \text{Nlabel}(V_{\text{src}}) \cup \text{VMerge}(B_{\text{Dmap}})$, where Nlabel maps every node from the source graph to the corresponding set of nodes used to connect each subgraphs in B_{Dmap}, it annotates each node with RecN information identifying its original node.
$\text{Nlabel}(V_{\text{src}}) = \{\mathsf{RecN}\ p\, u\, \&m \mid u \in V_{\text{src}}, \&m \in Z\}$
And VMerge collects all the nodes from the result of d function, which is defined as follows.
$\text{VMerge}(B_{\text{Dmap}}) = \{u \mid v \in V_{\text{src}},\ (v \mapsto G_{\text{D}}) \in B_{\text{Dmap}},\ u \in G_{\text{D}}.V\}$
The set of branches $B_{\text{srec}} = \text{NMerge}(B_{\text{Dmap}}) \cup \text{BMerge}(B_{\text{Dmap}})$. The function NMerge adds ϵ-edges to connect the RecN nodes obtained from Nlabel with the input nodes of the corresponding subgraph.
$\text{NMerge}(B_{\text{Dmap}}) = \{\mathsf{RecN}\ p\, v\, \&m \mapsto [\mathsf{Edge}(\epsilon, G_{\text{D}}.I(\&m))] \mid v \in V_{\text{src}},\ (v \mapsto G_{\text{D}}) \in B_{\text{Dmap}},\ \&m \in Z\}$
And $\text{BMerge}(B_{\text{Dmap}})$ collects the set of branches for the resulted graph by connecting all the subgraphs using ϵ-edges.
$\text{BMerge}(B_{\text{Dmap}}) = \{u \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(l, w) \Rightarrow x.i \\ x.i = \mathsf{Outm}(\&m, w),\ w \in V_{\text{src}} \Rightarrow \mathsf{Edge}(\epsilon, \mathsf{RecN}\ p\, w\, \&m) \\ x.i = \mathsf{Outm}(\&m, \&y) \Rightarrow x.i \end{bmatrix}_{i\in|x|} \mid$
$v \in V_{\text{src}},\ (v \mapsto G_{\text{D}}) \in B_{\text{Dmap}},\ (u \mapsto x) \in G_{\text{D}}.B\}$
Note that in this step an output marker branch in G_{D} becomes an ϵ-edge to the corresponding RecN node if it is of the form $(\&m, u)$, otherwise, it remains unchanged.
A remark is that the marker branches generated by the function ELabel finally become ϵ-edges, and those generated by function MLabel finally remain to be output markers for the result graph.
Finally, the input marker I_{srec} are assigned by Imap : $(\mathbf{X} \to \mathbf{V}) \to (\mathbf{Z} \times \mathbf{X} \to \mathbf{V})$, where
$\text{Imap}(I_{\text{src}}) = I_{\text{srec}}$ where,
$I_{\text{srec}} = \{(\&m, x) \mapsto \mathsf{RecN}\ p\, v\, \&m \mid \&m \in Z,\ (\&x \mapsto v) \in I_{\text{src}}\}$

In the above comprehension of the form $\{e \mid g\}$, we essentially wrap the information appeared in the g parts with the constructors defined in Section 4.1.

D Formal Semantics of Backward Evaluation for In-place Updates

We now proceed to present the complete backward semantics. $\underline{[]}$, $\&y$ and $()$ construct constant graphs in the forward evaluation. Therefore, for the backward evaluation, they accept no modification on the view.

$\mathscr{B}[\![\underline{[]}^p]\!](\rho, G') = \rho$ if $G' = \mathscr{F}[\![\underline{[]}^p]\!]\rho$

$\mathscr{B}[\![\underline{[\&m]}^p]\!](\rho, G') = \rho$ if $G' = \mathscr{F}[\![\underline{[\&m]}^p]\!]\rho$

$\mathscr{B}[\![()^p]\!](\rho, G') = \rho$ if $G' = \mathscr{F}[\![()^p]\!]\rho$

$\underline{[}\mathbf{l} : \mathbf{e}\underline{]}$ Its backward evaluation detaches the (possibly modified) edge from the top of the modified graph. Other modification on the graph is reflected to the other operand G_2 (as G_2')

$$\begin{aligned}&\mathscr{B}[\![\underline{[}l : e\underline{]}^p]\!](\rho, G') = \mathscr{B}[\![l]\!](\rho, a') \uplus_\rho \mathscr{B}[\![e]\!](\rho, G_2')\\ &\text{where } a &= l\rho\\ &\quad G_2 &= \mathscr{F}[\![e]\!]\rho\\ &\quad (a', G_2') &= \mathsf{decomp}_{\underline{[a:^p G_2]}}(G')\end{aligned}$$

Here, the decomposition function is defined as follows

$$\begin{aligned}&\mathsf{decomp}_{\underline{[a_1:^p G_2]}}(G') = \quad (a_1', (V' \setminus \{r'\}, B' \setminus \{r' \mapsto \zeta'\}, \{\& \mapsto v\}))\\ &\text{where } (V_2, B_2, \{\& \mapsto v\}) = G_2\\ &\quad (V', B', \{\& \mapsto r'\}) = G'\\ &\quad \zeta' = \text{the unique branch in } B' \text{ of the form } [\mathsf{Edge}(a_1', v)]\end{aligned}$$

.

The modified view G' is decomposed into its unique root branch $\zeta' = [\mathsf{Edge}(a_1', v)]$ from the original root r' and the rest of the graph rooted at v. If G' has more than one branches from the root node or the new root v does not match the root node of the original result G_2, the backward evaluation fails.

$\mathbf{e_1} \mathbin{\underline{+\!+}} \mathbf{e_2}$ we first decompose the updated graph G' and apply backward transformation on each of the subexpressions e_1 and e_2 using the fragment graphs. We have the following definition, given $\mathsf{decomp}_{G_1 \mathbin{\underline{+\!+}} G_2}(G')$ as the decomposition of the graph G', then

$$\begin{aligned}&\mathscr{B}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G_1') \uplus_\rho \mathscr{B}[\![e_2]\!](\rho, G_2')\\ &\quad \text{where } G_1 = \mathscr{F}[\![e_1]\!]\rho, \quad G_2 = \mathscr{F}[\![e_2]\!]\rho \quad (G_1', G_2') = \mathsf{decomp}_{G_1 \mathbin{\underline{+\!+}^p} G_2}(G')\end{aligned}$$

decomp is defined as below. $G_1 \setminus G_2$ for $G_1 = (V_1, B_1, I_1)$ and $G_2 = (V_2, B_2, I_2)$ denotes $(V_1 \setminus V_2, B_1 \setminus B_2, I_1 \setminus I_2)$, where $B_1 \setminus B_2$ and $I_1 \setminus I_2$ respectively removes the bindings in B_2 and I_2 from those in B_1 and I_1. Simple union of graphs $G_1 \cup G_2$ is defined by $(V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2)$. $\mathsf{reachable}$ collects all the parts reachable from input nodes, while $\mathsf{unreachable}$ collects the rest. $\mathsf{xreachable}$ restores complete graphs by restoring the unreachable parts. Note that the unreachable parts restored from the original graph can not be decomposed or modified in the reachable parts of the view.

$$\begin{aligned}&\mathsf{decomp}_{G_1 \mathbin{\underline{+\!+}^p} G_2}(G') = (\mathsf{xreachable}(G_1'', G_1), \mathsf{xreachable}(G_2'', G_2))\\ &\quad \text{where } (V', B', I') = G' \quad (V_i, B_i, I_i) = G_i\\ &\qquad G_i'' = \mathsf{reachable}((V', B', I_i))\\ &\quad \text{satisfying } M = \mathrm{dom}(I_1) = \mathrm{dom}(I_2)\\ &\qquad\qquad \forall \&m \in M,\ \mathsf{Edge}(\epsilon, v') \in B'(I'(\&m)) : (\&m \mapsto v') \in I_1 \cup I_2\\ &\mathsf{unreachable}(G) = G \setminus \mathsf{reachable}(G)\\ &\mathsf{xreachable}(G'', G) = \mathsf{reachable}(G'') \cup \mathsf{unreachable}(G)\end{aligned}$$

$\mathbf{e_1} \oplus \mathbf{e_2}$ It is like $\underline{+\!+}$, except that no ϵ-edge is involved.

$$\mathscr{B}[\![(e_1 \oplus e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G'_1) \uplus_\rho \mathscr{B}[\![e_2]\!](\rho, G'_2)$$
$$\text{where } G_1 = \mathscr{F}[\![e_1]\!]\rho$$
$$G_2 = \mathscr{F}[\![e_2]\!]\rho$$
$$(G'_1, G'_2) = \mathsf{decomp}_{G_1 \oplus^p G_2}(G')$$
$$\mathsf{decomp}_{G_1 \oplus^p G_2}(G') = (\mathsf{xreachable}(G''_1, G_1), \mathsf{xreachable}(G''_2, G_2))$$
$$\text{where } (V', B', I') = G' \quad (V_i, B_i, I_i) = G_i$$
$$G''_i = \mathsf{reachable}((V', B', I_i))$$

The decomposition function is almost the same as $\mathsf{decomp}_{G_1 \underline{+\!+}^p G_2}(G')$. The only difference is that it excludes the **satisfying** condition.

$\mathbf{e_1}$ @ $\mathbf{e_2}$ Because of unmatched I/O nodes, it may introduce unreachable part in the second argument during forward evaluation. So the backward evaluation carefully leaves those parts untouched to avoid unnecessary failure because of inconsistency, and those parts are within the ordinary computation (computation on reachable parts) before discarding by the @ operator.

$$\mathscr{B}[\![(e_1 \mathbin{@} e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G'_1) \uplus_\rho \mathscr{B}[\![e_2]\!](\rho, G'_2)$$
$$\text{where } G_1 = \mathscr{F}[\![e_1]\!]\rho$$
$$G_2 = \mathscr{F}[\![e_2]\!]\rho$$
$$(G'_1, G'_2) = \mathsf{decomp}_{G_1 @^p G_2}(G')$$
$$\mathsf{decomp}_{G_1 @^p G_2}(G') = (\mathsf{xreachable}(G''_1, G_1), \mathsf{xreachable}(G''_2, G_2))$$
$$\text{where } (V', B', I') = G' \quad (V_i, B_i, I_i) = G_i$$
$$B'' = \{u \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(\epsilon, v) \wedge u \in V_1 & \\ \wedge B_1(u).i = \mathsf{Outm}(\&m) & \\ \wedge (\&m \mapsto v) \in I_2 & \Rightarrow \mathsf{Outm}(\&m) \\ \text{otherwise} & \Rightarrow x.i \end{bmatrix}_{i \in |x|}$$
$$\mid (u \mapsto x) \in B'\}$$
$$G''_i = \mathsf{reachable}((V', B'', I_i))$$

$\&m := \mathbf{e}$ It "peels off" the marker on the left hand side from each of the input markers in G' at the front.

$$\mathscr{B}[\![(\&m := e)^p]\!](\rho, G') = \mathscr{B}[\![e]\!](\rho, G'_1)$$
$$\text{where } (V', B', I') = G'$$
$$I'_1 = \{(\&x \mapsto v) \mid (\&m.\&x \mapsto v) \in I'\}$$
$$G'_1 = (V', B', I'_1)$$

cycle(e) It removes the ϵ-edges introduced in the forward evaluation and restores the original output markers.

$$\mathscr{B}[\![(\mathbf{cycle}(e))^p]\!](\rho, G') = \mathscr{B}[\![e]\!](\rho, G'_2)$$
$$\text{where } (V', B', I') = G' \quad (V, B, I) = \mathscr{F}[\![e]\!]\rho$$
$$B_{\text{cycle}} = \{u \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(\epsilon, v) & \\ \wedge B(u).i = \mathsf{Outm}(\&m) & \Rightarrow \mathsf{Outm}(\&m) \\ \text{otherwise} & \Rightarrow x.i \end{bmatrix}_{i \in |x|}$$
$$\mid (u \mapsto x) \in B'\}$$
$$G'_2 = (V, B_{\text{cycle}}, I)$$

Labels A label constant and label equality accepts no modification.

$$\mathscr{B}[\![a]\!](\rho, a') = \rho \quad \text{if } a' = a$$
$$\mathscr{B}[\![(l_1 = l_2)]\!](\rho, e') = \rho \quad \text{if } e' = \mathscr{F}[\![(l_1 = l_2)]\!]\rho$$

Emptiness A backward evaluation of an emptiness checking is defined by

$\mathscr{B}[\![\textbf{isEmpty}(e)]\!](\rho, e') = \rho \quad \textbf{if } e' = \mathscr{F}[\![\textbf{isEmpty}(e)]\!]\rho$

As the reason explained before, we simply do not accept any change of boolean values in the backward evaluation.

λ Expressions

The backward evaluation of e_1 generates new bindings ρ'_1 including the ones for the bound variables. These bindings are used for the backward evaluation of e_2 to reproduce the updated environment ρ'_2. The overall result will be the merger of the updated bindings. Note that the bindings for the bound variables should be removed before merging as we do for the semantics of **srec**.

$$\begin{aligned}
&\mathscr{B}[\![(\lambda(\$x_1, .., \$x_n).e_1\ e_2)^p]\!](\rho, G') \\
&= (\rho'_1 \setminus \{\$x_1 \mapsto _\} \setminus ... \setminus \{\$x_n \mapsto _\}) \uplus_\rho \rho'_2 \\
&\text{where } v_1, .., v_n = \mathscr{F}[\![e_2]\!]\rho \\
&\qquad \rho' = \rho \cup \{\$x_1 \mapsto v_1\} \cup ... \cup \{\$x_n \mapsto v_n\} \\
&\qquad \rho'_1 = \mathscr{B}[\![e_1]\!](\rho', G') \\
&\qquad \rho'_2 = \mathscr{B}[\![e_2]\!](\rho, (\rho'_1(\$x_1), .., \rho'_1(\$x_n)))
\end{aligned}$$

Products The backward evaluation of products simply unifies the result of backward evaluation on its subexpressions.

$$\begin{aligned}
&\mathscr{B}[\![(e_1, e_2)^p]\!](\rho, (G'_1, G'_2)) = \rho'_1 \uplus_\rho \rho'_2 \\
&\text{where } \rho'_1 = \mathscr{B}[\![e_1]\!](\rho, G'_1) \\
&\qquad \rho'_2 = \mathscr{B}[\![e_2]\!](\rho, G'_2)
\end{aligned}$$

foldr

$$\begin{aligned}
&\mathscr{B}[\![\textbf{foldr}(\odot, e)(e_\mathrm{a})]\!](\rho, G') = \mathscr{B}[\![e]\!](\rho, \bar{g}'_n) \uplus_\rho \mathscr{B}[\![e_\mathrm{a}]\!](\rho, [g'_1 \ldots g'_n]) \\
&\text{where} \quad [g_1, \ldots, g_n] = \mathscr{F}[\![e_\mathrm{a}]\!]\rho \\
&\qquad \bar{g}_n = \mathscr{F}[\![e]\!]\rho \\
&\qquad \bar{g}_k = g_k \odot \bar{g}_{k+1}, 1 \le k < n \\
&\qquad (g'_1, \bar{g}'_1) = \mathsf{decomp}_{g_1 \odot \bar{g}_1}(G') \\
&\qquad (g'_k, \bar{g}'_k) = \mathsf{decomp}_{g_k \odot \bar{g}_k}(\bar{g}'_{k-1}), 1 < k < n
\end{aligned}$$

Decomposition with respect to standard constructors are straightforward.

$$\begin{aligned}
&\mathsf{decomp}_{(x,y)}(x', y') = (x', y') \\
&\mathsf{decomp}_{\textbf{cons}(x,y)}(\textbf{cons}(x', y')) = (x', y')
\end{aligned}$$

Decomposition with respect to binary operator constructed through λ expressions are given below. Unlike other decomposition functions, the original environment ρ is also required.

$$\begin{aligned}
&\mathsf{decomp}_{(\lambda(\$x,\$y).e)(x,y)}(\rho, z) = (\rho'(\$x), \rho'(\$y)) \\
&\text{where} \quad \rho' = \mathscr{B}[\![e]\!](\rho \cup \{\$x \mapsto x, \$y \mapsto y\}, z)
\end{aligned}$$

Structural Recursion Backward evaluation of a structural recursion function is defined by the following stepwise procedure. It performs the inverse procedure of the forward evaluation. The most crucial point is to use the trace information recorded in the nodes of view graph to decompose the view graph into subgraphs and to remove the ϵ-edges generated during the forward evaluation. Further, when we merge the updated environment, we first merge local updates, then merge it to the whole updated environment. Since there might be overlapped parts among them, we let the backward evaluation fail if the overlapped parts are inconsistent. The backward evaluation is,

$$\begin{aligned}
&\mathscr{B}[\![\textbf{srec}(e_\mathrm{b}, d)(e_\mathrm{a})]\!](\rho, G'_{\mathrm{srec}}) = \rho' \\
&\quad \text{where } (V'_{\mathrm{srec}}, B'_{\mathrm{srec}}, I'_{\mathrm{srec}}) = G'_{\mathrm{srec}}
\end{aligned}$$

It also consists of three stages, basically are the reverse operations in the forward evaluation.

1. We first decompose the updated view graph into subgraphs by the trace information encapsulated in RecD and RecN, which is exactly the reverse procedure of Merge, and is defined as follows:
   $\text{Merge}'(G'_{\text{srec}}) = B'_{\text{Dmap}}$
   where,
   $B'_{\text{Dmap}} = \{v \mapsto (\text{VMerge}'(V'_{\text{srec}}, v), \text{BMerge}'(B'_{\text{srec}}, v), \text{IMerge}(I'_{\text{srec}}, v)) \mid v \in V\}$
   Here VMerge′, BMerge′ and IMerge′ evaluate the subgraph associated with node v by collecting the nodes belonging to the same group identified by the last argument of RecD. The input and output marker of each subgraph are obtained from disconnecting the ϵ-edges associated with RecN nodes.
   $\text{VMerge}'(V'_{\text{srec}}, v) = \{\mathsf{RecD}\ p\,u\,v \mid \mathsf{RecD}\ p\,u\,v \in V'_{\text{srec}}\}$
   $\text{BMerge}'(B'_{\text{srec}}, v) = \{\mathsf{RecD}\ p\,u\,v \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(l, \mathsf{RecD}\ p\,w\,v) \Rightarrow x.i \\ x.i = \mathsf{Edge}(\epsilon, \mathsf{RecN}\ p\,w\,\&m) \Rightarrow \mathsf{Outm}(\&m, w) \\ x.i = \mathsf{Outm}(\&m, \&y) \Rightarrow x.i \end{bmatrix}_{i \in |x|} \mid (\mathsf{RecD}\ p\,u\,v \mapsto x) \in B'_{\text{srec}}\}$
   $\text{IMerge}'(V'_{\text{srec}}, v) = \{\&m \mapsto u \mid \mathsf{RecN}\ p\,v\,\&m \mapsto [\mathsf{Edge}(\epsilon, u)] \in B'_{\text{srec}}\}$
2. Then apply the backward evaluation of function d on each subgraph to get updated environment ρ'_u for each list of branches. $\text{Dmap}'(B_{\text{Dmap}}) = B'_{\text{EMap}}$
   where,
   $B'_{\text{EMap}} = \{u \mapsto \mathscr{B}[\![d]\!](\rho_u, \text{DLabel}'(B'_{\text{Dmap}}(u)))(\$b) \mid u \in V\}$

   In this step we first use $\text{DLabel}'(B'_{\text{Dmap}}(u))$ to erase the label RecD to the graph G'_{D} obtained from $B'_{\text{Dmap}}(u)$,
   $\text{DLabel}'(G'_{\text{D}}) = G'$
   where
   $G' = (V', B', I')$, $G'_{\text{D}} = (V'_{\text{D}}, B'_{\text{D}}, I'_{\text{D}})$
   $V' = \{v \mid \mathsf{RecD}\ p\,v\,u \in V'_{\text{D}}\}$
   $B' = \{v \mapsto \begin{bmatrix} x.i = \mathsf{Edge}(l, \mathsf{RecD}\ p\,w\,u) \Rightarrow \mathsf{Edge}(l, w) \\ x.i = \mathsf{Outm}(\&m, m') \Rightarrow x.i \end{bmatrix}_{i \in |x|} \mid (\mathsf{RecD}\ p\,v\,u \mapsto x) \in B'_{\text{D}}\}$
   $I' = \{\&m \mapsto v \mid (\&m \mapsto \mathsf{RecD}\ p\,v\,u) \in I'_{\text{D}}\}$
   Hence, we can apply the backward evaluation of d on this graph, and extract B'_{EMap} from the updated environment. Note that the environment ρ_u is obtained from the second step of forward evaluation.
3. Finally, we apply the backward evaluation of e_{b}: EMap′. Then reconstruct the updated source graph G' by composing the bindings obtained from the backward evaluation of e_{b}. Thus , we are able to reconstruct the updated environment ρ' for the source graph.
   $\text{EMap}'(B'_{\text{EMap}}) = B' = \{u \mapsto \begin{bmatrix} x.i = G_{\text{E}_{\text{u,i}}} \Rightarrow \mathsf{Edge}(\rho'_{u,i}(\$l), \rho'_{u,i}(\$g.I(\&))) \\ \rho'_{u,i} = \mathscr{B}[\![e_{\text{b}}]\!](\rho_{u,i}, \text{ELabel}'(G_{u,i})) \\ x.i = G_{\text{M}_{\text{u,i}}} \Rightarrow \text{MLabel}'(G_{\text{M}_{\text{u,i}}}) \end{bmatrix}_{i \in |x|} \mid (u \mapsto x) \in B'_{\text{EMap}}\}$
   We distinguish the case of different type of branches in B'_{EMap}. If it contains nodes with RecE information, then we should apply *ELabel′* on this graph, and then

apply backward evaluation of e_{b} on this labeled edge. The reconstructed edge can be extracted from the updated environment $\rho'_{u,i}$. Otherwise, we apply MLabel′ to get an output marker. The functions ELabel′, MLabel′ are defined as the inverse operation of ELabel and MLabel.

$\mathrm{ELabel}'(G'_{\mathrm{E}}) = G'$
where
$G' = (V', B', I')$, $G'_{\mathrm{E}} = (V'_{\mathrm{E}}, B'_{\mathrm{E}}, I'_{\mathrm{E}})$
$V' = \{w \mid \mathsf{RecE}\ p\, w\, u\, i \in V_{\mathrm{E}}\}$
$B' = \{\mathsf{RecE}\ p\, w\, u\, i \mapsto \begin{bmatrix} x'.j = \mathsf{Edge}(l, \mathsf{RecE}\ p\, w'\, u\, i) \Rightarrow \mathsf{Edge}(l, w') \\ x'.j = \mathsf{Outm}(\&m, v) \Rightarrow \mathsf{Outm}(\&m) \end{bmatrix}_{j \in |x|}$
$\quad \mid (\mathsf{RecE}\ p\, w\, u\, i \mapsto x) \in B'_{\mathrm{E}},\ \&m \in Z\}$
$I' = \{\&m \mapsto w \mid (\&m \mapsto \mathsf{RecE}\ p\, w\, u\, i) \in I'_{\mathrm{E}}\}$
$\mathrm{MLabel}'(G'_{\mathrm{M}}) = \&y$
if $(\&m, \&y') \in \mathbf{OutMarker}(G'_{\mathrm{M}})$, then $\&y = \&y'$
Finally, we can reconstruct the updated environment
$G' = (V' \cup \bigcup \rho'_{v,i}(\$g).V, B' \cup \bigcup \rho'_{v,i}(\$g).B, I_{\mathrm{src}})$
where V'=$\{v \mid (v \mapsto s) \in B'\} \cup \{w \mid \mathsf{Edge}(l, w) \in s, (v \mapsto s) \in B'\}$
B'=$\mathrm{EMap}'(B'_{\mathrm{EMap}})$
$\rho' = \mathscr{B}[\![e_{\mathrm{a}}]\!](\rho, G') \uplus_{\rho} \biguplus\{\rho'_{v,i} \setminus \{\$l \mapsto \_\} \setminus \{\$g \mapsto \_\}\}$
Note that we unify all the updated sub-environments $\rho'_{u,i}$ evaluated from the body expression e_{b} of **srec**, with the updated environment evaluated from the graph constructor expression e_{a}.

E Reflection to Edge Deletion without Final Check

A natural question is to avoid the final check and the failure of evaluation. Here we provide an alternative solution to avoid the final check in the backward evaluation of edge deletion, which is to allow multiple deletions and to use an iterative procedure. We define the function corr_{set} as follows:

$$
\begin{aligned}
&\mathsf{corr}_{set}(\{(u,i)\}) && = \{(u,i)\} \quad \text{if } u \in \mathit{SrcID} \\
&\mathsf{corr}_{set}(A \cup B) && = \mathsf{corr}_{set}(A) \cup \mathsf{corr}_{set}(B) \\
&\mathsf{corr}_{set}(\{(\mathsf{RecE}\ p\, u\, v\, i, j)\}) && = \mathsf{corr}_{set}(\{(u,j)\}) \cup \mathsf{corr}_{set}(\{(v,i)\}) \\
&\mathsf{corr}_{set}(\{(\mathsf{RecD}\ p\, u\, v, j)\}) && = \mathsf{corr}_{set}(\{(u,j)\}) \\
&\mathsf{corr}_{set}(\{\zeta\}) && = \emptyset \quad \text{otherwise}
\end{aligned}
$$

We start from $D_0 = \{(u,i)\}$, where (u,i) is the deleted edge on the view. Then, we can compute the set of all the correlated edges on the source $D'_0 = \mathsf{corr}_{set}(D_0)$ that could cause deletion for one of the edges in set D_0 on the view. If it is an empty set, then the evaluation fails for invalid deletion on the edges generated from the transformation. Otherwise, we could find the set of corresponding edges $D_1 = D_0 \cup \{(u,i) \mid \mathsf{corr}((u,i)) \in D'_0\}$. Then we can continue to compute $D'_1, D_2, D'_2, \ldots$ by corr_{set} and collect $D_2, D_3, \ldots$ iteratively. The iteration terminates at some fixpoint where $D_n = D_{n-1}$. We take $G'_S = G_S - D'_n$ as the updated source, and $G'_V = G_V - D_n$ to be the updated view.

This procedure would definitely terminate, for the graphs are finite, and $D_{k-1} \subseteq D_k$ for $k = 1, 2, 3, \ldots$. Therefore, there must be a fixpoint. Moreover, the (*WPUTGET*) property is also guaranteed, since the function corr_{set} collects the correspondent edges

from the body expressions as well as the argument expressions. And the procedure terminates when no other edges could be affected anymore. Moreover, if we exclude **isEmpty** from λ_{FG}, the remaining boolean expressions becomes invariant after deletion. Thus the consistency between G'_S and G'_V is maintained. To support all the boolean expressions, we can add a special mark during the forward evaluation of **Condition** clauses, and we disallow the deletion on such edges. Hence corr function tells *FAIL* if we need to delete such edges, which leads to an invalid update.

As a final remark, the iterative procedure might cause multiple deletion of edges in the source with one action from the user. For the previous example, if we remove the edge a on the view, the iterative call to $\mathsf{corr}_{\text{set}}$ ends up with both G'_S and G_V to be empty graphs. This method sacrifices the least change principle [19] to obtain a consistent pair of updated graphs. We could provide an interactive interface for the user to decide whether to take such pair as the updated results.

F Proofs for well-behavedness

The peculiarity of our language (with respect to the well-behavedness) compared to other bidirectional transformation languages is summarized with the following four aspects.

1. Duplication. λ_{FG} duplicates inputs using multiple occurrences of the same variables. View update operation should modify these duplicates consistently (intra-copy consistency), but we accept imperfectly consistent updates, allowing updates on only part of these duplicates in the view.
2. Variable reference (non-point-free). References to free variables in the terms allow receiving data from language constructs that are not immediately preceding. In the point-free style, data flows only between directly adjacent combinators, like $\mathsf{id} \times \mathsf{flatten} \circ \mathsf{unzip} \circ \mathsf{map}$. On the contrary, variable references allow to jump language constructs.
3. Condition. Back propagation of updates may change the control flow. We detect and reject these changes.
4. Injection of information by the forward transformation. It corresponds to constants in the transformation. We prohibit updating this injected information triggered by view updates.
5. Indirect aliasing by label variable and graph variable in the presence of cycles. For example, graph variable bound by **srec** may refer to edges whose label is bound to the label variable introduced by the same **srec** through cycles.

With the above peculiarities, well-behavedness cannot by proved by very simple induction on the structure of λ_{FG}.

The brief idea of well-behavedness preservation can be found in a technical report [13] in an unordered setting, but here we further clarify the extension of backward semantics to take the above aspects into consideration, namely, we take the dependencies between free variables (dependencies between bound edges) into account.

To do this, forward semantics is extended to return traces paired with the original values, and the variable environment correspondingly stores trace information associ-

ated with the values bound to the variables. More precisely, graph and label type values are extended to be paired with trace information.

$$\sigma = \ldots \mid (\mathbf{Graph} \times \mathit{Trace}) \mid (\mathbf{Label} \times \mathit{Trace}_{\mathrm{L}})$$

The shaded part represents the extended part and we apply the same shading in the sequel. The type of forward semantics for graph-valued expressions thus becomes

$\mathscr{F}[\![e]\!] : \mathit{Env} \to (\mathbf{Graph} \times \mathit{Trace})$

and similarly for label-valued expressions:

$\mathscr{F}[\![e]\!] : \mathit{Env} \to (\mathbf{Label} \times \mathit{Trace}_{\mathrm{L}})$

$\mathit{Trace} = (\mathit{Node} \cup \mathit{Edge}) \to \mathit{Node} \cup \mathit{Edge} \cup \mathit{Pos}$

$\mathit{Trace}_{\mathrm{L}} = \mathit{Edge} \cup \mathit{Pos}$

$\mathit{Trace}_{\mathrm{L}}$ corresponds to the trace for labels, which consists of either source edge or the code position which created the label value.

Since we do not change the rule producing values before this extension, we just denote the original constructor semantics using the constructor itself.

$$\mathscr{F}[\![\underline{[]}^p]\!]_\rho = ({}_G\underline{[]}^p, \{G.\mathrm{I}(\&) \mapsto p\}) \quad \text{(T-Emp)}$$

$$\mathscr{F}[\![()^p]\!]_\rho = ({}_G()^p, \{\}) \quad \text{(G-Emp)}$$

$$\mathscr{F}[\![\underline{[\&m]}^p]\!]_\rho = ({}_G\underline{[\&m]}^p, \{G.\mathrm{I}(\&) \mapsto p\}) \quad \text{(OMrk)}$$

$$\mathscr{F}[\![(e_1 \oplus e_2)^p]\!]\rho = (g_1 \oplus g_2, t_1 \cup t_2) \quad \text{where } ((g_1, t_1), (g_2, t_2)) = (\mathscr{F}[\![e_1]\!]_\rho, \mathscr{F}[\![e_2]\!]_\rho) \quad \text{(G-Uni)}$$

$$\mathscr{F}[\![(e_1 \mathbin{@} e_2)^p]\!]_\rho = (g_1 \mathbin{@}^p g_2, t_1 \cup t_2) \quad \text{where } ((g_1, t_1), (g_2, t_2)) = (\mathscr{F}[\![e_1]\!]_\rho, \mathscr{F}[\![e_2]\!]_\rho) \quad \text{(Apnd)}$$

$$\mathscr{F}[\![(\&m \coloneqq e)^p]\!]_\rho = (\&m \coloneqq g, t) \quad \text{where } (g, t) = \mathscr{F}[\![e]\!]_\rho \quad \text{(Imrk)}$$

$$\mathscr{F}[\![\mathbf{cycle}^p(e)]\!]_\rho = ({}_G\,\mathbf{cycle}^p(g), t \cup \{v \mapsto p \mid (\&x \mapsto v) \in G.\mathrm{I}\}) \quad \text{where } (g, t) = \mathscr{F}[\![e]\!]_\rho \quad \text{(Cyc)}$$

$$\mathscr{F}[\![\underline{[}e_1 : e_2\underline{]}^p]\!]_\rho = ({}_G\underline{[}l : e_1\underline{]}^p, \{G.\mathrm{I}(\&) \mapsto p, (G.\mathrm{I}(\&), [1]) \mapsto \tau\} \cup t) \quad \text{where } ((l, \tau), (g, t)) = (\mathscr{F}[\![e_1]\!]_\rho, \mathscr{F}[\![e_2]\!]_\rho) \quad \text{(Edg)}$$

$$\mathscr{F}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!]_\rho = ({}_G(e_1 \mathbin{\underline{+\!+}} e_2)^p, t_1 \cup t_2 \cup \{v \mapsto p \mid (\&x \mapsto v) \in G.\mathrm{I}\}) \quad \text{where}((g_1, t_1), (g_2, t_2)) = (\mathscr{F}[\![e_1]\!]_\rho, \mathscr{F}[\![e_2]\!]_\rho) \quad \text{(T-Uni)}$$

$$\mathscr{F}[\![\mathsf{a}^p]\!]_\rho = (\mathsf{a}, p) \quad (a \in \mathcal{L}) \quad \text{(LCnst)}$$

$$\mathscr{F}[\![\$x^p]\!]_\rho = \rho(\$x) \quad \text{(Var)}$$

$$\mathscr{F}[\![(\lambda(\$x_1, \ldots, \$x_n).e_1\ e_2)^p]\!]_\rho = \mathscr{F}[\![e_1]\!]_{\rho \cup \{\$x_1 \mapsto y_1, \ldots, \$x_n \mapsto y_n\}} \quad \text{where } (y_1, \ldots, y_n) = \mathscr{F}[\![e_2]\!]_\rho \quad \text{(LApp)}$$

List type expressions create lists consisting of pair of values and traces generated by the element expressions. Thanks to the transparent extension, the definition of the

forward semantics remains unchanged for variable references and λ-expression applications.

$$
\begin{aligned}
&\mathscr{F}[\![\mathbf{foldr}(\odot, e)(e_\mathrm{a})]\!]_\rho = f\ l \\
&\quad \text{where} \quad l = \mathscr{F}[\![e_\mathrm{a}]\!]_\rho \\
&\qquad\quad f\ \mathbf{nil} = \mathscr{F}[\![e]\!]_\rho \\
&\qquad\quad f\ \mathbf{cons}(x, xs) = x \odot' (f\ xs) \\
&\qquad\qquad \text{where } y \odot' ys = \begin{cases} (\pi_1 y \odot \pi_1 ys, \pi_2 y \cup \pi_2 ys) & \odot \in \{\underline{+\!+}, @\} \\ y \odot ys & \odot = \mathbf{cons} \end{cases}
\end{aligned}
\tag{FLDR}
$$

foldr behaves differently only when graph constructors are fed as $\odot$.

In the following definition for **srec**, we use two auxiliary functions that operate on traces in a way similar to ELabel and DLabel.

$$
\begin{aligned}
&\mathsf{rece}_{p,(u,i)} t = \{(f\ x) \mapsto \tau \mid (x \mapsto \tau) \in t\} \\
&\qquad \text{where} f\, x = \begin{cases} (\mathsf{RecE}\ \ p\ w\ u\ i, j) & \text{if } x = (w, j) \in \mathit{Edge} \\ \mathsf{RecE}\ \ p\ x\ u\ i & \text{if } x \in \mathit{Node} \end{cases} \\
&\mathsf{recd}_{p,u} t = \{(f\ x) \mapsto \tau \mid (x \mapsto \tau) \in t\} \\
&\qquad \text{where} f\, x = \begin{cases} (\mathsf{RecD}\ \ p\ w\ u, j) & \text{if } x = (w, j) \in \mathit{Edge} \\ \mathsf{RecD}\ \ p\ x\ u & \text{if } x \in \mathit{Node} \end{cases}
\end{aligned}
$$

We also need the external trace for ϵ-elim.

$$
\begin{aligned}
&\epsilon\text{-}\mathsf{elim}((G, t)) = (G', t') \\
&\quad \text{where } (V, B', I) = G' \\
&\qquad B' = \{v \mapsto [B(v).i_n \mid p \in \mathrm{Pb}(G, v), p = (v \xrightarrow{\epsilon}{}_{i_0} \ldots v_n \rightarrow_{i_n})] \mid v \in V\} \\
&\qquad t' = \{v \mapsto t(v) \mid v \in V\} \cup \{(v, i) \mapsto t((v, \mathrm{Pb}(G, v).i)) \mid (v, i) \in \mathsf{edges}(G')\}
\end{aligned}
\tag{EELIM}
$$

$$
\begin{aligned}
&\mathscr{F}[\![(\mathbf{srec}_Z(e_\mathrm{b}, d)(e_\mathrm{a}))^p]\!]_\rho = (g', \bigcup_{v \in g.\mathrm{V}} (\pi_2 \circ B')\ v \cup t'_V) \\
&\quad \text{where } g' = \mathrm{Merge}(g.\mathrm{V}, B_{\mathrm{DMap}}, g.\mathrm{I}) \\
&\qquad (g, t) = \epsilon\text{-}\mathsf{elim}(\mathscr{F}[\![e_\mathrm{a}]\!]_\rho) \\
&\qquad \mathrm{EMap}(B) = \{u \mapsto \mathrm{BMap}(x) \mid (u \mapsto x) \in B\} \\
&\qquad \mathrm{BMap}(x) = \left[\begin{cases} (\mathrm{EL} \times \mathsf{re}).\mathscr{F}[\![e_\mathrm{b}]\!]_{\rho'} & b = \mathsf{Edge}(l, v) \\ & \rho' = \rho \cup \{\$l \mapsto (l, t(u, i)), \\ & \qquad \$g \mapsto (g|_v, t|_v)\} \\ & \mathrm{EL}\ g = \mathrm{ELabel}(g, v, u, i) \\ & \mathsf{re}t = \mathsf{rece}_{p,(u,i)} t \\ (\mathrm{MLabel}(\&y, u, i), \{\}) & b = \mathsf{Outm}(\&y) \end{cases} \middle| \ (b, i) \in_\mathrm{L} x \right] \\
&\qquad \mathrm{DMap}(B) = \{u \mapsto ((\lambda g.\mathrm{DLabel}(g, u)) \times \mathsf{recd}_{p,u}).\mathscr{F}[\![d]\!]_{\rho \cup \{\$b \mapsto x\}} \mid (u \mapsto x) \in B\} \\
&\qquad B' = \mathrm{DMap}(\mathrm{EMap}(g.\mathrm{B})) \\
&\qquad B_{\mathrm{DMap}} = \{v \mapsto g \mid (v \mapsto (g, t)) \in B'\} \\
&\qquad t'_V = \{v \mapsto p \mid v \in \mathrm{NLabel}(g.\mathrm{V})\}
\end{aligned}
\tag{SREC}
$$

$[e \mid (x, i) \in_{\mathrm{L}} l] \stackrel{\text{def}}{=} [e \mid x = l.i]_{i \in |l|}$

The basic idea in this extension to augment trace information in the forward semantics is to maintain correspondence between source node/edge and those in the view. Since the composition is realized through variable bindings, the composition of the correspondence is maintained through the extended environment. The process is like hypothetical syllogism; if A is related to B and B is related to C, then we relate A to C. A variable plays a role of mediation by B.

Nullary graph constructors (G-Emp)(T-Emp)(OMrk) generate no trace information for edges, since they do not create them. Only node trace that associates the node with its code position is generated for node-creating constructors. For label constant expression (LCnst), its code position is generated as the trace information. Graph construction expressions create traces for both nodes and edges. For edge-constructing expression (Edg), the constructed edge is associated with the trace of the label expression. The edge is identified by its origin node and branch position. The position uses the list of integers as the local trace we already introduced in the paper. Since it also creates nodes, the node is associated with the code position as its trace. The trace for the second expression is unified with these traces. Graph binary constructors (T-Uni)(G-Uni)(Apnd) unify the trace information from both operands. Since the environment stores both the value and trace, variable reference looks up these information directly. λ-expression (LApp) extends the environment by the value and trace created by the argument e_2 to evaluate body expression e_1. The semantics for **srec** (Srec) shows how environments for label variable is generated using trace t of its argument expression e_{a}. $t|_v$ is a restriction of the domain of t to the set of edges and nodes reachable from the node v. Note that the mapping goes through auxiliary function ϵ-elim (EElim) to obtain the mapping between nodes/edges before/after ϵ elimination. For d function, the environment for its argument is generated by the list of graphs paired with list of traces and the traces are inherited from those generated by e_{b} for the list of sibling subgraphs for each node of the input graph g.

Given these extended semantics, we obtain, for the top-level environment initialized using input graph g_s with identity traces

$$\rho_0 = \{\$db \mapsto (g_s, \{(v \mapsto v) \mid v \in g_s.\mathrm{V}\} \cup \{(v, i) \mapsto (v, i) \mid (v \mapsto x) \in g_s.\mathrm{B}, (\_, i) \in_{\mathrm{L}} x\})\}$$

a trace that maps a node/edge in the view to a source node/edge if there is a corresponding node/edge in the source, or code position if the node/edge is created in the transformation e, by

$$(g_{\mathrm{T}}, t) = \mathscr{F}[\![e]\!]_{\rho_0}$$

At the beginning of the backward transformation, the edge-renaming is detected by the "delta-discovery" from original graph g_{T} and the updated graph g'_{T} as $\Delta = (g_{\mathrm{T}}, g'_{\mathrm{T}})$ by comparing its two components. It includes the map from edge in the view to its new label value : upd : Upd, $Upd = Edge \to Label$.

$$\begin{aligned}\mathsf{upd}_{\Delta} = \{(v, i) \mapsto l' \mid (v \mapsto x) \in \pi_1(\Delta).\mathrm{B}, (\mathsf{Edge}(l, \_), i) \in_{\mathrm{L}} x,\\ (\mathsf{Edge}(l', \_), i) = \pi_2(\Delta).\mathrm{B}(v).i, l \neq l'\}\end{aligned}$$

Then we can calculate the propagated updates by $\mathsf{utrans} : \mathit{Upd} \times \mathit{Trace} \to \mathit{Upd}$ to any other intermediate view for a subexpression e' and its trace t' under its environment $\rho' : (g', t') = \mathscr{F}[\![e']\!]_{\rho'}$

$$\mathsf{upd}' = \mathsf{utrans}(\mathsf{upd}_{\Delta}, t') = \{\zeta' \mapsto \mathsf{upd}_{\Delta}(\zeta) \mid \zeta' \in \mathsf{edges}(\pi_2(\Delta)), \zeta \in \mathrm{dom}(\mathsf{upd}_{\Delta}), t'(\zeta') = t(\zeta)\}$$

where $\mathsf{edges} g$ collects every edge in the graph g.

$$\mathsf{edges} g = \{(v, i) \mid (v \mapsto x) \in g.\mathrm{B}, (\_, i) \in_{\mathrm{L}} x\}$$

Now we describe the backward semantics using the traces and update propagation function. The backward transformation fails for the attempts to update constants in the transformation. That corresponds to *Upd* defined for an edge ζ such that $t(\zeta) \in \mathit{Pos}$. For variable reference, we update entry of environment accordingly, but here we also update binding of other variable affected, just like we did for computation of upd'.
$\mathsf{pr} : \mathit{Env} \times \mathit{Upd} \times \mathit{Trace} \to \mathit{Env}$.

$\mathsf{pr}(\rho, \mathsf{upd}, t) = \rho'$
where, for a graph variable $\$g$
$\rho'(\$g) = (g', t')$
where $(g, t') = \rho(\$g)$
$g' = (g.\mathrm{V}, \mathrm{B}', \mathrm{g.I})$
$B' = \{v \mapsto [u(v, b, i) \mid (b, i) \in_{\mathrm{L}} x] \mid (v \mapsto x) \in g.\mathrm{B}\}$
$$u(v, b, i) = \begin{cases} \mathsf{Edge}(\mathsf{upd}(\zeta), u) & b = \mathsf{Edge}(\_, u) \wedge \exists \zeta. t'((v, i)) = t(\zeta) \\ b & \text{otherwise} \end{cases}$$
For a label variable $\$l$
$\rho'(\$l) = (l', \tau')$
where $(l, \tau') = \rho(\$l)$
$$l' = \begin{cases} \mathsf{upd}(\zeta) & \tau' = t(\zeta) \\ l & \text{otherwise} \end{cases}$$

By doing this, we detect inconsistent edge renaming caused by duplicates and aliasing at the merging phase using $\uplus$ for binary operators and binders like **srec**.

For a variable we do not just simply update its bindings as $\mathscr{B}[\![\$v]\!](\rho, G') = \rho[\$v \leftarrow G']$ but we propagate this change to other bindings using upd_{Δ} as defined above.

$$\mathscr{B}[\![\$g]\!](\rho, G') = \mathsf{pr}(\rho, \mathsf{utrans}(\mathsf{upd}_{\Delta}, t')) \quad \text{where}(g', t') = \rho(\$x) \qquad (\textsc{BVar})$$

Therefore,
$\mathscr{B}[\![(e_1 \mathbin{\underline{+\!+}} e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G'_1) \uplus_{\rho} \mathscr{B}[\![e_2]\!](\rho, G'_2)$
where $(G_1, t_1) = \mathscr{F}[\![e_1]\!]\rho, \quad (G_2, t_2) = \mathscr{F}[\![e_2]\!]\rho$
$(G'_1, G'_2) = \mathsf{decomp}_{G_1 \underline{+\!+}^p G_2}(G')$
will detect failure via the propagation, for example, in case of
$\mathscr{B}[\![\$db \mathbin{\underline{+\!+}} \underline{[}\$l : \underline{[]}]\!](\rho, G')$
ρ includes at least both bindings of $\$db$ and $\$l$. Since $\mathscr{B}[\![\$db]\!]$ returns bindings with not only entry of $\$db$ but also $\$l$ is updated, if they are affected with each other, and $\mathscr{B}[\![\$l]\!]$

returns correspondingly variables with both entries updated, the merge operation $\uplus$ for both backward transformation detects binding conflicts (if exists) and fails.

Next we recap the backward semantics of **srec** in the following. Given the revised backward semantics of variable references, the backward semantics remains unchanged.

$$
\begin{array}{l}
\mathscr{B}[\![\mathbf{srec}(e_\mathrm{b}, d)(e_\mathrm{a})]\!](\rho, G'_{\mathrm{srec}}) = \mathscr{B}[\![e_\mathrm{a}]\!](\rho, \epsilon\text{-}\mathsf{elim}^{-1}(G')) \uplus_\rho \rho'' \\
\quad \text{where } (V_{\mathrm{src}}, B_{\mathrm{src}}, I_{\mathrm{src}}) = g = \epsilon\text{-}\mathsf{elim}(\mathscr{F}[\![e_\mathrm{a}]\!]_\rho) \\
\qquad B'_{\mathrm{EMap}} = \mathrm{DMap}'(\mathrm{Merge}'(G'_{\mathrm{srec}})) \\
\qquad\quad \mathrm{Merge}'(G) = \{u \mapsto (\mathrm{VMerge}'_u(G.\mathrm{V}), \mathrm{BMerge}'_u(G.\mathrm{B}), \mathrm{IMerge}'_u(G.\mathrm{B})) \mid u \in V_{\mathrm{src}}\} \\
\qquad\qquad \mathrm{VMerge}'_v(V) = \{v' \mid v' \in V, v' = \mathsf{RecD}\ p\, u\, v\} \\
\qquad\qquad \mathrm{BMerge}'_v(B) = \{v' \mapsto [\mathrm{bmapB}'(b) \mid b \in x] \mid (v' \mapsto x) \in B, v' = \mathsf{RecD}\ p\, u\, v\} \\
\qquad\qquad\quad \mathrm{bmapB}'\ (\mathsf{Edge}(\epsilon, \mathsf{RecN}\ p\, w\, \&m)) = \mathsf{Outm}(\&m, w) \\
\qquad\qquad\quad \mathrm{bmapB}'\ b = b \quad \text{otherwise} \\
\qquad\qquad \mathrm{IMerge}'_v(B) = \{\&m \mapsto u \mid \mathsf{RecN}\ p\, v\, \&m \mapsto [\mathsf{Edge}(\epsilon, u)] \in B\} \\
\qquad\quad \mathrm{DMap}'(B) = \{u \mapsto \mathscr{B}[\![d]\!](\rho_u, \mathrm{DLabel}'(B(u)))(\$b) \mid u \in V_{\mathrm{src}}\} \\
\qquad B' = \mathrm{EMap}'(B'_{\mathrm{EMap}}) \\
\qquad\quad \mathrm{EMap}'(B) = \{u \mapsto [\mathrm{emapB}'(u, b, i) | (b, i) \in_\mathrm{L} x] \mid (u \mapsto x) \in B\} \\
\qquad\qquad \mathrm{emapB}'(u, b, i) = \begin{cases} \mathscr{B}[\![e_\mathrm{b}]\!](\rho', \mathrm{ELabel}'(b)) & B_{\mathrm{src}}(u).i = \mathsf{Edge}(l, v) \\ & \rho' = \rho \cup \{\$l \mapsto l, \$g \mapsto g|_v\} \\ \mathrm{MLabel}'(b) & \text{otherwise} \end{cases} \\
\qquad B''_{\mathrm{EMap}} = \{u \mapsto [\mathrm{emapB}''(u, b, i) | b \in_\mathrm{L} x] \mid (u \mapsto x) \in B'_{\mathrm{EMap}}\} \\
\qquad\quad \mathrm{emapB}''(u, b, i) = \begin{cases} \mathsf{Edge}(b(\$l), b(\$g).\mathrm{I}(\&)) & B_{\mathrm{src}}(u).i = \mathsf{Edge}(\_, \_) \\ b & \text{otherwise} \end{cases} \\
\qquad G' = (V' \cup \bigcup_{(v \mapsto x) \in B''_{\mathrm{EMap}}} \cup [\rho(\$g).\mathrm{V} \mid \rho \in x], B' \cup \bigcup_{(v \mapsto x) \in B''_{\mathrm{EMap}}} \cup [\rho(\$g).\mathrm{B} \mid \rho \in x], I_{\mathrm{src}}) \\
\qquad\quad V' = \{v \mid (v \mapsto s) \in B'\} \cup \{w \mid \mathsf{Edge}(l, w) \in s, (v \mapsto s) \in B'\} \\
\qquad\quad \rho'' = \biguplus_{(v \mapsto x) \in B''_{\mathrm{EMap}}} \uplus [b \setminus \{\$l \mapsto \_\} \setminus \{\$g \mapsto \_\} \mid b \in x] \\
\quad \mathrm{MLabel}'(G'_\mathrm{M}) = \mathsf{Outm}(\&y) \\
\qquad \text{where } (\&m, \&y') \in \mathbf{OutMarker}(G'_\mathrm{M}) \\
\quad \mathrm{ELabel}'(G'_\mathrm{E}) = (V', B', I') \\
\qquad \text{where } V' = \{\mathrm{stripE}(v) \mid v \in G'_\mathrm{E}.\mathrm{V}\} \\
\qquad\qquad B' = \{\mathrm{stripE}(v) \mapsto [\mathrm{stripB}(b) \mid b \in_\mathrm{L} x] \mid (v \mapsto x) \in G'_\mathrm{E}.\mathrm{B}, \&m \in Z\} \\
\qquad\qquad I' = \{\&m \mapsto \mathrm{stripE}(v) \mid (\&m \mapsto v) \in G'_\mathrm{E}.\mathrm{I}\} \\
\qquad\qquad \mathrm{stripE}\ (\mathsf{RecE}\ p\, w\, u\, i) = w \\
\qquad\qquad \mathrm{stripB}\ \ \mathsf{Edge}(l, v) = \mathsf{Edge}(l, \mathrm{stripE}(v)) \\
\qquad\qquad \mathrm{stripB}\ \ \mathsf{Outm}(\&m, v) = \mathsf{Outm}(\&m) \\
\quad \mathrm{DLabel}'(G'_\mathrm{D}) = (V', B', I') \\
\quad \text{where } V' = \{\mathrm{stripD}(v) \mid v \in G'_\mathrm{D}.\mathrm{V}\} \\
\qquad\qquad B' = \{\mathrm{stripD}(v) \mapsto [\mathrm{stripB(b)} \mid b \in x] \mid \in (v \mapsto x) \in G'_\mathrm{D}.\mathrm{I}\} \\
\qquad\qquad I' = \{\&m \mapsto \mathrm{stripD}(v) \mid (\&m \mapsto v) \in G'_\mathrm{D}.\mathrm{I}\} \\
\qquad\qquad \mathrm{stripD}\ (\mathsf{RecD}\ p\, w\, u) \quad = w \\
\qquad\qquad \mathrm{stripB}\ (\mathsf{Edge}(l, v)) \quad\ = \mathsf{Edge}(l, \mathrm{stripD}(v)) \\
\qquad\qquad \mathrm{stripB}\ (\mathsf{Outm}(\&m, m')) = \mathsf{Outm}(\&m, m') \\
\quad \text{For the backward semantics of } \mathbf{if}, \text{ the propagation by } \mathsf{pr} \text{ and } \mathsf{utrans} \text{ is used.}
\end{array}
$$

$$\mathscr{B}\left[\!\!\left[\begin{array}{l}\textbf{if } e_1 \textbf{ then } e_2\\ \textbf{else } e_3\end{array}\right]\!\!\right](\rho, G') = \begin{cases} \rho_2' & \text{if } \mathscr{F}[\![e_1]\!]\rho \wedge \mathscr{F}[\![e_1]\!]\rho_2'', \text{ where } \rho_2' = \mathscr{B}[\![e_2]\!](\rho, G') \\ & \rho_2'' = \mathsf{pr}(\rho_2', \mathsf{utrans}(\mathsf{upd}_\Delta, t_1)) \\ & \quad (g_1, t_1) = \mathscr{F}[\![e_1]\!]_\rho \\ \rho_3' & \text{if } \neg\mathscr{F}[\![e_1]\!]\rho \wedge \neg\mathscr{F}[\![e_1]\!]\rho_3'', \text{ where } \rho_3' = \mathscr{B}[\![e_3]\!](\rho, G') \\ & \rho_3'' = \mathsf{pr}(\rho_3', \mathsf{utrans}(\mathsf{upd}_\Delta, t_2)) \\ & \quad (g_2, t_2) = \mathscr{F}[\![e_2]\!]_\rho \\ \textit{FAIL} & \text{otherwise} \end{cases} \tag{BIF}$$

Example 5. Suppose we have transformation

$$\textbf{srec}(\lambda(\$l, \$g).\ \textbf{if } \$l = \mathsf{a} \textbf{ then } \$db \textbf{ else } \underline{[}\$l : \underline{[}\$l : \underline{[]]}\underline{]}, id)(\$db)$$

and source graph bound to $\$db$ is $\underline{[}\mathsf{a} : \underline{[}\mathsf{b} : \underline{[]]}\underline{]}$, and view graph $\underline{[}\mathsf{a} : \underline{[}\mathsf{a} : \underline{[]]}\underline{]}$ is updated to $\underline{[}\mathsf{c} : \underline{[}\mathsf{a} : \underline{[]]}\underline{]}$. Then we compute the update operation translation utrans to the source edges. In this case, it is renaming a to c at the top level edge, because the view is created by the **then** clause of **if**, which is a direct reference to $\$db$. Then, during the backward transformation of **if** which is reduced to that of the **then** clause, we update the binding of free variables in the condition $\$l = \mathsf{a}$, which is $\$l$ in this case, and obtain the corresponding source edge by looking up the trace stored in the environment to find the edge a. Since the binding intersects with the user's update above, the binding is also updated to label c. This is achieved by (BVAR). Then we re-evaluate ($\$l = \mathsf{a}$) which is false, and compare with the value during the forward transformation and detect control flow change. So the backward transformation of **if** fails and rejects the update.

Given the detection of update conflict and attempts to update constants, we can focus on the successful propagation for the proof of *WPUTGET* by induction on the structure of λ_{FG} expressions.

Before we start the proof of well-behavedness of $\underline{+\!\!+}$, we need the following lemmas.

Lemma 1 (Well-behavedness of $\mathsf{decomp}_{\underline{+\!\!+}}$).

$$\begin{aligned} &\forall g_1, g_2 \in \mathcal{G}_Y^X.\mathsf{decomp}_{g_1 \underline{+\!\!+} g_2}(g_1 \mathbin{\underline{+\!\!+}} g_2) &&= (g_1, g_2) \\ &\forall g_1, g_2, g \in \mathcal{G}_Y^X.(\underline{+\!\!+}) \circ \mathsf{decomp}_{g_1 \underline{+\!\!+} g_2}(g) &&= g \end{aligned}$$

The equations respectively correspond to GetPut and PutGet.

Lemma 2 (Well-behavedness of $\uplus$).

$$\begin{aligned} \rho \uplus_\rho \rho &= \rho \\ \rho' = \rho_1 \uplus_\rho \rho_2 &\Rightarrow \rho' \uplus_\rho \rho' = \rho' \end{aligned}$$

Theorem 2 (Agreement of $\mathscr{B}[\![\_]\!]$ and prop). *Given expression e, its result of transformation g with trace t, update Δ of g to g' and the successful backward transformation with g', the result ρ' agrees with direct propagation of Δ along with trace t using* prop*, i.e.,*

$$\begin{aligned} &\mathsf{prop}(\Delta, t, \rho) = \mathsf{pr}(\rho, \mathsf{upd}_\Delta, t) \\ &\mathscr{F}[\![e]\!]_\rho = (g, t) \wedge \Delta = (g, g') \wedge \rho' = \mathscr{B}[\![e]\!](\rho, g') \Rightarrow \mathsf{prop}(\Delta, t, \rho) = \rho' \end{aligned}$$

This theorem also intuitively shows that we avoid useless backward semantics that always does nothing to the source environment, which still satisfies *WPUTGET*.

Proof Theorem 2 can be proved using induction on the structure of λ_{FG}. For $\underline{+\!\!+}$, we show $\mathsf{prop}(\Delta, t, \rho) = \mathscr{B}[\![e_1 \mathbin{\underline{+\!\!+}} e_2]\!](\rho, g')$.

$\mathsf{prop}(\Delta, t, \rho)$
= { distribution property of Δ and t over $\underline{+\!\!+}$ }
$\mathsf{prop}(\Delta_1 \cup \Delta_2, t_1 \cup t_2, \rho)$
= { property of prop and $\uplus$ }
$\mathsf{prop}(\Delta_1, t_1, \rho) \uplus \mathsf{prop}(\Delta_2, t_2, \rho)$
= { induction hypotheses on e_1 and e_2 }
$\mathscr{B}[\![e_1]\!](\rho, g'_1) \uplus_\rho \mathscr{B}[\![e_1]\!](\rho, g'_2)$
= { definition of $\mathscr{B}[\![\_]\!]$ for $\underline{+\!\!+}$ }
$\mathscr{B}[\![e_1 \mathbin{\underline{+\!\!+}} e_2]\!]\rho(g')$

Other cases are treated similarly.

Theorem 3 (Label forward propagation). *Edges in the view graph has the same label as that of the origin.*

$$\mathscr{F}[\![e]\!]_\rho = (g, t) \Rightarrow \forall \zeta \in \mathsf{edges}(g), \mathsf{label}(\zeta) = \mathsf{label}(t(\zeta))$$

This can be proved by a straightforward induction.

Corollary 1 (Copy has the same label). *Given* $\mathscr{F}[\![e]\!]_\rho = (g, t)$,

$$\forall \zeta_1, \zeta_2 \in \mathsf{edges}(g), t(\zeta_1) = t(\zeta_2) \Rightarrow \mathsf{label}(\zeta_1) = \mathsf{label}(\zeta_2)$$

This corollary leads to Theorem 4, because right after forward transformation (i.e., before view updates), all the edges in the equivalence class has the same labels, meaning that no more propagation is possible. So this view corresponds to the one that includes the maximum updates.

Theorem 4 (Maximum propagation). *The graph* g'' *in* $(g'', t'') = \mathscr{F}[\![e]\!]_{\mathscr{B}[\![e]\!](\rho, g')}$ *contains all the updates applied for all duplicates in the view on updates* $\Delta = (g, g') = (\pi_1(\mathscr{F}[\![e]\!]_\rho), g')$.

It corresponds to intra-view update propagation in [16].

The proof of theorem 4 can be done using the induction on the structure of λ_{FG} expressions, assuming maximum propagation property of subexpressions and that of $\uplus$.

By theorem 4, we have that the entire round-trip of expression e consisting of the backward transformation on the target graph g for update on a given edge ζ will propagate to all the other edges in g within the equivalence class of ζ, denoted by $[\zeta]_\sim$ defined by $\zeta_1 \sim \zeta_2 \Leftrightarrow t(\zeta_1) = t(\zeta_2)$ for $\mathscr{F}[\![e]\!]_\rho = (g, t)$.

This theorem and theorem 2 leads to a well-behavedness property that is slightly stronger than *WPUTGET*, because *WPUTGET* only guarantees that the second and the third backward transformation lead to the same updated environment, but does not characterize the view graph of the second forward transformation.

To complete the proof of *WPUTGET* for an entire transformation, we also check that control flow change do not occur for the other execution paths for the edges in the equivalence class. So we have to check this control flow stability through checking all the if conditions along with these paths. With the assumption that these conditions remain unchanged, the round-trip propagates updates on (subset of) edges in the equivalence class and the backward transformation on the new target graph lead to the same updated environment so that *WPUTGET* is satisfied.

The second and the third view has the same shape.

Definition 1 (shape of graphs). *Graphs g_1 and g_2 has the same shape, denoted by $g_1 \doteq g_2$ if and only if they are exactly the same after forgetting their edge labels.*

$$
\begin{aligned}
& g_1 \doteq g_2 \Leftrightarrow g_1.\mathrm{V} = g_2.\mathrm{V} \wedge g_1.\mathrm{I} = g_2.\mathrm{I} \wedge \mathsf{fl}(g_1.\mathrm{B}) = \mathsf{fl}(g_2.\mathrm{B}) \\
& \textit{where } \mathsf{fl}(B) \quad = \{v \mapsto [f b \mid b \in x] \mid (v \mapsto x) \in B\} \\
& \qquad f(\mathsf{Edge}(l, v)) = \mathsf{Edge}(\epsilon, v) \\
& \qquad f(\mathsf{Outm}(m)) \ = \mathsf{Outm}(m)
\end{aligned}
$$

Definition 2 (Shape genericity). *Function f is shape generic if and only if, for any two graphs g_1 and g_2 with $g_1 \doteq g_2$, application of f produces the graphs with the same shape.*

$$g_1 \doteq g_2 \rightarrow f(g_1) \doteq f(g_2)$$

Theorem 5. *Under unchanged control flow, all λ_{FG} expressions are shape generic.*

Proof can be done using a simple induction on the structure of λ_{FG}. No graph constructor inspects labels, so by induction on their operand expressions, they preserves the shape. For **srec**, the intermediate results generated has the same shape regardless of the labels in the input graph. Therefore, with induction hypotheses on the body, argument and d function, **srec** also preserves the shape. Since we assume unchanged control flow (which is checked by pr), **if** expression preserves the shape by the induction hypotheses on both clauses.

Theorem 6. *Under unchanged control flow, all λ_{FG} expressions generate the same trace for the source graphs with identical shape.*

This can be proved in a similar manner as the proof of Theorem 5.

Now we prove *WPUTGET* for edge renaming.

Theorem 7. *All λ_{FG} expression satisfy WPUTGET for edge renaming.*

Proof. By Theorem 2, backward transformation propagates edge renaming on the view to those of the source. By the assumption that backward transformation succeeds, control flow does not change. Since the edge renaming does not change the shape, the forward transformation on the updated environment generates another view graph with the same shape and same trace, i.e., we have $g' \doteq \pi_1(\mathscr{F}[\![e]\!]_{\mathscr{B}[\![e]\!](\rho,g')})$. By Theorem 4, update is propagated to all the edges in the equivalence class, so there will be no conflict encounter on another backward transformation with the view $(\mathscr{F}[\![e]\!]_{\mathscr{B}[\![e]\!](\rho,g')})$. By Theorem 2, results on this backward transformation agrees with direct propagation with

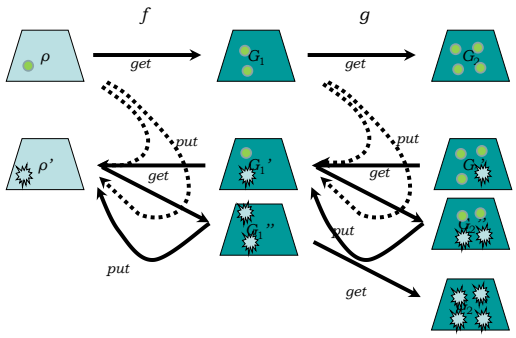

Fig. 9. Compositionality of *WPUTGET*

the trace, which, by Theorem 6 has the same as the previous trace, so that result in the same updated environment.

In a formal reasoning,

$\mathscr{B}[\![e]\!](\rho, \pi_1(\mathscr{F}[\![e]\!]_{\mathscr{B}[\![e]\!](\rho,g')}))$

$= \quad \{\ \mathscr{B}[\![e]\!](\rho, g')$ succeeds, Thm 2, $\pi_2(\mathscr{F}[\![e]\!]_\rho) = \pi_2(\mathscr{B}[\![e]\!](\rho, \pi_1(\mathscr{F}[\![e]\!]_{\mathscr{B}[\![e]\!](\rho,g')})))$, Thm 4 $\}$

$\mathscr{B}[\![e]\!](\rho, g')$

Definition 3 (Completion of view label updates). *Given* $(g, t) = \mathscr{F}[\![e]\!]_\rho$ *and view update* $\Delta = (g, g')$*. For each edge-renaming on* $\zeta \in \mathsf{edges}(g)$*, completion of view label updates applies the same renaming to all the other view edges* $\zeta' \in [\zeta]_\sim$*.*

$$
\begin{aligned}
&\mathsf{compl}(g, \Delta, t) = g' \\
&\quad \textit{where } g' \quad = (g.\mathrm{V}, \mathrm{B}', \mathrm{g.I}) \\
&\qquad\quad B' \quad = \{v \mapsto [u(v, b, i) \mid (b, i) \in_{\mathrm{L}} x] \mid (v \mapsto x) \in g.\mathrm{B}\} \\
&\qquad\quad u(v, b, i) = \begin{cases} \mathsf{Edge}(\mathsf{upd}_\Delta(\zeta), u) & b = \mathsf{Edge}(l, u) \wedge (v, i) \in [\zeta]_\sim \\ b & \textit{otherwise} \end{cases}
\end{aligned}
$$

The backward transformation $\mathscr{B}[\![e]\!](\rho, g')$, if succeeds and results in ρ', leads to the same updated environment as the one obtaining without the completion of view label updates. The completed view coincides with $\mathscr{F}[\![e]\!]_{\rho'}$.

The compositionality of *WPUTGET* is illustrated by Figure 9. This situation can be caused by λ_{FG} expression like

$$(\lambda \$g.copy(\$g) \mathbin{\underline{+\!+}} \$g)(copy(\$db) \mathbin{\underline{+\!+}} \$db)$$

such that $copy(g)$ copies the graph deeply. It is just used to generate graph equivalent to g but has different set of node ids. Then the transformation duplicates the input twice. These duplicates are sequentially composed by the λ expression. The figure shows that the view side effects are caused only by duplicates (we have totally four copies) (control flow change is detected and rejected by the backward semantics of **if**). Suppose the view G_2 is updated to G_2' by updating only one of the duplicates. Then the backward transformation of the second transformation g will propagate the update, and the second forward transformation of g will propagate the change to the other copy in the view

(G''_2). Assuming *WPUTGET* for g, the backward transformation by G''_2 agrees at graph G'_1. The backward transformation and next forward transformation of f will propagate the change on G'_1 to the other copy to produce G''_1. Forward transformation of g on it will produce G'''_2 in which the updates are propagated to all the four copies. Since they are updates on the copies, backward transformation of g on G'''_2 succeeds and goes back to G''_1, and again, the updates are all on the copies caused by f, so backward transformation f on G''_1 will go back to ρ'. Therefore, backward transformation of $g \circ f$ relative to ρ results in ρ', satisfying *WPUTGET* on the whole.

In general, backward transformation arbitrarily updates variable binding environments through free variables, so this simple compositionality does not hold. That is why the proof of *WPUTGET* does not use *WPUTGET* of subexpressions as induction hypotheses.