Bidirectional Transformation on Ordered Graphs

Soichiro Hidaka[1], Fei Yang[2], Kazuyuki Asada[3], Hiroyuki Kato[1], Keisuke Nakano[3], and Zhanjiang Hu[1]

[1] National Institute of Informatics, Japan
{hidaka,asada,kato,hu}@nii.ac.jp
[2] BASICS, Department of Computer Science, Shanghai Jiao Tong University, China
iamyf@sjtu.edu.cn
[3] University of Electro-Communications, Japan
ksk@cs.uec.ac.jp

Abstract. The linguistic (language-based) approach plays an important role in development of structured and well-behaved bidirectional transformations. It has been successfully applied to solve the callenging problem of bidirectional transformation on graphs, where a clear bidirectional semantics if given based on a bulk semantics of the structural recursion. However, the graphs that can be dealt with are limited to unordered ones, and it has not yet been known how to treat ordered graphs such as XML graphs in which the child nodes of a node are ordered. λ_{FG} is a language for querying ordered graphs, a bulk semantics of structural recursion is provided for ordered graphs. In this paper, we show that ordered graphs can be bidirectionalized by a three-stage procedure. We provided a bidirectionalized procedure to eliminate ϵ-edges. Further, a clear bidirectional semantics for transformation language λ_{FG} can be defined. The key technical point is a novel way of bidirectionalizing the bulk semantics in λ_{FG}.

Keywords: Ordered Graphs, Bidirectional Transformation, Bulk Semantics, λ_{FG} Language

1 Introduction

Bidirectional transformations[5, 4] provide a novel mechannism for sychronizing and maintaining the consistency of information between input and output. The are pervasive and has may potential applications, including the synchronization of replicated data in different formats[5], presentation-oriented structured document development, interactive user interface design[11], coupled software transformation, and the well-known *view updating* mechanism which has been intensively studied in the database community[1, 6].

The linguistic (language-based) approach[5] gives a promising way for development of structured and well-behaved bidirectional transformation, in wich every expreesion simultaneously specifies both a forward and the responding (correct) backward transformation, and every composite of expressions defines a structured way of gluing smaller bidirectional transformations to a bigger one. Despite its usefulness for bidirectional transformation on lists and trees[5, 2, 10, 9, 12–14], it is a challenge to deal with bidirectional transformation on graphs. First, 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. Second, graphs have *shared node and cycles*, which makes both forward and backward trasformation computation more complicated than that on trees; naïve computation would visit the same nodes many times and possibly infinitely.

In our previous work[8], we challenged the probelm by showing that the linguistic approach can be applied to bidirectional transformation on graphs, where a cleawr bidirectional semantics is given for UnCAL, a graph algebra for the known graph query language UnQL[3]. The key to this success is the bulk semantics of the structural recursion function a structural recursion is evaluated by first processing *in parallel* on all edges of the input graph and then combining the results. This bulk semantics relies on introduction of ϵ-edges to grapohs, 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 unorered. It has not yet been known how to treat ordered graphs such as XML graphs in which the child nodes of a node are ordered. One might consider encoding the ordered graphs in terms of unordered ones by introducing specialized edge labels, but this would make it difficult to keep consistency of these labels during transformation. In fact, it is an open problem, as pointed out in[3], how labels during tranformation on ordered graphs. In[7] , a transformation language for ordered graph λ_{FG} is provided, a bulk semantics for transformation on ordered graphs is also given. Thus, a natural question is how can we bidirectionalize transforamtion on ordered graphs via λ_{FG}.

In this paper, we show that the ordered graphs can be bidirectionalized. The work is based on the equivalence relation of bisimilarity and the query language λ_{FG} on ordered graphs. The main technical contributions are three folds. First, a novel three-stage bidirectional transformation strategy is designed for ordered graphs. Next we give a bidirectionalized procedure to eliminate ϵ-edges. Finally, we construct a bidirectional evaluation semantics for λ_{FG}.

2 Preliminaries

Ordered graphs is a graph model in which the branchings are ordered. We start from introducing a formal definition for ordered graphs, and give a bisimilarity equivalence relation on the ordered graphs.

2.1 Ordered Graphs

The graphs we treated here are multi-rooted, directed, and edge-labeled graphs with order on outgoing edges. This graph model has three prominent features: *ϵ-edges*, *markers*, and *graph concatenation*. An ϵ-edges represents the a shortcut between two nodes, it is similar with the ϵ-transition in automata. Markers can be treated as interfaces that connect the nodes to other graphs. A node can be marked with *input* and *output markers*. Graph concatenation sequentially aligns two graphs in a given order.

The ordered graph model is formally defined as follows. We write $\mathcal{L}$ for the set of *labels* and $\mathcal{L}_\epsilon$ for $\mathcal{L} \cup \{\epsilon\}$. Let X and Y be finite sets of input and output markers; we add the prefix & for markers like &x. An *ordered graph* G is defined by a triple (V, B, I), where

- V is the set of *nodes*,
- $B : V \to List(\mathcal{L}_\epsilon \times V + Y)$ is a *branch function* mapping a node to a list of *branches*: a branch in $\mathcal{L}_\epsilon \times V + Y$ is either a *labeled edge* **Edge**(l, v) or an *output marker* **Outm**&y), and
- $I : X \to V$ is a function which determines the *input nodes* (roots) of the graph.

Here, every node in the range of I is called a root. Note that a root node may have incoming edges. However, we can convert it to a equivalent graph having no such incoming edges.

We denote a graph with the input marker set X and the output marker set Y by G_Y^X. Moreover, we write $\mathcal{G}_Y^X$ to represent the set of such graphs. Another assumption is that the graphs we are going to talk about are restricted to finite graphs, which means the finiteness on the set of nodes. ${\mathcal{G}^c}_Y^X$ is used to denote the set of ordered graph with a countble width. It is obvious that the set of finite ordered graphs $\mathcal{G}_Y^X$ is a subset of ${\mathcal{G}^c}_Y^X$.

For the 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 $\mathcal{G}_Y$ ot denote $G_Y^{\{\&\}}$.

2.2 Graph Constructors

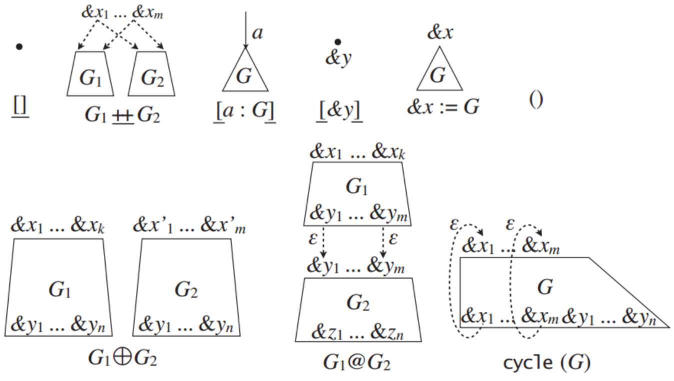

Fig. 1. Graph Constructors

Figure 1 sumarizes the nine costructors we used to describe arbitrary graphs.

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

The single node graph constructor $\underline{[]}$ construct a root-only graph. $G_1 \underline{+\!\!+} G_2$ adds two ϵ-edges from a new root to the roots of G_1 and G_2, respectively. $\underline{[}a : G\underline{]}$ constructs a graph adding an a labeled edge point 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 or an edge. Furthermore, $G_1 \oplus G_2$ constructs a graph by using a componentwise (V, B, I) union, $\underline{+\!\!+}$ differs from $\oplus$ in that $\underline{+\!\!+}$ unifies input nodes while $\oplus$ does not. $\oplus$ requires them to be identical. $G_1 @ G_2$ composes two graphs by connecting the output nodes of G_1 with the corresponding input nodes of G_2 with ϵ edges. $\mathbf{cycle}(G)$ connects the output nodes of G with the corresponding input nodes with ϵ edges, that will form cycles. The newly constructed nodes have unique identifiers. The formal definition of the sementics is given in λ_{FG}.

2.3 Proper Branches

In the definition of ordered graphs, the ϵ-edges are introduced to represent the shortcut between nodes. We need to relate two connected nodes by *proper branch*, where we can ignore the shortcuts. A *proper branch* of a node v is defined as a path from v going through zero or more ϵ-edges until it reaches a non-ϵ-edge or output marker.

Let $G = (V, B, I) \in \mathcal{G}^{cX}_{Y}$ and $v \in V$. The path starting from v

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

is called a *proper branch* of v if the i_n-th branch $B(v_n).i_n$ is not an ϵ-edge. (i.e. a non-ϵ-edge or an output marker. The set of all proper branches of v in G is denoted by $Pb(G, v)$.

Like the order on the branches, a total order is introduced on proper branches. Given two proper branches in an arbitrary graph, $p = (v \xrightarrow{\epsilon}{}_{i_0} v_1 \ldots \xrightarrow{\epsilon}{}_{i_{n-1}} v_n \longrightarrow{}_{i_n})$, and $p' = (v \xrightarrow{\epsilon}{}_{i'_0} v'_1 \ldots \xrightarrow{\epsilon}{}_{i'_{n'-1}} v'_{n'} \longrightarrow{}_{i'_n})$. Let their branch index sequences be $\tilde{p} \overset{def}{=} (i_0, \ldots, i_{n-1}, i_n)$ and $\tilde{p'} \overset{def}{=} (i'_0, \ldots, i'_{n'-1}, i'_{n'})$. We define $p \leqslant_{Pb} p' \overset{def}{\iff} \tilde{p} \leqslant_l \tilde{p'}$ where $\leqslant_l$ is the lexicographical order between branch index sequences. This order plays an important roll in the definition of bisimilarity and the procedure of bidirectionalizing 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 a output marker.

2.4 Bisimilarity for Ordered Graphs

For the soundness of the graph model, we need an equivalence relation that can judge whether two arbitrary graphs are the same. At first glimpse, graph isomophism is the most straight forward approach. However, it is not necessarily that two graphs are isomophic when they behaves the same. Another option is language equivalence, which is focused on the language produced by the graphs, But unfortunately, it is proved to be undecidable in automata theory. Semantic equivalence is widely used in program verification to overcome the above difficulties. Bisimilarity is a kind of semantic equivalence, which is focused on the observable behavior of the graphs.

On unordered graphs, we use the notion of value equivalence (Bisimilarity) as the equivalence relation used in verfication. A similar bisimilarity on ordered graphs is given.

For two graphs $G = (V, B, I)$and $G' = (V', B', I')$ in $\mathcal{G}^{cX}_{Y}$, a relation R between V and V' is called a *bisimulation relation*, if for any vRv', there is an order isomorphism $f : (Pb(G, v), \leqslant_{Pb}) \to (Pb(G', v'), \leqslant_{Pb})$ satisfying he following order-preserving property: For any proper branch $p = (v\xrightarrow{\epsilon}{}_{i_0}v_1 \ldots \xrightarrow{\epsilon}{}_{i_{n-1}}v_n\rightarrow{}_{i_n}) \in Pb(G, v)$ with $f(p) = (v\xrightarrow{\epsilon}{}_{i'_0}v'_1 \ldots \xrightarrow{\epsilon}{}_{i'_{n'-1}}v'_{n'}\rightarrow{}_{i'_{n'}}) \in Pb(G', v')$, we have

- Edge Correspondence: if $B(v_n).i_n = \mathbf{Edge}(l, u)$ for some $l \in \mathcal{L}, u \in V$, then $\exists u' \in V'$ s.t. $B'(v'_{n'}).i'_{n'} = \mathbf{Edge}(l, u')$ and uRu',
- Marker Correspondence: if $B(v_n).i_n = \mathbf{Outm}(\&y)$ for some $\&y \in Y$, then $B'(v'_{n'}).i'_{n'} = \mathbf{Outm}(\&y)$.

Two graphs G and G' are *bisimilar* (denoted by $G \sim G'$) if there is a bisimulation relation R s.t. for every input marker $\&x \in X$, $I(\&x)RI'(\&x)$.

The notion of bisimilarity only consider the reachable part of a given graph. This means the unreachable parts are disregarded, i.e. two bisimilar garphs are still bisimilar if one adds subgraphs unreachable frome input nodes.

3 λ_{FG}: A Transformation Language for Ordered Graphs

3.1 The Core λ_{FG}

λ_{FG} is a transformation language for transforming (querying) ordered graphs. There is a formal semantics for constructing and querying an ordered graph in λ_{FG}. It is an extenstion of typed λ-calculus with graph constructors and list functions. It has a powerful feature of structural recursion. It is capable of querying with manipulating the sibling of nodes, which gives a great expresiveness power to the language. The syntex of λ_{FG}

is given as below. It also has a system of strictly defined type inference rules.

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

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

Here we focus on the explaination of *structural recursion*, which is a powerful graph transformation mechanism. It provides the capability to descirbe the queries and transformations that garantees the termination of the computation and preserves the finiteness.

In general the structure recursion is written of the form $\mathbf{srec}(e, d)$, wherer e can be considered as the body function applied on the labeled edges and d is a map computation on nodes. For a simple case, we can consider $d = \mathbf{foldr}(\odot, \iota_{\odot})$ for some monoid $(\odot, \iota_{\odot})$ on $\mathbf{G}^{Z}_{Z \times \alpha}$. The structural recursion function $f \overset{def}{=} \mathbf{srec}(e, d)$ satisfies the following equations (herer, equality means bisimilarity):

$$
\begin{array}{ll}
f(\underline{[\,]}) & = \iota_{\odot} \\
f(g_1 \underline{+\!\!+} g_2) & = f(g_1) \iota f(g_2) \\
f(\underline{[}l : g\underline{]}) & = e(l, g) @ f(g) \\
f(\&x := g) & = (\oplus_{\&z \in Z}(\&z, \&x) := \underline{[}(\&z, \&\underline{]}) @ f(g) \\
f(()) & = () \\
f(g_1 \oplus g_2) & = f(g_1) \oplus f(g_2)
\end{array}
$$

Example 1. This example shows how to manipulate edges of the graph and change its shape by structural recursion, The following function *a*2*d*_*xc* replaces all labels a with d and contracts c-labeled edges.

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

Compared with the structural recurcion in UnCAL, structural recurtion in λ_{FG} can deal with orered graphs and computations on the sibling dimension of graphs. This ability is given by the function of d.

3.2 Bulk Semantics of Structural Recursion

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 a whole. It is useful in parallel computation and the proof of termination and finiteness-preserving property. In the construction of bidirectional semantics, we will also apply the bidirectionlization on bulk semantics.

Before the introduction of bulk semantics, let us consider the type of the structural recursion function **srec**(e, d) in advance. A type inference rule is given below.

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

Here, e is the body function of the transformation on edges. it performs a similar role as the body function in the unordered version. It maps the a subgraph of a labeled edge into the result graph. For the d function, it works as a rearrangement function over the lists. It maps the each lists by manipulate the order of the branches. In the bulk semantics, the evaluation can be applied in parallel. It mainly involves three steps.

1. Map computation on edges with e: the function e is applied on each labeled edge, yield a set of intermediate result graphs for e. The set graphs will be rearranged in the next step.
2. Map computation on nodes with d: the function d is applied on each nodes. The previous set of graphs are merged by the rearrangement measure defined in d, leads to a set of merged graphs for each node.
3. Groups new graphs with ϵ-edges: to get the final result, the graphs for each node need to be grouped by ϵ-edges. The ϵ-edges are added according to the input and output markers of the previous result.

4 An Overview of Bidirectional Transformation

In UnCAL, there is a framework that could bidirectionalize unordered graph transformation. It consists of a two stage bidirectionalizing strategy and it could solve the problem that the graphs may contain shared nodes or cycles. This framework provides us a some heuristic idea to do the bidirectionalizing on λ_{FG}.

The challenging parts of bidirectionalizing on λ_{FG} comes from the d function in the structural recursion. It leads to a three step bulk semantics, which would make it hard to trace back from the view. Further, the d function only accepts graphs without ϵ-edges. Thus, an ϵ-elimination procedure has to be applied before the structural recursion for an arbitrary graph. Also, the ϵ-edge should be eliminated before it is presented to the customer. Moreover, the order of the branchings must be treated carefully in our bidirectionalization.

4.1 Bidirectional Properties

The goal in bidirectionalizing λ_{FG} is to approach the bidirectionalization in ordered graphs by providing a bidirectional semantics in λ_{FG}. The semantics should have bidirectional properties defined in the previous work in UnCAL. Let $\mathscr{F}[\![e]\!]\rho$ denote a forward evaluation (get) of expression e under environment ρ to produce a view, and $\mathscr{B}[\![e]\!](\rho, G')$ denote a backward evaluation (put) of expression e under environment ρ to reflect a possibly modified view G' to the source by computing an updated environment. 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 need to satisfy the following two important properties:

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

The (GETPUT) property states that unchanged view G should give no change on the environment ρ in the backward evaluation, while the (WPUTGET) property states that the modified view G' and the view obtained by backward evaluation followed by forward evaluation may differ, but both view have the same effect on the original source if backward evaluation is applied. A pare of forward and backward evaluation is *well-behaved* if it satisfies (GETPUT) and (WPUTGET) proties. In the rest of the paper, the forward and backward evaluations will be presented for λ_{FG}, and the following theorem holds.

Theorem 1 (Well-behavedness). *The proposed forward and backward evaluations 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. When we perform the modification on the view, the most reasonable choice is to reflet it on the corresponding part on the source. This is the *location correspondence* property. Moreover, the modification on the view graph could be categorised into three types: in place updates, edge deletion and edge insertion. In order to maintain the well-behavedness between the updated source and the modified view, we also need to apply some operation on the source. A reasonable assumption is that a certain type of modification on the view yields the same type of operation on the source, which could be called the *operation correspondence* property. We say this kind of reflection on updates are rational.

The rationality property is also important as it gives us a discriptive goal in designing the bidirectionalizing the transformation. The goal is to let the updates on the source to meet the intention of the changing on the view.

4.2 Three-Stage Bidirectionalization

In an arbitrary transformation, there could be a big gap between the source and view graoh. It makes it hard to reflect changes on the view to the source. The basic idea to divide the forward evaluation into three stages, each one could be bidirectionalized.

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

In our bidirectional transformation framework, we will propose 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 this paper. Thus, the whole transformation is bidirectionlized by a inductively defined procedure.

5 Bidirectionalizing ϵ-elimination procedure

In the previous work of λ_{FG}, the input graph of the d function in structural recursion does not accept any ϵ-edges. However, we can not safely assume that an arbitrary input graph does not contain any ϵ-edges. Further, after applying the transformation via bulk semantics, some ϵ-edges might be introduced to maintain the structure of the result graph. In most applications, ϵ-edges are assumed to be unobservable. Thus, we wish to produce a view without any ϵ to the users. In this seciton, a bidirectionalized ϵ-elimination procedure is proposed.

First, there is an effective procedure that whether the ϵ-elimination could be applied successfully on a finte ordered graph G. The details could be found in the paper of λ_{FG}.

Lemma 1. *Given an finite ordered graph G possibly with ϵ-edges, there is an effective procedure to decide whether there exists an ordered graph G', s.t. $G \sim G'$ and has neither ϵ-edges nor infinite width. If yes, produce such G'.*

The idea of the ϵ-elimination procedure is to use labeled edges to connect the shortcut between proper branches, the procedure is defined as below. Note that we use a list of integers to represent the total orders over the branches and proper branches on a node.

Definition 1. *For a graph $G = (V, B, I) \in \mathcal{G}^{cX}_{Y}$, the ϵ-elimination ϵ-elim(G) of G is a graph $(V, B'I) \in \mathcal{G}^{cX}_{Y}$ where $|B'(v)| \overset{def}{=} Pb(G, v)$, and $B'(v).\tilde{p} \overset{def}{=} Pb(G, v).\tilde{p}.last$ for $p = (v \overset{\epsilon}{\longrightarrow}{}_{i_0} \ldots v_n \longrightarrow{}_{i_n})$ in $B'(v)$.*

Next we will introduce how to reflect view updates within ϵ-elim(G) to the source graph G. We will use the correspondence between the proper branches in the source graph and the branches in the view graph.

1. For the inplace-updates on an arbitrary edge $B'(v).\tilde{p} \in \epsilon$-elim$(G)$, the corresponding labeled edge on the view is $Pb(G, v).\tilde{p}.last$. We could apply the same update operation on this edge.
2. For the edge deletion on an arbitrary edge $B'(v).\tilde{p} \in \epsilon$-elim$(G)$, we update the source graph by delete the correcponding edge $Pb(G, v).\tilde{p}.last$.

3. For the edge insertion, since the set of nodes V is the same on the view and the source. Thus, we just need to insert the edge to the same place 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 proper branches. Thus, the corresponding edge of these proper branches on the view graph also need to be updated to mainain the consistency between the source and the view. For inplace updates and edge deletion, the corresponding edge is deleted on the view, and for edge insertion, the edge is added according to the newly added proper branches. Therefore, the whole procedure would satisfy the well-behavedness. Moreover, the rationality is also infered from this constructive bidirectionalizing procedure.

Lemma 2. *The proposed bidirectionalized ϵ-elimination procedure satisfy the well-behavedness and the rationality property.*

6 Traceble Forward Evaluation

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. A backward transformation for this procedure specifies how to reflect view updates to the source graph. However, the structural information kept by ϵ-edges will not appear explicitely in the view graph. Moreover, the newly created parts within the transformation procedure also make it hard to detect the generated source for everything in the view graph. In this sence, some appropriate trace information 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 make it produces traceable view after forward transformation.

6.1 Local Trace Information

Recall the semantics in λ_{FG}, a view is obtained by evaluating a λ_{FG} expression with an ordered graph. Every node in the view graph is either a graph obtained from the source graph, or a node constructed by te λ_{FG} expression, except the node generated through a structural recursion(in bulk semantics). For an arbitrary structural recursion function $\mathbf{srec}(e_1, d)(e_2)$, the input of the body function e_1 is evaluated binding variables from a part of evaluated result in e_2; and the input of the rearrangement d function is evaluated by binding variables from the result in e_1. So the node in the result graph may originate from not only the whole *srec* expression but also a sub-expression in e_2. Thus, a more complicated inductively defined trace information should be added on the nodes on the result graph.

Here, we use the a *traceable view* in which each node has information for tracing its origin. This locally added information, called $TraceID$, is defined by

$$\begin{aligned} TraceID \coloneqq\ & SrcID \\ \mid\ & Code\, Pos\, Marker \\ \mid\ & RecN\, Pos\, TraceID\, Marker \\ \mid\ & RecE\, Pos\, TraceID\, TraceID\, Num \\ \mid\ & RecM\, Pos\, Marker\, TraceID\, Num \\ \mid\ & RecD\, Pos\, TraceID\, TraceID \end{aligned}$$

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 order of the branches.

Now we briefly expain the meaning of each $TraceID$. $SrcID$ means the node is generated from the corresponding one in the source. $Code\, Pos\, Marker$ is the information for the nodes created by the λ_{FG} expreesion on the marked code position. $RecN$, $RecE$, $RecM$, $RecD$ are four kinds of trace information used in bidirectionalizing structural recursion $\mathbf{srec}(e, d)$. $RecN\, RecE\, RecM$ represent the node in the intermediate result after e is applied, which are generated from the nodes/edges/markers respectively, and $RecD$ information is added when the rearrangements on the lists are applied by d.

6.2 Enriched Forward Semantics

Now we will define the enriched forward semantics of λ_{FG} that could generate a traceable view. Recall the core of λ_{FG} language introduced in 3.1, the tracing information is recorded when a node is created. Let e^p denote an λ_{FG} subexpression e at code position p. We write $\rho(\$x)$ for G when $(\$x \mapsto G) \in \rho$, and $e\rho$ means the variable substitution using ρ in the expression e. We inductively define the enriched forward semantics $\mathscr{F}\llbracket e^p \rrbracket \rho$ for each λ_{FG} construct of e.

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

Graph Constructor Expressions The semantics of graph constructor expressions is straightforward according to the construction in Fig1. For instance, the enriched semantic of single node graph constructor is,

$\mathscr{F}\llbracket \underline{[]}^p \rrbracket \rho = (\{Code\, p\}, \{Code\, p \mapsto []\}, \{\& \mapsto Code\, p\})$

Note that the code position is added in the node, which infers that it is generated by the expression of in position p. Moreover, the branch set is nonempty despite there is no edge in the graph.

As another example, the semantics for the expression $e_1 \underline{++} e_2$ is defined below. Note that there is no such combinator in unordered version for that it remains the order of the graphs when it links the input markers.

$\mathscr{F}\llbracket (e_1 \underline{++} e_2)^p \rrbracket \rho = \mathscr{F}\llbracket e_1 \rrbracket \rho \underline{++}^p \mathscr{F}\llbracket e_2 \rrbracket \rho$

$\mathbf{let}\, G_1 \underline{++}^p G_2 = (V_1 \cup V_2 \cup V', B_1 \cup B_2 \cup B', I')$

$\mathbf{where}\, (V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2$

$M = \mathbf{inMarker}(G_1) = \mathbf{inMarker}(G_2)$

$V' = \{Code\, p\, \&m \mid \&m \in M\}$

$B' = \{Code\, p\, \&m \mapsto [\mathbf{Edge}(\epsilon, I_1(\&m)), \mathbf{Edge}(\epsilon, I_2(\&m))] \mid \&m \in M\}$
$I' = \{\&m \mapsto Code\, p\, \&m \mid \&m \in M\}$

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.

$e_1 \oplus e_2$ is a componentwise union operator like $\underline{+\!\!+}$, exept that no ϵ-edges are involved.

$\mathscr{F}\llbracket (e_1 \oplus e_2)^p \rrbracket \rho = \mathscr{F}\llbracket e_1 \rrbracket \rho \oplus \mathscr{F}\llbracket e_2 \rrbracket \rho$
$\mathbf{let}\, G_1 \oplus G_2 = (V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2)$
$\mathbf{where}\, (V_1, B_1, I_1) = G_1; (V_2, B_2, I_2) = G_2$

For the empty graph constructor and constant marker graph, it is easy to define

$\mathscr{F}\llbracket ()^p \rrbracket \rho = (\{\}, \{\}, \{\})$

$\mathscr{F}\llbracket \underline{[}\&m\underline{]}^p \rrbracket \rho = (\{Code\, p\}, \{Code\, p \mapsto [\mathbf{Out}m(\&m)]\}, \{\& \mapsto Code\, p\})$

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

$\mathscr{F}\llbracket \underline{[}l : e\underline{]}^p \rrbracket \rho = (\{Code\, p\} \cup V, \{Code\, p \mapsto [(l\rho, I(\&))]\} \cup B, \{\& \mapsto Code\, p\})$
$\mathbf{where}\, (V, B, I) = \mathscr{F}\llbracket e \rrbracket \rho$

TraceID is generated for the newly constructed node.

$\mathscr{F}\llbracket (\&m := e)^p \rrbracket \rho = (\&m := \mathscr{F}\llbracket e \rrbracket \rho)$
$\mathbf{let}\, (\&m := G) = (V, B, I')$
$\mathbf{where}\, (V, B, I) = G$
$I' = \{\&m.\&x \mapsto v \mid (\&x \mapsto v) \in I\}$

Here "." is the Skolem function introduced before.

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

$\mathscr{F}\llbracket (e_1 @ e_2)^p \rrbracket \rho = \mathscr{F}\llbracket e_1 \rrbracket \rho\, @^p \mathscr{F}\llbracket e_2 \rrbracket \rho$
$\mathbf{let}\, G_1 @^p G_2 = (V_1 \cup V_2, B_1' \cup B_2, I_1)$
$\mathbf{where}\, (V_1, B_1, I_1) = G_1,\ (V_2, B_2, I_2) = G_2$

$$B_1' = \{u \mapsto \left[\begin{array}{ll} x.i = \mathbf{Edge}(l, v) & \Rightarrow x.i \\ \quad = \mathbf{Outm}(\&m) & \Rightarrow (\epsilon, I_2(\&m)) \end{array} \right]_{i \in |x|} \mid (u \mapsto x) \in B_1\}$$

$\mathbf{cycle}(e)$ is defined as follows

$\mathscr{F}\llbracket (\mathbf{cycle}(e))^p \rrbracket \rho = \mathbf{cycle}^p(\mathscr{F}\llbracket e \rrbracket \rho)$
$\mathbf{let}\, \mathbf{cycle}^p(G) = (V, B', I)$
$\mathbf{where}\, (V, B, I) = G$

$$B' = \{u \mapsto \left[\begin{array}{ll} x.i = \mathbf{Edge}(l, v) & \Rightarrow x.i \\ \quad = \mathbf{Outm}(\&m) \wedge \&m \in dom(I) & \Rightarrow \mathbf{Edge}(\epsilon, I(\&m)) \\ \quad = \mathbf{Outm}(\&m) \wedge \&m \notin dom(I) & \Rightarrow x.i \end{array} \right]_{i \in |x|} \mid (u \mapsto x) \in B\}$$

Labels In λ_{FG} the type of labels and boolean is strictly defined. The forward semantics for label and label equality are

$\mathscr{F}\llbracket a^p \rrbracket \rho = \quad a\rho \quad (a \in \mathcal{L})$
$\mathscr{F}\llbracket (l_1 = l_2)^p \rrbracket \rho = \quad l_1 \rho = l_2 \rho$

Variables The query of ordered graph via λ_{FG} requries the variable abstraction and application operations. The forward semantics for variable looks up its binding environment ρ is.

$\mathscr{F}\llbracket(\$v)^p\rrbracket\rho = \rho(\$v)$

Condition The forward semantics of a condition is defined as

$$\mathscr{F}\llbracket(\textbf{if}\, e_1 \,\textbf{then}\, e_2 \,\textbf{else}\, e_3)^p\rrbracket\rho = \begin{cases} \mathscr{F}\llbracket e_2\rrbracket\rho \ \textbf{if}\ \mathscr{F}\llbracket e_1\rrbracket\rho \\ \mathscr{F}\llbracket e_3\rrbracket\rho \ \mathit{otherwise} \end{cases}$$

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

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

$\mathscr{F}\llbracket(\textbf{isEmpty}(e))^p\rrbracket\rho = \textbf{isEmpty}(\mathscr{F}\llbracket e\rrbracket\rho)$

Function on Lists For the function on the lists, here we consider it as a polymorphism on the nodes for simplicity, There is a formal way of bidirectionalizing the node polymorohism, which is done by previous work. So we will not include the details.

Structural Recursion The forward semantics of a structural recursion in λ_{FG} is given by *bulk semantics*. It can be formally defined by three stages:

$\mathscr{F}\llbracket(\textbf{srec}(e_b, d)(e_a))^p\rrbracket\rho = G_{srec}$

$\textbf{let}\, G = \mathscr{F}\llbracket e_a\rrbracket\rho, \quad Z = \mathit{Marker}(e_b)$

$\textbf{where}\, (V, B, I) = G$ (G has no ϵ-edges), $\quad G|_v = (V, B, \{\& \mapsto v\})$

1. First apply edge mappings using e.

$$B_v = \{u \mapsto \begin{bmatrix} x.i = \textbf{Edge}(l_i, v_i) \Rightarrow (\mathscr{F}\llbracket e_b\rrbracket\rho_{u,i}, v_i) \\ \rho_{u,i} = \rho \cup \{\$l \mapsto l_i, \$g \mapsto G|_{v_i}\} \\ x.i = \textbf{Out}m(\&y) \Rightarrow x.i \end{bmatrix}_{i\in|x|} |(u \mapsto x) \in B\}$$

Then use *RecN*, *RecE*, *RecM* to mark the nodes within the graph generated by the nodes, edges and markers respectively,

$V_{RecE_{u,i}} = \{RecE\ p\, w\, u\, i|(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = (G_{u,i}, v),\ w \in G_{u,i}.V\}$

$$B_{RecE_{u,i}} = \{RecE\ p\, w\, u\, i \mapsto \begin{bmatrix} x'.j = \textbf{Edge}(l_j, w_j) \Rightarrow \textbf{Edge}(l_j, RecE\ p\, w_j\, u\, i) \\ x'.j = \textbf{Out}m(\&m) \Rightarrow \textbf{Out}m(\&m, \&RecN\ p\, v\, \&m) \end{bmatrix}_{j\in|x'|} |(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = (G_{u,i}, v),\ (w \mapsto x') \in G_{u,i}.B,\ \&m \in Z\}$$

$I_{RecE_{u,i}} = \{\&m \mapsto RecE\ p\, w\, u\, i|(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = (G_{u,i}, v),\ (\&m \mapsto w) \in G_{u,i}.I\}$

$V_{RecM_{u,i}} = \{RecM\ p\, \&m\, u\, i|(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = \textbf{Out}m(\&y),\ \&m \in Z\}$

$B_{RecM_{u,i}} = \{RecM\ p\, \&m\, u\, i \mapsto [\textbf{Out}m(\&m, \&y)]|(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = \textbf{Out}m(\&y),\ \&m \in Z\}$

$I_{RecM_{u,i}} = \{\&m \mapsto RecM\ p\, \&m\, u\, i|(u \mapsto x) \in B_v,\ i \in |x|,\ x.i = \textbf{Out}m(\&y),\ \&m \in Z\}$

$V_{RecN} = \{RecN\ p\, v\, \&m|v \in V\ \&m \in Z\}$

$$B_{RecN} = \{RecN\ p\, u\, \&m \mapsto \begin{bmatrix} x.i = (G_{u,i}, v) \Rightarrow (V_{RecE_{u,i}}, B_{RecE_{u,i}}, I_{RecE_{u,i}}) \\ x.i = \textbf{Out}m(\&y) \Rightarrow (V_{RecM_{u,i}}, B_{RecM_{u,i}}, I_{RecM_{u,i}}) \end{bmatrix}_{i\in|x|} |u \in V,\ (u \mapsto x) \in B_v,\ \&m \in Z\}$$

2. Then apply node mapping using d.

$G_{d_v} = \mathscr{F}\llbracket d_v\rrbracket\rho_{d_v}$

$\textbf{where}\, v \in V_{RecN}, \quad \rho_{d_v} = \{\$b \mapsto B_{RecN}(v)\}$

3. We add *RecD* information on the result graphs and merge them by ϵ-edges.
$V_{RecD_v} = \{RecD\, p\, u\, v | v \in V_{RecN},\ u \in G_{d_v}.V\}$
$$B_{RecD_v} = \{RecD\, p\, u\, v \mapsto \begin{bmatrix} x.i = \textbf{Edge}(l_i, w_i) & \Rightarrow \textbf{Edge}(l_i, RecD\, p\, w_i\, v) \\ x.i = \textbf{Out}m(\&m, \&v') & \Rightarrow \textbf{Edge}(\epsilon, I_{RecD_v}(\&m, \&v')) \\ x.i = \textbf{Out}m(\&m, \&y) & \Rightarrow x.i \end{bmatrix}_{i \in |x|} |v \in$$
$V_{RecN},\ (u \mapsto x) \in G_{d_v}.B,\ \&m \in Z,\ v'(\neq v) \in V_{RecN}\}$
$I_{RecD_v} = \{(\&m, \&v) \mapsto RecD\, p\, u\, v | v \in V_{RecN},\ \&m \in Z,\ u = G_{d_v}.I(\&m)\}$
$I_{RecD} = \{(\&m, \&y) \mapsto u | \&m \in Z,\ \&y \in Dom(I),\ v = I(\&y),\ v' = RecN\, p\, v\, \&m,\ u =$
$I_{RecD_{v'}}(\&m, \&v')\}$
The final result is
$G_{srec} = (\bigcup_{v \in V_{RecN}} V_{RecD_v}, \bigcup_{v \in V_{RecN}} B_{RecD_v}, I_{RecD})$

7 Backward Evaluation

Thanks to the bulk semantics in λ_{FG} and the trace information generated at enriched forward evaluation, the similarity in shape between input and output graphs are retained. As in previous work of unordered graph, the graph constructors become invertible, and the backward evaluation $\textbf{rec}(e, d)$ is reduced to that of its body e and d. We again devide backward evaluation algorithm by updates supported: (1) edge-renaming, (2) edge-deletion, and (3) insertion of edges or a subgraph rooted at a node. In the following, we give the backward semantics for these updates.

7.1 Reflection of Inplace Updates

Backward semantics is, similarly to the forward semantics, defined inductively, i.e. backward evaluation of a λ_{FG} expression is computed by combining results of backward evaluations of the operand expressions. We first define backward semantics of the simple expressions, followed by that of structural recursion.

Graph Constructor Expressions

In order to merge the environment computed by the inductively defined,backward evaluation, we need a to define a merge operator.

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

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

$\underline{[]}$, $\&y$ and $()$ construct constant graphs in the forward computation. Therefore, for the backward computation, they accept no modification on the view.

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

$\underline{[}l : e\underline{]}$ backward computation detaches (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)

$\mathscr{B}\llbracket \underline{[}l : e\underline{]}^p \rrbracket(\rho, G') = \mathscr{B}\llbracket l \rrbracket(\rho, a') \uplus_\rho \mathscr{B}\llbracket e \rrbracket(\rho, G'_2)$

where $a = l\rho$

$G_2 = \mathscr{F}\llbracket e \rrbracket\rho$

$(a', G'_2) = decomp_{\underline{[}a:^p G_2\underline{]}}(G')$

Here, the decompositon function is defined as follows

$decomp_{\underline{[}a_1:^p G_2\underline{]}}(G') = \quad (a'_1, (V' \backslash \{r'\}, B' \backslash \{r' \mapsto \zeta'\}, \{\& \mapsto v\}))$

where $(V_2, B_2, \{\& \mapsto v\}) = G_2$

$(V', B', \{\& \mapsto r'\}) = G'$

$\zeta' = \ the\,unique\,branch\,in\,B'\,of\,the\,form\,([\mathbf{Edge}(a'_1, v)]).$

The modified view G' is decomposed into its unique root branch $\zeta' = [\mathbf{Edge}(a'_1, v)]$ 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.

$e_1 \underline{+\!\!+} e_2$ We have the following definition ,given $decomp_{G_1 \underline{+\!\!+} G_2)}(G')$ as the decomposition of the graph G'

$\mathscr{B}\llbracket (e_1 \underline{+\!\!+} e_2)^p \rrbracket(\rho, G') = \mathscr{B}\llbracket e_1 \rrbracket(\rho, G'_1) \uplus_\rho \mathscr{B}\llbracket e_2 \rrbracket(\rho, G'_2)$

where $G_1 = \mathscr{F}\llbracket e_1 \rrbracket\rho$

$G_2 = \mathscr{F}\llbracket e_2 \rrbracket\rho$

$(G'_1, G'_2) = decomp_{G_1 \underline{+\!\!+}^p G_2}(G')$

$decomp$ is defined as below. $G_1 \backslash G_2$ denotes $(V_1 \backslash V_2, B_1 \backslash B_2, I_1 \backslash I_2)$ where $B_1 \backslash B_1$ and $I_1 \backslash I_2$ respectively denote removal of 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 be $(V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2)$.

$decomp_{G_1 \underline{+\!\!+}^p G_2}(G') = (xreachable(G''_1, G_1), xreachable(G''_2, G_2))$

where $(V', B', I') = G' \quad (V_i, B_i, I_i) = G_i$

$G''_i = reachable((V', B', I_i))$

satisfying $M = Dom(I_1) = Dom(I_2)$

$\forall \& m \in M,\ \mathbf{Edge}(\epsilon, v') \in B'(I'(\& m)) : (\& m \mapsto v') \in I_1 \cup I_2$

$unreachable(G) = G \backslash reachable(G)$

$xreachable(G'', G) = reachable(G'') \cup unreachable(G)$

$e_1 \oplus e_2$ It is like Cao, except no ϵ-edge is involved.

$\mathscr{B}\llbracket (e_1 \oplus e_2)^p \rrbracket(\rho, G') = \mathscr{B}\llbracket e_1 \rrbracket(\rho, G'_1) \uplus_\rho \mathscr{B}\llbracket e_2 \rrbracket(\rho, G'_2)$

where $G_1 = \mathscr{F}\llbracket e_1 \rrbracket\rho$

$G_2 = \mathscr{F}\llbracket e_2 \rrbracket\rho$

$(G'_1, G'_2) = decomp_{G_1 \oplus^p G_2}(G')$

$decomp_{G_1 \oplus^p G_2}(G') = (xreachable(G''_1, G_1), xreachable(G''_2, G_2))$

where $(V', B', I') = G' \quad (V_i, B_i, I_i) = G_i$

$G''_i = reachable((V', B', I_i))$

It is the same as $decomp_{G_1 \underline{+\!\!+}^p G_2}(G')$ without **satisfying** condition.

$e_1 @ e_2$ Because of unmatched I/O nodes, it may introduce unreachable part in the second argument during forward computation. Backward computation carefully passes those parts backwards untouched to avoid unnecessary failure because of inconsistency, and these parts are part of ordinary computation (computation on reachable parts) before discarding by the @ operator.

$\mathscr{B}\llbracket (e_1 @ e_2)^p \rrbracket(\rho, G') = \mathscr{B}\llbracket e_1 \rrbracket(\rho, G'_1) \uplus_\rho \mathscr{B}\llbracket e_2 \rrbracket(\rho, G'_2)$

where $G_1 = \mathscr{F}\llbracket e_1 \rrbracket\rho$

$$G_2 = \mathscr{F}\llbracket e_2 \rrbracket\rho$$

$$(G'_1, G'_2) = decomp_{G_1 @^p G_2}(G')$$

$$decomp_{G_1 @^p G_2}(G') = (xreachable(G''_1, G_1), xreachable(G''_2, G_2))$$

$$\textbf{where}\ (V', B', I') = G' \quad (V_i, B_i, I_i) = G_i$$

$$B'' = \{u \mapsto \left[\begin{array}{ll} x.i = \textbf{Edge}(\epsilon, v) \wedge u \in V_1 & \\ \wedge B_1(u).i = \textbf{Out}m(\&m) & \\ \wedge (\&m \mapsto v) \in I_2 & \Rightarrow \textbf{Out}m(\&m) \\ otherwise & \Rightarrow x.i \end{array}\right]_{i\in|x|} \quad |(u \mapsto x) \in B'\}$$

$$G''_i = reachable((V', B'', I_i))$$

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

$$\mathscr{B}\llbracket (\&m := e)^p \rrbracket(\rho, G') = \mathscr{B}\llbracket e \rrbracket(\rho, G'_1)$$

$$\textbf{where}\ (V', B', I') = G'$$

$$I'_1 = \{(\&x \mapsto v) | (\&m.\&x \mapsto v) \in I'\}$$

$$G'_1 = (V', B', I'_1)$$

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

$$\mathscr{B}\llbracket (\textbf{cycle}(e))^p \rrbracket(\rho, G') = \mathscr{B}\llbracket e \rrbracket(\rho, G'_2)$$

$$\textbf{where}\ (V', B', I') = G' \quad (V, B, I) = \mathscr{F}\llbracket e \rrbracket\rho$$

$$B_{cycle} = \{u \mapsto \left[\begin{array}{ll} x.i = \textbf{Edge}(\epsilon, v) & \\ \wedge B(u).i = \textbf{Out}m(\&m) & \Rightarrow \textbf{Out}m(\&m) \\ otherwise & \Rightarrow x.i \end{array}\right]_{i\in|x|} \quad |(u \mapsto x) \in B'\}$$

$$G'_2 = (V, B_{cycle}, I)$$

Labels A label constant and label equality accepts no modification.

$$\mathscr{B}\llbracket a \rrbracket(\rho, a') = \rho \quad \textbf{if}\ a' = a$$

$$\mathscr{B}\llbracket (l_1 = l_2) \rrbracket(\rho, e') = \rho \quad \textbf{if}\ e' = \mathscr{F}\llbracket (l_1 = l_2) \rrbracket\rho$$

Variables A variable simply updates its bindings as $\mathscr{B}\llbracket \$v \rrbracket(\rho, G') = \rho[\$v \leftarrow G']$. Here, $\rho[\$v \leftarrow G']$ is an abbreviation for $(\rho \backslash \{\$v \mapsto \_\}) \cup \{\$v \mapsto G'\}$.

Condition A backward evaluation of a condition is defined by

$$\mathscr{B}\llbracket (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \rrbracket(\rho.G') = \begin{cases} \rho'_2 & \textbf{if}\ \mathscr{F}\llbracket e_1 \rrbracket\rho \wedge \mathscr{F}\llbracket e_1 \rrbracket\rho'_2 \\ \rho'_3 & \textbf{if}\ \neg\mathscr{F}\llbracket e_1 \rrbracket\rho \wedge \neg\mathscr{F}\llbracket e_1 \rrbracket\rho'_3 \\ FAIL\ otherwise & \end{cases}$$

$$\textbf{where}\ (\rho'_2, \rho'_3) = (\mathscr{B}\llbracket e_2 \rrbracket(\rho, G'), \mathscr{B}\llbracket e_3 \rrbracket(\rho, G'))$$

which 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 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 value of variables in the condition is produced from the updated bindings, and check that condition at the expression binding the variables. However, to simplify presentation, we omit this passing around of conditions and checkings in the following.

Emptiness A backward evaluation of an emptiness checking is defined by

$$\mathscr{B}\llbracket \textbf{isEmpty}(e) \rrbracket(\rho, e') = \rho \quad \textbf{if}\ e' = \mathscr{F}\llbracket \textbf{isEmpty}(e) \rrbracket\rho$$

As the reason explained before, we simply does not allow the change of boolean values in the backward evaluation.

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 of structural recursion to decompose the view graph into subgraphs before adding the ϵ-edges. Further, when we merge the updated environment, we first merge local changes, then merge it to the whole changes. Since there might be overlapped parts among theng, we let the backward evaluation fails if the overlapped parts are inconsistant. The backward evaluation of the body expression can be inductively defined.

$$\mathscr{B}[\![\mathbf{srec}(e_b, d)(e_a)]\!](\rho, G'_{srec}) = \rho'$$
$$\textbf{where } (V'_{srec}, B'_{srec}, I'_{srec}) = G'_{srec}$$

1. We first decompose the result graph into subgraphs by the trace information on *RecD* functions.
$$V'_{d_v} = \{u | v \in V_{RecN},\ (RecD\ p\ u\ v) \in V'_{srec}\}$$
$$B'_{d_v} = \{u \mapsto \left[\begin{array}{ll} x.i = \mathbf{Edge}(l_i, RecD\ p\ w_i\ v) & \Rightarrow \mathbf{Edge}(l_i, w_i) \\ x.i = \mathbf{Edge}(\epsilon, I_{RecD_v}(\&m, \&v')) & \Rightarrow \mathbf{Out}m(\&m, \&v')) \\ x.i = \mathbf{Out}m(\&m, \&y) & \Rightarrow x.i \end{array}\right]_{i \in |x|} | v \in V_{RecN},\ ((RecD\ p\ u\ v) \mapsto x) \in B'_{srec},\ \&m \in Z,\ v'(\neq v) \in V_{RecN}\}$$
$$I'_{d_v} = \{\&m \mapsto u | v \in V_{RecN},\ \&m \in Z,\ RecD\ p\ u\ v \in Range(I_{RecD_v})\}$$
$$G'_{d_v} = (V'_{d_v}, B'_{d_v}, I'_{d_v})$$
2. Then apply the backward evaluation of function d on each subgraph. Applying decomposition on the result using the trace information of *RecE* and *RecD*.
$$\rho'_{d_v} = \mathscr{B}[\![d_v]\!](\rho_{d_v}, G'_{d_v})$$
$$V'_{u,i} = \{v | u \in V,\ (RecN\ p\ u\ \&m \mapsto x) \in \rho'_{d_{RecN\ p\ u\ \&m}}(\$b),\ i \in |x|,\ (RecE\ p\ v\ u\ i) \in x.i.V,\ \&m \in Z\}$$
$$B'_{u,i} = \{v \mapsto \left[\begin{array}{ll} x'.j = \mathbf{Edge}(l_j, RecE\ p\ v_j\ u\ i) & \Rightarrow \mathbf{Edge}(l_j, v_j) \\ x'.j = \mathbf{Out}m(\&m, \&(RecN\ p\ u'\ \&m)) & \Rightarrow \mathbf{Out}m(\&m) \end{array}\right]_{j \in |x'|} | u \in V,\ (RecN\ p\ u\ \&m \mapsto x) \in \rho'_{d_{RecN\ p\ u\ \&m}}(\$b),\ i \in |x|,\ ((RecE\ p\ v\ u\ i) \mapsto x') \in x.i.B\}$$
$$I'_{u,i} = \{\&m \mapsto w | u \in V,\ (RecN\ p\ u\ \&m \mapsto x) \in \rho'_{d_{RecN\ p\ u\ \&m}}(\$b),\ i \in |x|,\ x.i.I(\&m) = (RecE\ p\ w\ u\ i)\}$$
$$G'_{u,i} = (V'_{u,i}, B'_{u,i}, I'_{u,i})$$
3. Apply the backward evaluation of e. Finally, reconstruct the updated view.
$$\rho'_{u,i} = \mathscr{B}[\![e_b]\!](\rho_{u,i}, G'_{u,i})$$
$$G' = (V' \cup \bigcup \rho'_{v,i}(\$g).V, B' \cup \bigcup \rho'_{v,i}(\$g), B, I)$$
$$\textbf{where } V' = \{\{v\} \cup \{w | Edge(l, w) \in s\} | (v \mapsto s) \in B'\}$$
$$B' = \{v \mapsto \left[\begin{array}{ll} x.i = \mathbf{Edge}(l_i, v_i) & \Rightarrow \mathbf{Edge}(\rho'_{v,i}(\$l), v_i) \\ x.i = \mathbf{Out}m(\&m) & \Rightarrow x.i \end{array}\right]_{i \in |x|} | (v \mapsto x) \in B\}$$
$$\rho' = \mathscr{B}[\![e_a]\!](\rho, G') \uplus_\rho \biguplus \{\rho'_{v,i} \backslash \{\$ \mapsto \_\} \backslash \{\$g \mapsto \_\}\}$$

7.2 Reflection of Edge Deletion

Similarly to the treatment in the previous work, we reflect deletion of an edge in the view as deletion of corresponding edge int eh source using trace IDs. In unordered graph, edge is identified by the triple of origin node, label and destination node, but here we can identify with a pair of origin node and branch position.

corr defined below takes a pair of origin node ID and branch position that uniquely identify an edge in the view, and computes the corresponding edge as a pair of origin node ID and branch position in the source.

$$corr((u,i)) = (u,i) \quad if\, u \in SrcID$$

$$corr((RecE\; p\, u\, v\, i, j)) = \begin{cases} corr((u,j)) \; if\, corr((u,j)) \neq FAIL \\ corr((v,i)) \;\; if\, corr((u,j)) = FAIL \end{cases}$$

$$corr((RecD\; p\, u\, v, j)) = corr((u,j))$$

$$corr(\zeta) = FAIL \quad otherwise$$

$FAIL$ means failure on finding the corresponding edge.

Note that only labeled edges are allowed to be deleted in the view. Thus, the nodes with trace information *RecN* and *RecM* will never appear in the input of *corr*.

Using this function, we compute the set of corresponding edges from set of deleted edges on the updated view G'_{view}. If one of the computation *corr* fails, then the backward evaluation fails. Otherwise, we compute the updated source G'_{src} by removal of the set of corresponding edges (branches). Note that we use the list of integers as the branch number, thus the relative position the following (sibling) branches will not change if we delete one edge.

Finally, get the updated view $G''_{view} = \mathscr{F}[\![e]\!]\rho[\$g \rightarrow G'_{src}]$ in order to keep the consistancy of the well-behavedness between updated source and view. Moreover, we need to check if this updated view is bisimilar to the changed view G'_{view}.

The (WPUTGET) property is straight forward, since our final check actually will promise that. Thus the backward transformation on edge deletion is also well-behaved.

7.3 Reflection of Edge Insertion

The method used in previous work of handling insertion in unordered graph 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 plase, by the following function *tr*.

$$tr(SrcID) = SrcID \quad tr(RecN\,\_\,v\,\_) = tr(v)$$

$$tr(Code\,\_\,\_) = FAIL \quad tr(RecE\,\_\,v\,\_\,\_) = tr(v)$$

$$tr(RecD\,\_\,v\,\_\,\_) = tr(v) \quad tr(RecM\,\_\,\_\,\_\,\_) = FAIL$$

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'_{view} using the Universal Resolving Algorithm (URA), wherer G'_{view} is obtained by adding ϵ-edge from v at position i to G_{vins}.

We conjecture that well-behavedness of the above insertion reflection can be directly derived from the soundness of URA.

7.4 Well-Behavedness

Now, according to the inductive way of building the bidirectional semantics, we have the well-behavedness property.

Proof. For the bidirectional evaluation semantics, this statement can be proved by induction on the structural of expression e, in a way similar to that in unordered graph. Note that we perform aggressive check for branch changing behavior of **if** for this well-behavedness. Further, the bidirectionalization of ϵ-elimination procedure is also well-behaved. Thus Theorem1 is proved.

8 Conclusion

In this paper, we give a way of bidirectionalizing the transformation on ordered graphs. A three-stage bidirectionalizing strategy is provided, and we proved that each stage could be bidirectionalized.

Acknowledgement.

References

1. François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.
2. Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. *ACM SIGPLAN Notices*, 43(1):407–419, 2008.
3. Peter Buneman, Mary Fernandez, and Dan Suciu. Unql: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal: The International Journal on Very Large Data Bases*, 9(1):76–110, 2000.
4. Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.
5. J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *ACM SIGPLAN Notices*, volume 40, pages 233–246. ACM, 2005.
6. Stephen J Hegner. Foundations of canonical update support for closed database views. In *ICDT'90*, pages 422–436. Springer, 1990.
7. Soichiro Hidaka, Kazuyuki Asada, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Structural recursion for querying ordered graphs. page 305. ACM, September 2013. ICFP'13.
8. Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.
9. Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

10. Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. *Bidirectionalization transformation based on automatic derivation of view complement functions*, volume 42. ACM, 2007.
11. Lambert Meertens. Designing constraint maintainers for user interaction, 1998.
12. Janis Voigtländer. Bidirectionalization for free!(pearl). In *ACM SIGPLAN Notices*, volume 44, pages 165–176. ACM, 2009.
13. Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining syntactic and semantic bidirectionalization. In *ACM Sigplan Notices*, volume 45, pages 181–192. ACM, 2010.
14. Meng Wang, Jeremy Gibbons, and Nicolas Wu. Incremental updates for efficient bidirectional transformations. In *ACM SIGPLAN Notices*, volume 46, pages 392–403. ACM, 2011.