ISSN 1884-0760

GRACE TECHNICAL REPORTS

Towards Bidirectional Transformations on Ordered Graphs

Soichiro Hidaka Kazuyuki Asada Hiroyuki Kato
Keisuke Nakano Zhenjiang Hu

GRACE-TR 2011–07 December 2011

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/

Towards Bidirectional Transformations on Ordered Graphs*

Soichiro Hidaka[1], Kazuyuki Asada[1], Hiroyuki Kato[1], Keisuke Nakano[2], and Zhenjiang Hu[1]

National Institute of Informatics[1]
University of Electro-Communications[2]

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 challenging problem of bidirectional transformation on graphs, where a clear bidirectional semantics is 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. In this paper, we show that the bulk semantics of structural recursion can be extended to ordered graphs, and that a clear bidirectional semantics of a new graph transformation language can be defined. The key technical point is a novel definition of bisimilarity between ordered graphs with ε edges.

Keywords: Ordered Graphs, Graph Transformation, Bisimulation

1 Introduction

Bidirectional transformations [FGM+05, CFH+09] provide a novel mechanism for synchronizing and maintaining the consistency of information between input and output. They are pervasive and have many potential applications, including the synchronization of replicated data in different formats [FGM+05], presentation-oriented structured document development [HMT08], interactive user interface design [Mee98], coupled software transformation [Läm04], and the well-known *view updating* mechanism which has been intensively studied in the database community [BS81, Heg90].

The linguistic (language-based) approach [FGM+05] gives a promising way for development of structured and well-behaved bidirectional transformation, in which every expression 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 transformations on lists and trees [FGM+05, BFP+08, MHN+07, HMT08, Voi09, VHMW10, WGW11], 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 nodes and cycles*, which makes both forward and backward computation more complicated than that on trees; naïve computation would visit the same nodes many times and possibly infinitely.

* This is a full version of the paper submitted to the First International Workshop on Bidirectional Transformations (2012). EASST style file is used to format this article.

In our previous work [HHI+10], we challenged 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 for the known graph query language UnQL [BFS00]. The key to this success is the bulk semantics of the structural recursion: a structural recursion is evaluated by first processing *in parallel* all edges of the input graph and then combining the results. This bulk semantics relies on introduction of ε edges 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. 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 [BFS00], how to structure transformations on ordered graphs. In particular, it is unclear how to define a bulk semantics for transformations on ordered graphs.

In this paper, we show that the bulk semantics can be extended to ordered graphs, and that a clear bidirectional semantics of UnCAL$^{\circ}$, an ordered version of UnCAL, can be defined. The main technical contributions are three folds. First, we design a new graph transformation language UnCAL$^{\circ}$ (Section 2), which is similar to the graph algebra UnCAL [BFS00] for unordered graphs. Next, we give a novel definition of bisimulation relation on ordered graphs with ε edges, propose the bulk semantics of structural recursion, and prove bisimulation genericity (well-definedness on bisimilarity) of the constructors and the structural recursion (Section 3). Finally, based on the above results, we show that the bidirectionalization method in [HHI+10] can be adapted to define bidirectional semantics of UnCAL$^{\circ}$ (Section 4).

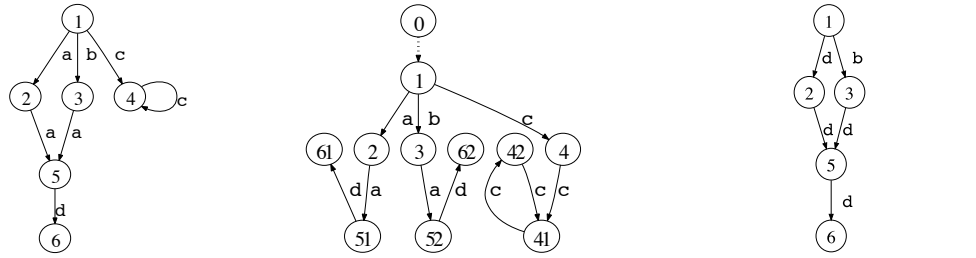
2 UnCAL$^{\circ}$: A Transformation Language for Ordered Graphs

UnCAL$^{\circ}$ is a graph transformation language, which is similar to UnCAL [BFS00], but can distinguish orders of outgoing edges. In UnCAL$^{\circ}$, we will use $[]$ and $+\!\!+$ for ordering instead of $\{\}$ and $\cup$ used in UnCAL. The purpose of this section is to give an intuition of UnCAL$^{\circ}$. The formal definition of the semantics of UnCAL$^{\circ}$ is shown in Section 3.

2.1 Graph Model

In UnCAL$^{\circ}$, graphs are rooted, directed, and edge-labeled graphs with order on outgoing edges. This graph data model has two prominent features, *markers* and ε*-edges*. Nodes may be marked with *input* and *output markers*, which are used as an interface to connect them to other graphs. An ε-edge represents a shortcut of two nodes, working like the ε-transition in an automaton. We use $\mathscr{L}$ to denote the set of labels, $\mathscr{L}_\varepsilon$ to denote $\mathscr{L} \cup \{\varepsilon\}$, X to denote the set of input markers, and Y to denote the set of output markers.

An ordered graph G is represented as a triple (V, B, I), where V is a set of nodes, $B : V \to \mathit{List}(\mathscr{L}_\varepsilon \times V + Y)$ is a function that maps a node to a list of elements, which is either a pair of a label and a node, or an output marker, and $I : X \to V$ is a function that maps an input marker to a



(a) A Simple Graph (b) An Equivalent Graph (c) Result of $a2d\_xc$ on (a)

Figure 1: Graph Equivalence Based on Bisimulation

node, which is called *input node*. Unlike input markers, more than one node can be marked with an identical output marker. They are called *output nodes*. Intuitively, input nodes are root nodes of the graph (we allow a graph to have multiple root nodes, and for singly rooted graphs, we often use default marker `&` to indicate the root), while an output node can be seen as a "context-hole" of graphs where an input node with the same marker will be plugged in later.

Note that multiple-marker graphs are meant to be an internal data structure for graph composition. In fact, initial source graphs of our transformation have one input marker (single-rooted) and no output markers (no holes). For instance, the graph in Fig. 1(a) is denoted by (V,B,I) where $V = \{1,2,3,4,5,6\}$, $B(1) = [(\mathtt{a},2),(\mathtt{b},3),(\mathtt{c},4)]$, $B(2) = [(\mathtt{a},5)]$, $B(3) = [(\mathtt{a},5)]$, $B(4) = [(\mathtt{c},4)]$, $B(5) = [(\mathtt{d},6)]$, $B(6) = []$, and $I(\&) = 1$.

Notion of Graph Equivalence While a formal definition of graph equivalence will be given in Section 3, two graphs are value equivalent if they are bisimilar. For instance, the graph in Fig. 1(b) is value equivalent to the graph in Fig. 1(a); the new graph has an additional ε-edge (denoted by the dotted line), duplicates the graph rooted at node 5, and unfolds and splits the cycle at node 4. Unreachable parts are also disregarded, i.e., two bisimilar graphs are still bisimilar if one adds subgraphs unreachable from input nodes.

2.2 Graph Constructors

Figure 2 summarizes the nine graph constructors. To treat order of outgoing edges, we use $[]$ and $+\!\!+$ instead of $\{\}$ and $\cup$ used in UnCAL. Here, $[]$ constructs a root-only graph, $[a : G]$ constructs a graph by adding an edge with label $a \in \mathscr{L}_\varepsilon$ pointing to the root of graph G, and $G_1 +\!\!+ G_2$ adds two ε-edges from a new root to the roots of G_1 and G_2, respectively. Also, $\&x := G$ associates an input marker, $\&x$, to the root node of G, $\&y$ constructs a graph with a single node marked with an output marker $\&y$, and $()$ constructs an empty graph that has neither a node nor an edge. Furthermore, $G_1 \oplus G_2$ constructs a graph by using a componentwise (V, B and I) union. $+\!\!+$ differs from $\oplus$ in that $+\!\!+$ unifies input nodes while $\oplus$ does not. $\oplus$ requires input markers of operands to be disjoint, while $+\!\!+$ requires them to be identical. $G_1 \mathbin{@} G_2$ composes two graphs vertically by connecting the output nodes of G_1 with the corresponding input nodes of G_2 with ε-edges, and $\mathbf{cycle}(G)$ connects the output nodes with the input nodes of G to form cycles. The formal

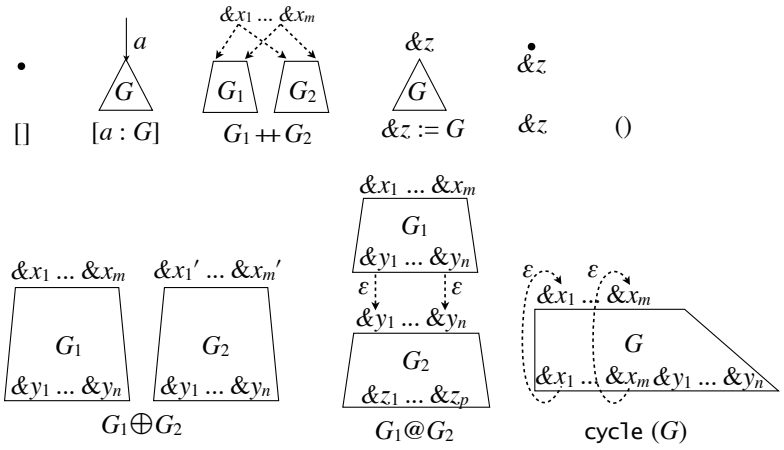


Figure 2: Graph Constructors

$$
\begin{array}{rcll}
e & ::= & [] \mid [l : e] \mid e +\!\!+ e \mid \&x := e \mid \&y \mid () & \\
& \mid & e \oplus e \mid e \mathbin{@} e \mid \mathbf{cycle}(e) & \{ \text{ constructor } \} \\
& \mid & \$g & \{ \text{ graph variable } \} \\
& \mid & \mathbf{let}\,\$g = e\,\mathbf{in}\,e & \{ \text{ variable binding } \} \\
& \mid & \mathbf{if}\,l = l\,\mathbf{then}\,e\,\mathbf{else}\,e & \{ \text{ conditional } \} \\
& \mid & \mathbf{rec}(\lambda(\$l, \$g).e)(e) & \{ \text{ structural recursion application } \} \\
l & ::= & a \mid \$l & \{ \text{ label } (a \in \mathscr{L}_\varepsilon) \text{ and label variable } \}
\end{array}
$$

Figure 3: UnCAL^{o} Language

definition of the semantics of these constructors can be found in Section 3. It is worth noting that this set of constructors are powerful enough to describe any ordered graph.

2.3 UnCAL^{o} Syntax

The syntax of UnCAL^{o} is depicted in Fig. 3. As will be seen in Section 3, UnCAL^{o} is bisimulation generic, i.e., for bisimilar inputs, each UnCAL^{o} expression results in bisimilar result. It consists of the graph constructors, variables[1], variable bindings, conditionals, and structural recursion. We have already described the graph constructors, while variables, variable bindings and conditionals are self explanatory. Now we introduce *structural recursion*, which is a powerful mechanism in UnCAL^{o} to describe graph transformations. A structural recursion is a function f, which satisfies the following equations.

$$
\begin{array}{lcl}
f([]) & = & [] \\
f([\$l : \$g]) & = & e \mathbin{@} f(\$g) \\
f(\$g_1 +\!\!+ \$g_2) & = & f(\$g_1) +\!\!+ f(\$g_2),
\end{array}
$$

[1] We prefix \$ to represent variables.

and f can be encoded by $\mathbf{rec}(\lambda(\$l,\$g).e)$. For $[]$, f results in $[]$. For a graph with an edge labelled $\$l$ connected to a graph $\$g$, f results in connecting vertically the result of e with the result of recurring f to $\$g$. For a graph with $\$g_1$ preceding $\$g_2$, f results in concatenating two graphs of $f(\$g_1)$ and $f(\$g_2)$.

Example 1 *The following structural recursion a2d_xc replaces all labels* `a` *with* `d` *and contract edges labeled* `c`.

$$\begin{aligned} a2d\_xc(\$db) = \mathbf{rec}(\lambda(\$l,\$g).\ &\mathbf{if}\ \$l = \texttt{a}\ \mathbf{then} && [\texttt{d} : \&] \\ &\mathbf{else\ if}\ \$l = \texttt{c}\ \mathbf{then} && [\&] \\ &\mathbf{else} && [\$l : \&])\ (\$db) \end{aligned}$$

Applying the function a2d_xc to the graph in Fig. 1(a) yields the graph in Fig. 1(c). □

Example 2 *Consider an ordered graph representation of books. Since "sections" are ordered and there are some references in books, we can see books as ordered graphs. The following structural recursion toc, which is adapted from [RSGV09], computes the table of contents of books in which sections can be arbitrarily nested:*

$$\begin{aligned} toc(\$db) = \mathbf{rec}(\lambda(\$l,\$g).\ &\mathbf{if}\ \$l = \texttt{section}\ \mathbf{then}\ [\texttt{section} : [gettitle(\$g), \&]] \\ &\mathbf{else}\ \ \&)\ (\$db) \end{aligned}$$

where the function gettitle results in the title of the section.

$$\begin{aligned} gettitle(\$g) = \mathbf{rec}(\lambda(\$l_1,\$g_1).\ &\mathbf{if}\ \$l_1 = \texttt{title}\ \mathbf{then}\ [\texttt{title} : \mathbf{rec}(\lambda(\$l_2,\$g_1).[\$l_2 : []])\ (\$g_1)] \\ &\mathbf{else}\ \ [])\ (\$g) \end{aligned}$$

□

3 Ordered Graph and Structural Recursion

In the previous section, we gave syntax of UnCAL$^{\mathrm{o}}$, its intuitive meaning, and its usage. In this section, we give semantics of UnCAL$^{\mathrm{o}}$. In the next section, we extend the semantics here for bidirectional transformation, where ε-edge and bulk semantics for structural recursion—main contents in this section—are importantly used to keep source information during forward transformation. Thus we heavily use ε-edge, so the bisimilarity for ordered graph with ε-edges is important.

Here we first give definitions of ordered graph and its bisimilarity. These ordered graphs up to bisimilarity form the domain of the semantics of UnCAL$^{\mathrm{o}}$. Next we define graph constructors and structural recursion with bulk semantics. Then we show *bisimulation genericity* (well-definedness on bisimilarity) of the graph constructors and the structural recursion (Theorem 1).

The reader who believe our proofs and naturality of definitions can skip this section except for Section 3.1, Definitions 6 and 7.

3.1 Formal Definition of Ordered Graph

Definition 1 (Ordered Graph) Let $\mathscr{L}$ be a set of *labels*, and $\mathscr{L}_\varepsilon$ be the disjoint union $\mathscr{L} \cup \{\varepsilon\}$. Let X be a finite set of *input markers* and Y be a finite set of *output markers*.

An *ordered graph*, or just a *graph* G is a triple (V, B, I) where

- V is a set of *nodes*,
- $B : V \to \coprod_{n\in\mathbb{N}} (\mathscr{L}_\varepsilon \times V + Y)^n$ is a function, which maps a node to its *ordered branches* consisting of *labeled edges* or *output markers*, and
- $I : X \to V$ is a function, which determines *roots (input nodes)* of the graph.

The set of all graphs (defined with X and Y) is denoted by DB_Y^X (meaning DataBase following [BFS00]). For a graph G, we refer to each component of the triple by record notation with labels V, B, I: i.e., $G = (G.\mathrm{V}, G.\mathrm{B}, G.\mathrm{I})$. □

Remark 1 *In the paper [BFS00], unordered graph was defined additionally with the following three requirements: an input function I is injective; there is no incoming edge to an input node; and there is no outgoing edge from an output node. While, the above definition of ordered graph is relaxed on such points. The both styles are in fact equivalent: i.e., for the above style of a graph, if we extend each input node with a fresh epsilon edge incoming from a fresh new input node, and also replace each output marker occurrence on an output node by a fresh epsilon edge outgoing to a fresh node having only the branch of the output marker, then the resulting graph satisfies the above three requirements, and still is bisimilar (Definition 5) to the given graph. This transformation is possible for both unordered case and ordered case.*

As meta-variables for markers, we use $\&x$, $\&y$, $\&z$, etc. Since we often consider single-rooted graphs, i.e., graphs whose sets of input markers are singleton, we choose a *default marker* denoted by $\&$, and $DB_Y^{\{\&\}}$ is denoted by DB_Y.

In the above definition, $\coprod_{n\in\mathbb{N}} (\mathscr{L}_\varepsilon \times V + Y)^n = List(\mathscr{L}_\varepsilon \times V + Y)$. To represent ordered branch, "list in polynomial style" is more essential than "list as initial algebra", since with polynomial style, the more generalized notion of *graph with countably wide ordered branch* can be similarly defined with $B : V \to \coprod_{L\in\mathbb{L}} (\mathscr{L}_\varepsilon \times V + Y)^L$ where $\mathbb{L}$ is the set of countable linear ordered sets (up to order isomorphism).

For a list x, $|x|$ denotes the length of x, and we regard $|x|$ also as the set of component-indices of x, i.e., $|x| = \{0, ..., |x|-1\}$. Then for $i \in |x|$, $x.i$ denotes the i-th component of x. We often represent a list as $[x_i]_{i\in n}$, as well as an extensional style such as $[x, y, z]$. For a tuple $x \in \prod_{i\in I} X_i$ and an index $i \in I$, we use $x.i$ to denote the i-th component of x: e.g., for $x \in A \times B$, $x = (x.0, x.1)$. For sets A and B, an element of the coproduct $A + B$ is either $\mathrm{in}_l(a)$ with $a \in A$ exclusively or $\mathrm{in}_r(b)$ with $b \in B$. For ternary coproduct, we use in_l, in_m, and in_r.

For a graph $G = (V, B, I)$, a node $v \in V$, and a branch index $i \in |B(v)|$, the list component $B(v).i$ is either an l-labeled edge $\mathsf{E}(l,\ v') \stackrel{\text{def}}{=} \mathrm{in}_l(l, v')$ from v to v', or an output marker $\mathsf{O}(\&y) \stackrel{\text{def}}{=} \mathrm{in}_r(\&y)$. Do not confuse the notion of branch with the notion of edge; a branch might be an output marker as well as an edge. An output marker can be considered as a "future (collection of) branches"; an output marker is a variable (or place holder) to which all the branches of a root of another graph can be "substituted" through an ε-edge by @ operator or **cycle** operator (see Figure 2 and Definition 6).

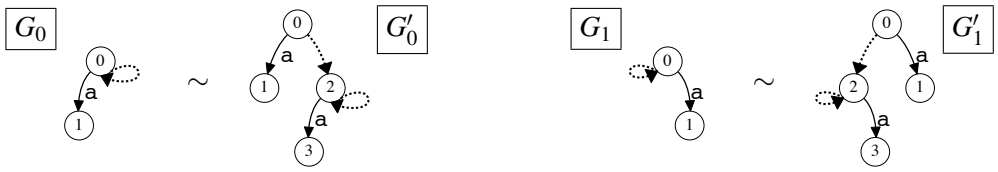


Figure 4: graphs with left-to-right-stream branching and with right-to-left-stream branching

For a graph G, we call G *finite* if $G.\mathrm{V}$ is finite. In this section, we also consider infinite graphs because we need infinite graph (in fact infinite tree) to define bisimilarity even for finite graph. In other sections, we treat only finite graphs and call them just (ordered) graphs.

For a graph $G \in DB_Y^X$, and a node $v \in G.\mathrm{V}$, $G|_v$ denotes the graph in DB_Y whose set of nodes are restricted to the set of all nodes "accessible through consecutive branches" from v.

For a graph $G \in DB_Y^X$, G is a *tree* if (i): $G.\mathrm{I}$ is injective, (ii): for every root node, there is no incoming edge, (iii): for every non-root nodes, there is exactly one incoming edge, and (iv) any node can be accessible from a root node in finite steps. (For finite graphs, the condition (iv) is derived from the conditions (ii) and (iii).) The conditions (ii) and (iii) keep out cycle and sharing from tree. Note that if t is a tree, so is $t|_v$ for any $v \in t.\mathrm{V}$, and also that any multi-rooted tree $t \in DB_Y^X$ is the disjoint union of the family $(t|_{t.\mathrm{I}(\&x)}))_{\&x \in X}$ of the single-rooted trees.

3.2 Bisimilarity

Bisimilarity between ordered graph is—if the notion does not involve ε-edge—easily defined with a general coalgebraic definition (e.g. [Rut00, SR11]) applied to the endofunctor $V \mapsto List(\mathscr{L} \times V + Y)$. However, for ordered graph with ε-edges, the definition of bisimilarity is not obvious. In the unordered case, we can take the approach in the paper [BJ10], but for ordered case, it is not available.

First let us see the bisimilarity for the case without ε-edge.

Definition 2 For ordered graphs $G, G' \in DB_Y^X$ without ε-edges, G and G' are *bisimilar* (and denoted by $G \sim_{\mathscr{L}} G'$) if there is a relation R between $G.\mathrm{V}$ and $G'.\mathrm{V}$ such that (i): for every input marker $\&x \in X$, $(G.\mathrm{I}(\&x))R(G'.\mathrm{I}(\&x))$, and (ii): R is a *bisimulation relation*: i.e., for any vRv', $|G.\mathrm{B}(v)| = |G'.\mathrm{B}(v')|$, and then for any $i \in |G.\mathrm{B}(v)|$, either if $(G.\mathrm{B}(v)).i = \mathsf{E}(l,\ v_1)$, then there is $v'_1 \in G'.\mathrm{V}$ such that $(G'.\mathrm{B}(v')).i = \mathsf{E}(l,\ v'_1)$ and $v_1 R v'_1$, or if $(G.\mathrm{B}(v)).i = \mathsf{O}(\&y)$, then $(G'.\mathrm{B}(v)).i = \mathsf{O}(\&y)$. □

Now let us see some ordered graphs with ε-edges, to have some feeling of bisimilarity. Consider graphs in Figure 4. The graphs G_i should be intuitively bisimilar respectively to their 1-step-unfolding G'_i, because unfolding generally maintains bisimilarity. Here, we should distinguish G_0 from G_1. This is because, observing their further unfolding, it is found that G_0 is a tree with height one and with countable width of branch, and the order of the branching is like an infinite stream from left to right; on the other hand, the graph G_1 is similar one but the branch is like an infinite stream from right to left.

In the case without ε-edge, we just compare the list of branches under the condition "$|G.\mathrm{B}(v)| =$

$|G'.\mathrm{B}(v)|$" above; on this point for the case with ε-edge, we first extend the notion of list of branches to that of linear ordered set of branches, and then compare them in terms of order isomorphism. For the examples, G_0 has no maximum (right side) but has minimum (left side), while G_1 has maximum and has no minimum, so they are not isomorphic, and hence not bisimilar. (Order isomorphism was implicit in the case without ε-edge, since for finite linear ordered sets, to be order isomorphic is determined just by their cardinalities.) In order to define sets of all such branches and to define linear orders on them, we first unfold graphs to trees.

Thus we define bisimilarity for ordered graph with ε-edges in two steps: we first unfold graphs to trees, which might be infinite even for finite graphs; and then determine "proper" branches by taking transitive closure of ε-edges, and "bi-simulate" them with order isomorphisms.

The definition of unfolding of graph to tree is usual one, which keeps ε-edges, regarding ε as a normal label in $\mathscr{L}$.

Definition 3 (Unfolding) Let $G = (V,B,I) \in DB_Y^X$. We define a tree $\mathrm{uf}(G) \in DB_Y^X$ as in the following.

We will define only for the case that X is a singleton, say $\{\&\}$, then for the other case, we can define $\mathrm{uf}(G)$ as the disjoint union tree of family $(\mathrm{uf}(G|_{(G.\mathrm{I})(\&x)}))_{\&x \in X}$ of single-rooted graphs.

First we define a family of disjoint sets $(V_n)_{n\in\mathbb{N}}$ by induction on $n \in \mathbb{N}$, as V_n will be the set of n-depth nodes $v \in \mathrm{uf}(G).\mathrm{V}$, and there $v = (\text{the depth n of } v,\ \text{the parent node of } v,\ \text{the corresponding node in the original graph } G,\ \text{the branch index of } v)$.

- $V_0 \stackrel{\text{def}}{=} \{\ (0,\bot,I(\&),\bot)\ \}$, where the element will be the root, hence $\bot$ in the two undefinable components is just some dummy.
- $V_{n+1} \stackrel{\text{def}}{=} \{\ (n{+}1,x,v,i)\ \mid\ x \in V_n,\ i \in |B(x.2)|,\ B(x.2).i = \mathrm{in}_l(l,v)\ \}$

Then $\mathrm{uf}(G) \stackrel{\text{def}}{=} (\bigcup_{n\in\mathbb{N}} V_n, B', \{\& \mapsto (0,\bot,I(\&),\bot)\}) \in DB_Y$ where for $x \in V_n$,

$$B'(x) \stackrel{\text{def}}{=} \left[\begin{array}{rcl} B(x.2).i = \mathsf{E}(l,\ v) & \Rightarrow & \mathsf{E}(l,\ (n{+}1,x,v,i)) \\ = \mathsf{O}(\&y) & \Rightarrow & \mathsf{O}(\&y) \end{array}\right]_{i\in|B(x.2)|} \qquad \square$$

For graphs G and G' (possibly with ε-edges), we can define their *bisimilarity observing* ε (denoted by $G \sim_! G'$) just as $G \sim_{\mathscr{L}_\varepsilon} G'$, where we regard ε as a normal label. Then it is easily shown that $G \sim_! G'$ if and only if $\mathrm{uf}(G)$ and $\mathrm{uf}(G')$ are (uniquely) isomorphic. We use this notion of bisimilarity observing ε in the proof of bisimulation genericity (Theorem 1).

Now let us go to the second step; first we consider transitive closure of ε-edges.

For a tree $T \in DB_Y^X$ and a node $v \in T.\mathrm{V}$, let v_0 be a *self-or-ancestor* of v, i.e., there is a path $v_0 \stackrel{l_0}{\to}_{i_0} \ldots v_{n-1} \stackrel{l_{n-1}}{\to}_{i_{n-1}} v_n (\stackrel{\text{def}}{=} v) (n \in \mathbb{N})$ where $v_j \stackrel{l_j}{\to}_{i_j} v_{j+1}$ means $G.\mathrm{B}(v_j).i_j = \mathsf{E}(l_j,\ v_{j+1})$. Note that by the condition of tree, such path is unique. Then for a branch index $i \in |T.\mathrm{B}(v)|$ of v, the pair (v,i) is *proper branch of* v_0 if (i): all l_j are ε, and (ii): if $T.\mathrm{B}(v).i = \mathsf{E}(l,\ v')$ for some l and v', then $l \neq \varepsilon$. (Note that for the case that $T.\mathrm{B}(v).i = \mathsf{O}(\&y)$, the condition (ii) holds.) For a tree $T \in DB_Y^X$ and a node $v_0 \in T.\mathrm{V}$, $\mathrm{Pb}(T,v_0)$ denotes the set of all proper branches of v_0.

Then there is a natural linear order $\leq_{\mathrm{Pb}}$ on $\mathrm{Pb}(T,v_0)$: for two different proper branches (v,i) and (v',i'), we compare the two branches of the lowest one—in terms of depth of tree—among common self-or-ancestors of v and v'. Formally, let $v_0 \stackrel{\varepsilon}{\to}_{i_0} \ldots v_{n-1} \stackrel{\varepsilon}{\to}_{i_{n-1}} v_n (\stackrel{\text{def}}{=} v)$ and

$v'_0 (\stackrel{\text{def}}{=} v_0) \xrightarrow{\varepsilon}_{i'_0} \ldots v'_{n'-1} \xrightarrow{\varepsilon}_{i'_{n'-1}} v'_{n'} (\stackrel{\text{def}}{=} v')$ be respectively the paths from v_0 to v and v'; and also let $i_n \stackrel{\text{def}}{=} i$ and $i'_{n'} \stackrel{\text{def}}{=} i'$. Then by conditions in the definition of proper branch, $(v_{\min(n,n')}, i_{\min(n,n')}) \neq (v'_{\min(n,n')}, i'_{\min(n,n')})$, hence we can take the least k such that $(v_k, i_k) \neq (v'_k, i'_k)$ as k_0. Then we can show that $v_{k_0} = v'_{k_0}$, hence $i_{k_0} \neq i'_{k_0}$. Now we define as $(v,i) <_{\text{Pb}} (v',i') \stackrel{\text{def}}{\Longleftrightarrow} i_{k_0} < i'_{k_0}$.

Now let us go to the second step for defining bisimilarity of ordered graph.

Definition 4 (Bisimilarity for Tree) For a pair of trees $T, T' \in DB_Y^X$, T and T' are *bisimilar* if there is a relation R between $T.\mathrm{V}$ and $T'.\mathrm{V}$ such that

- for every input marker $\&x \in X$, $(T.\mathrm{I}(\&x)) R (T'.\mathrm{I}(\&x))$, and
- R is a *bisimulation relation*: i.e., for any $v_0 R v'_0$, there is an order isomorphism $f : (\mathrm{Pb}(T, v_0), \leq_{\text{Pb}}) \to (\mathrm{Pb}(T', v'_0), \leq_{\text{Pb}})$ satisfying the following property. For any proper branch $(v,i) \in \mathrm{Pb}(T, v_0)$, let (v', i') be $f(v,i)$. Then,
  - for any $l \in \mathscr{L}, u \in T.\mathrm{V}$, if $T.\mathrm{B}(v).i = \mathsf{E}(l,\ u)$, then there exist u' such that $T'.\mathrm{B}(v').i' = \mathsf{E}(l,\ u')$ and uRu', and
  - for any $\&y \in Y$, if $T.\mathrm{B}(v).i = \mathsf{O}(\&y)$, then $T'.\mathrm{B}(v').i' = \mathsf{O}(\&y)$.

□

It can be checked that the above bisimilarity relation is symmetric and an equivalence relation.

Definition 5 (Bisimilarity for Ordered Graph with ε-edges) For graphs $G, G' \in DB_Y^X$, G and G' are *bisimilar* (and denoted by $G \sim G'$) if $\mathrm{uf}(G)$ and $\mathrm{uf}(G')$ are bisimilar (in the sense of Definition 4). (Bisimilarity is called *value equivalence* in [BFS00].) □

For a tree $T \in DB_Y^X$, T and $\mathrm{uf}(T)$ are equal up to the unique isomorphism. Hence for any pair of trees, the two bisimilarity above are equivalent, and not ambiguous.

For a pair of graphs $G, G' \in DB_Y^X$, if G and G' has no ε-edge, then $G \sim G' \Longleftrightarrow G \sim_! G' \Longleftrightarrow G \sim_{\mathscr{L}} G'$. Otherwise, if $G \sim_! G'$ then $G \sim G'$, but the converse is not necessarily true.

3.3 Graph Constructors and Bulk Semantics for Structural Recursion

Now we define graph constructors and a structural recursion; we use the same function name as the name of a function symbol in the syntax.

Definition 6 (Graph Constructors)

- $[] \stackrel{\text{def}}{=} (\{\&\}, \{\& \mapsto []\}, \mathrm{id}_{\{\&\}}) \in DB_Y$
- For $G \in DB_Y$, $[l : G] \stackrel{\text{def}}{=} (G.\mathrm{V} \cup \{v_0 : \mathrm{fresh}\}, G.\mathrm{B} \cup \{v_0 \mapsto [\mathsf{E}(l,\ G.\mathrm{I}(\&))]\}, \{\& \mapsto v_0\}) \in DB_Y$
- For $G_l \in DB_Y^X$ and $G_r \in DB_Y^X$, $G_l +\!\!+ G_r \stackrel{\text{def}}{=} (G_l.\mathrm{V} + X + G_r.\mathrm{V}, B', \mathrm{in}_m) \in DB_Y^X$ where

$$B'(\mathrm{in}_k(v)) \underset{(k=l \text{ or } r)}{\stackrel{\text{def}}{=}} \begin{bmatrix} G_k.\mathrm{B}(v).i = \mathsf{E}(l,\ v') & \Rightarrow & \mathsf{E}(l,\ \mathrm{in}_k(v')) \\ = \mathsf{O}(\&y) & \Rightarrow & \mathsf{O}(\&y) \end{bmatrix}_{i \in |g_k.\mathrm{B}(v)|}$$

$$B'(\mathrm{in}_m(\&x)) \stackrel{\text{def}}{=} [\mathsf{E}(\varepsilon,\ \mathrm{in}_l(G_l.\mathrm{I}(\&x))),\ \mathsf{E}(\varepsilon,\ \mathrm{in}_r(G_r.\mathrm{I}(\&x)))]$$

- For $G \in DB_Y$, $(\&x := G) \stackrel{\text{def}}{=} (G.\mathrm{V}, G.\mathrm{B}, \{\&x \mapsto G.\mathrm{I}(\&)\}) \in DB_Y^{\{\&x\}}$
- For $\&y \in Y$, $\&y \stackrel{\text{def}}{=} (\{\&\}, \{\& \mapsto [\mathsf{O}(\&y)]\}, \mathrm{id}_{\{\&\}}) \in DB_Y$
- $() \stackrel{\text{def}}{=} (\emptyset, \emptyset, \mathrm{id}_\emptyset) \in DB_Y^\emptyset$
- For $G_l \in DB_Y^{X_l}$ and $G_r \in DB_Y^{X_r}$ such that $X_l \cap X_r = \emptyset$, $G_l \oplus G_r \stackrel{\text{def}}{=} (G_l.\mathrm{V} + G_r.\mathrm{V}, B', I') \in DB_Y^{X_l \cup X_r}$ where

  $$B'(\mathrm{in}_k(v)) \underset{(k=l \text{ or } r)}{\stackrel{\text{def}}{=}} \left[\begin{array}{rcl} G_k.\mathrm{B}(v).i = \mathsf{E}(l,\ v') & \Rightarrow & \mathsf{E}(l,\ \mathrm{in}_k(v')) \\ = \mathsf{O}(\&y) & \Rightarrow & \mathsf{O}(\&y) \end{array} \right]_{i \in |G_k.\mathrm{B}(v)|}$$

  $$I'(\&x) \stackrel{\text{def}}{=} \mathrm{in}_l(G_l.\mathrm{I}(\&x)) \text{ (if } \&x \in X_l) \text{ or } \mathrm{in}_r(G_r.\mathrm{I}(\&x)) \text{ (if } \&x \in X_r)$$
- For $G_l \in DB_Y^X$ and $G_r \in DB_Z^Y$, $G_l \mathbin{@} G_r \stackrel{\text{def}}{=} (G_l.\mathrm{V} + G_r.\mathrm{V}, B', \mathrm{in}_l \circ G_l.\mathrm{I}) \in DB_Z^X$ where

  $$B'(\mathrm{in}_k(v)) \underset{(k=l \text{ or } r)}{\stackrel{\text{def}}{=}} \left[\begin{array}{rcl} G_k.\mathrm{B}(v).i = \mathsf{E}(l,\ v') & \Rightarrow & \mathsf{E}(l,\ \mathrm{in}_k(v')) \\ = \mathsf{O}(\&w) & \Rightarrow & \left\{ \begin{array}{rcl} k = l & \Rightarrow & \mathsf{E}(\varepsilon,\ \mathrm{in}_r(G_r.\mathrm{I}(\&w))) \\ = r & \Rightarrow & \mathsf{O}(\&w) \end{array} \right. \end{array} \right]_{i \in |g_k.\mathrm{B}(v)|}$$
- For $G \in DB_{X \cup Y}^X$ such that $X \cap Y = \emptyset$, $\mathbf{cycle}(G) \stackrel{\text{def}}{=} (G.\mathrm{V}, B', G.I) \in DB_Y^X$ where

  $$B'(v) \stackrel{\text{def}}{=} \left[\begin{array}{rcl} B(v).i = \mathsf{E}(l,\ v') \text{ or } \mathsf{O}(\&y)\,(\&y \in Y) & \Rightarrow & B(v).i \\ = \mathsf{O}(\&x)\,(\&x \in X) & \Rightarrow & \mathsf{E}(\varepsilon,\ G.\mathrm{I}(\&x)) \end{array} \right]_{i \in |B(v)|}$$

□

Remark 2 *The definition of* **cycle** *above is different from that in the paper [BFS00]: i.e., we just added cycling epsilon edges, while in loc. cit. additionally each input node is extended with a fresh epsilon edge incoming from a fresh new input node. This is just (a part of) the transformation in Remark 1 and hence the both are equivalent.*

Definition 7 (Bulk Semantics of Structural Recursion) For $e : \mathscr{L} \times DB_Y \to DB_Z^Z$, a *structural recursion function (for ordered graph)* $\mathbf{rec}(e) : DB_Y^X \to DB_{Z \times Y}^{Z \times X}$ is defined as the following.

Let $G = (V, B, I) \in DB_Y^X$. We extend e to $\bar{e} : \mathscr{L}_\varepsilon \times DB_Y \to DB_Z^Z$ which maps $(\varepsilon, \_)$ to the "identity" graph $(Z, \{\&z \mapsto [\mathsf{O}(\&z)]\}, \mathrm{id}_Z)$. For a list $x \in List(\mathscr{L}_\varepsilon \times V + Y)$, let $x|_{\text{edge}} \stackrel{\text{def}}{=} \{(i, l, v) \in \mathbb{N} \times \mathscr{L}_\varepsilon \times V \mid i \in |x|,\ x.i = \mathsf{E}(l,\ v)\}$.

Then $\mathbf{rec}(e)(G) \stackrel{\text{def}}{=} (V', B', I')$ where

- $V' \stackrel{\text{def}}{=} (Z \times V) + (\coprod_{v \in V, (i,l,v') \in B(v)|_{\text{edge}}} \bar{e}(l, G|_{v'}).\mathrm{V})$

  (We simplify the indices in $\coprod_{v \in V, (i,l,v') \in B(v)|_{\text{edge}}}$ just as $\coprod_{v,i,l,v'}$, in the following.)
- $B' : (Z \times V) + (\coprod_{v,i,l,v'} \bar{e}(l, G|_{v'}).\mathrm{V}) \to \coprod_n \left(\mathscr{L}_\varepsilon \times \left((Z \times V) + (\coprod_{v,i,l,v'} \bar{e}(l, G|_{v'}).\mathrm{V}) \right) + Z \times Y \right)^n$

  $$\mathrm{in}_l(\&z, v) \ \mapsto \ \left[\begin{array}{rcl} B(v).i = \mathsf{E}(l,\ v') & \Rightarrow & \mathsf{E}\left(\varepsilon,\ \mathrm{in}_r\left((v,i,l,v'), \bar{e}(l, G|_{v'}).I(\&z)\right)\right) \\ = \mathsf{O}(\&y) & \Rightarrow & \mathsf{O}((\&z, \&y)) \end{array} \right]_{i \in |B(v)|}$$

  $$\mathrm{in}_r((v,i,l,v'), u) \underset{(x \stackrel{\text{def}}{=} (\bar{e}(l, G|_{v'}).\mathrm{B})(u))}{\mapsto} \left[\begin{array}{rcl} x.i = \mathsf{E}(l',\ u') & \Rightarrow & \mathsf{E}\left(l',\ \mathrm{in}_r\left((v,i,l,v'), u'\right)\right) \\ = \mathsf{O}(\&z) & \Rightarrow & \mathsf{E}\left(\varepsilon,\ \mathrm{in}_l(\&z, v')\right) \end{array} \right]_{i \in |x|}$$

- $I' : Z \times X \to (Z \times V) + (\coprod_{v,i,l,v'} \bar{e}(l, G|_{v'}).\mathrm{V}) : (\&z, \&x) \mapsto \mathrm{in}_l(\&z, I(\&x))$

□

It should be noticed that the above graph constructors are closed on finiteness, i.e., they map finite graphs to a finite graph; and also if e is closed on finiteness, so is $\mathbf{rec}(e)$.

From now we show that the above functions on graphs are bisimulation generic.

Definition 8 (Bisimulation Genericities) Let $f : DB_{Y_1}^{X_1} \times ... \times DB_{Y_n}^{X_n} \to DB_Y^X$. Then f is *bisimulation generic* if for $G_1 \sim G_1', ..., G_n \sim G_n'$, $f(G_1, ..., G_n) \sim f(G_1', ..., G_n')$; and f is *strongly bisimulation generic* if both f is *bisimulation generic observing* ε, i.e., for $G_1 \sim_! G_1', ..., G_n \sim_! G_n'$, $f(G_1, ..., G_n) \sim_! f(G_1', ..., G_n')$, and f is bisimulation generic *for trees*, i.e., for trees $T_1 \sim T_1', ..., T_n \sim T_n'$, $f(T_1, ..., T_n) \sim f(T_1', ..., T_n')$. □

Since our bisimilarity was defined in two steps, proving bisimulation genericity is complicated, so we split the bisimulation genericity into the two simpler bisimulation genericities:

Lemma 1 *Let $f : DB_{Y_1}^{X_1} \times ... \times DB_{Y_n}^{X_n} \to DB_Y^X$. If f is strongly bisimulation generic, then f is bisimulation generic.* □

Proof. Let $f : DB_Y^X \to DB_{Y'}^{X'}$ be a unary function to keep presentation simple, and assume the both assumption in the statement and $G \sim G'$. Our goal is to prove that $f(G) \sim f(G')$.

By the definition of the bisimilarity, $G \sim G'$ implies $\mathrm{uf}(G) \sim \mathrm{uf}(G')$. Now f is bisimulation generic for trees $\mathrm{uf}(G)$ and $\mathrm{uf}(G')$, so $f(\mathrm{uf}(G)) \sim f(\mathrm{uf}(G'))$. On the other hand, since f is bisimulation generic observing ε, and since $G \sim_! \mathrm{uf}(G)$ and $G' \sim_! \mathrm{uf}(G')$, it holds that $f(G) \sim_! f(\mathrm{uf}(G))$ and $f(G') \sim_! f(\mathrm{uf}(G'))$. Since $\sim_!$ implies $\sim$, hence $f(G) \sim f(\mathrm{uf}(G))$ and $f(G') \sim f(\mathrm{uf}(G'))$. Thus it is derived that $f(G) \sim f(G')$. □

We remark that the converse of this lemma is trivially false. Because, take such graphs G_0, G_1, and G_0' that $G_0 \sim G_1$, not $G_0 \sim_! G_1$, $G_0 \sim_! G_0'$, and $G_0 \neq G_0'$. Then we define $f(G) \stackrel{\mathrm{def}}{=}$ if $G = G_0$ then G_1 else G. Then, since $f(G) \sim G$, f is bisimulation generic; but f is not bisimulation generic observing ε, since $G_0 \sim_! G_0'$ and not $f(G_0) \sim_! f(G_0')$.

Theorem 1 *The graph constructors and the structural recursion are strongly bisimulation generic. The latter means that for $e : \mathscr{L} \times DB_Y \to DB_Z^Z$, if for any $l \in \mathscr{L}$, $e(l, \text{-})$ is strongly bisimulation generic, then so is $\mathbf{rec}(e) : DB_Y^X \to DB_{Z \times Y}^{Z \times X}$.* □

Proof. We omit similar parts to the proof of the similar theorem in the paper [BFS00], and describe other points proper to this ordered case. On the case of graph constructors, the proofs are straightforward. For the structural recursion, first we can directly show that if e is bisimulation generic observing ε, then so is $\mathbf{rec}(e)$. We can also directly show that for any graph g and $e, e' : \mathscr{L} \times DB_Y \to DB_Z^Z$, if $e \sim_! e'$ (pointwisely), then $\mathbf{rec}(e)(G) \sim_! \mathbf{rec}(e')(G)$. Then using this, we can show that if e is bisimulation generic for trees, then so is $\mathbf{rec}(e)$. □

4 Bidirectional semantics of Ordered UnCAL

In this section, we provide bidirectional semantics for UnCAL$^{\circ}$ proposed in Section 2, enriching the semantics given in Section 3. We adapt our previous work [HHI+10] on unordeded UnCAL. The overall framework in the previous work—enriching forward semantics by trace information, and providing backward semantics for the forward transformation—remains unchanged. In particular, we decompose target graphs in the backward transformation based on operators having produced the trace information, and produce updated variable binding environment as a result of backward transformation. As in the forward semantics, the order between the branches of each node is respected in the backward transformation. The major difference between the bidirectional semantics of unordered UnCAL and that of UnCAL$^{\circ}$ is that branches are represented as ordered lists instead of sets. However, the fact that the union operator is no longer commutative does not essentially change the backward evaluation. For example, when we unify the result of binary operation, the order between the operands are not reflected.

Bidirectional Properties Our goal in bidirectionalizing UnCAL$^{\circ}$ is to have bidirectional properties inherited from the previous work [HHI+10]. 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 of $\{\$x \mapsto X, \dots\}$ where X is a graph G or a label l. The following are two properties we aim at:

$$\frac{\mathscr{F}[\![e]\!]\rho = G}{\mathscr{B}[\![e]\!](\rho, G) = \rho}(\textsc{GetPut}) \qquad \frac{\mathscr{B}[\![e]\!](\rho, G') = \rho' \quad \mathscr{F}[\![e]\!]\rho' = G''}{\mathscr{B}[\![e]\!](\rho, G'') = \rho'}(\textsc{WPutGet})$$

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 views have the same effect on the original source if backward evaluation is applied. A pair of forward and backward evaluations is *well-behaved* if it satisfies (GETPUT) and (WPUTGET) properties. In the rest of this paper, we will present forward and backward evaluations (bidirectional semantics) for UnCAL$^{\circ}$ and prove the following theorem.

Theorem 2 (Well-behavedness) *The proposed forward and backward evaluations are well-behaved, provided their evaluations succeed.* □

In the rest of this section, we first provide forward semantics enriched with trace information, and then define backward semantics using the trace information.

4.1 Traceable Forward Evaluation

As in our previous work [HHI+10], we augment node IDs of the view with trace information. The trace information has a structure similar to the index structure of products and coproducts in Definition 6 and 7, except that we add code position information for bidirectionalization. The

trace ID is defined by

$$\begin{aligned} \mathit{TraceID} ::= {} & \mathit{SrcID} \mid \mathsf{Code}\ \mathit{Pos}\ \mathit{Marker} \\ & \mid \mathsf{RecN}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{Marker} \mid \mathsf{RecE}\ \mathit{Pos}\ \mathit{TraceID}\ \mathit{TraceID}\ \mathit{Num}, \end{aligned}$$

where *SrcID* ranges over identifiers uniquely assigned to all nodes of the database, *Pos* ranges over code positions in the UnCAL$^{\circ}$ expression, *Marker* ranges over input/output markers, and *Num* stands for integers. This trace ID is different from that of the previous work in that in the RecE part, *TraceID* and ordinal *Num* is used instead of edge. The *Num* part represents from which branch of the input graph of **rec** the node is produced. In unordered setting, the position in the set of branches are not important, so the edge, i.e, triple of source node, label and target node was sufficient. In ordered setting, he have to keep track of the position. For example, if a node w is generated from the i-th branch (l, v_t) of node v_s through the **rec** construct at the code position p, then trace ID of the form $\mathsf{RecE}\ w\ v_s\ i$ is produced.

4.2 Enriched Forward Semantics

We now formally define the enriched forward semantics for finite graphs by simple augmentation of trace IDs to the semantics in the previous section. We still use (V,B,I) representation, but we use *TraceID* structure for nodes in V instead of products and coproducts. In addition, we replace the composition of markers in $X \times Z$ by a monoid $(\cdot, \&)$, with $\&$ as the identity, i.e., $\& \cdot \& x = \& x \cdot \& = \& x$. The type of **rec** will be $(\mathscr{L} \times DB_Y \to DB_Z^Z) \to DB_Y^X \to DB_{Z\cdot Y}^{Z\cdot X}$, where $Z \cdot X$ is defined by $\{\& z \cdot \& x \mid \& z \in Z, \& x \in X\}$. In particular, $\{\&\} \cdot Z = Z \cdot \{\&\} = Z$ for any Z. This treatment is inherited from [BFS00], and easier to implement. It is not for the sake of bidirectionalization. For example, $\mathbf{rec}(\lambda(\$l, \$t).e)(G) \in DB_Y^X$ for $e : (\mathscr{L} \times DB_Y \to DB_{\&})$ and $G \in DB_Y^X$, meaning that a **rec** with the body that consists only of the default marker does not change the type of their input graphs through applications. So $+\!\!+$ in $\mathbf{rec}(\lambda(\$l, \$t).e)(G) +\!\!+ G$ is well defined (recall that operands of $+\!\!+$ must have identical set of input markers). In order for the new **rec** to be well-defined, however, we should restrict the combination of Z and X so that $Z \cdot X$ is isomorphic to $Z \times X$. For example, if both Z and X include the default marker $\&$ and a non-default marker in common, then I component in the result of **rec** is ill-defined, mapping identical input marker to more than one nodes. In practice, Z either contains only the default marker, or contains only non-default markers. In the following definitions, e^p denotes an UnCAL$^{\circ}$ subexpression e at code position p, $\rho(\$x)$ denotes G when $(\$x \mapsto G) \in \rho$. ρ is naturally used as variable substitution in UnCAL$^{\circ}$ expressions, e.g., $e\rho$ for an expression e. As in [HHI+10], we inductively define the enriched forward semantics $\mathscr{F}[\![e^p]\!]\rho$ for each UnCAL$^{\circ}$ construct of e.

Graph Constructor Expressions. The semantics of graph constructor expressions is a straightforward extension of that in section 3. For instance, we have

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

Note that we have nonempty B despite there is no edge in the graph.

As another example, the semantics for the expression $e_1 +\!\!+ e_2$ is defined below.

$$\mathscr{F}[\![(e_1 +\!\!+ e_2)^p]\!]\rho = \mathscr{F}[\![e_1]\!]\rho +\!\!+^p \mathscr{F}[\![e_2]\!]\rho \quad G_1 +\!\!+^p G_2 = (V \cup G_1.\mathrm{V} \cup G_2.\mathrm{V}, B \cup G_1.\mathrm{B} \cup G_2.\mathrm{B}, I)$$
$$\begin{aligned} \text{where } M &= \mathsf{inMarker}(G_1) = \mathsf{inMarker}(G_2) \qquad V = \{\mathsf{Code}\ p\ \&m \mid \&m \in M\} \\ B &= \{\mathsf{Code}\ p\ \&m \mapsto [\mathsf{E}(\varepsilon,\ I_1(\&m)), \mathsf{E}(\varepsilon,\ I_2(\&m))] \mid \&m \in M\} \\ I &= \{\&m \mapsto \mathsf{Code}\ p\ \&m \mid \&m \in M\}, \end{aligned}$$

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

$e_1 \oplus e_2$ It performs a componentwise union like $+\!\!+$, except that no ε-edges are involved.

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

where $\oplus$ is a componentwise union operator for two graphs. A graph $G_1 \oplus G_2$ is defined by

$$\begin{aligned} G_1 \oplus G_2 &= (V_1 \cup V_2, B_1 \cup B_2, I_1 \cup I_2) \\ &\quad \text{where } (V_1, B_1, I_1) = G_1 \\ &\qquad\ \ (V_2, B_2, I_2) = G_2 \\ &\qquad\ \ \mathsf{inMarker}(G_1) \cap \mathsf{inMarker}(G_2) = \emptyset \end{aligned}$$

Nullary constructors $\&y$ **and** $()$ These expressions construct constant graphs: $\&y$ constructs a node with a default input marker $\&$ and an output marker $\&y$, and $()$ constructs the empty graph.

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

$[l : e]$

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

Trace ID Code is generated for the newly constructed node.

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

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

where $@^p$ is an append operator for two graphs concerning position p. A graph $G_1 \mathbin{@}^p G_2$ is defined by

$$\begin{aligned} G_1 \mathbin{@}^p G_2 &= (G_1.\mathrm{V} \cup G_2.\mathrm{V}, B_1' \cup G_2.\mathrm{B}, G_1.\mathrm{I}) \\ &\quad \text{where } B_1' = \left\{ u \mapsto \begin{bmatrix} x.i = (l,v) & \Rightarrow & (l,v) \\ = \&m & \Rightarrow & (\varepsilon, G_2.\mathrm{I}(\&m)) \end{bmatrix}_{i \in |x|} \mid (u \mapsto x) \in G_1.\mathrm{B} \right\} \end{aligned}$$

Note that @ may introduce unreachable parts in the right operand due to unmatched input/output nodes.
$\&m := e$ It distributes the marker on the left operand to each of the input markers of the graph in the right operand, using the Skolem function "." introduced at the beginning of this section.

$$\mathscr{F}[\![(\&m := e)^p]\!]\rho = (\&m := \mathscr{F}[\![e]\!]\rho),$$

where $:=$ is an operator for a marker distribution for a graph. A graph $(\&m := G)$ is defined by

$$\begin{aligned}(\&m := G) &= (V, E, I') \\ \text{where } (V, E, I, O) &= G \\ I' &= \{\&m.\&x \mapsto v \mid (\&x \mapsto v) \in I\}\end{aligned}$$

cycle(e) It is defined as follows

$$\mathscr{F}[\![(\mathbf{cycle}(e))^p]\!]\rho = \mathbf{cycle}^p(\mathscr{F}[\![e]\!]\rho) \qquad \mathbf{cycle}^p(G) = (G.\mathrm{V}, B', G.\mathrm{I})$$
$$\text{where } B'(u) = \begin{bmatrix} G.\mathrm{B}(u).i = \mathsf{E}(l,\ v) & \Rightarrow\ \mathsf{E}(l,\ v) \\ = \mathsf{O}(\&m) \wedge \&m \in dom(I) & \Rightarrow\ \mathsf{E}(\varepsilon,\ I(\&m)) \\ = \mathsf{O}(\&m) \wedge \&m \notin dom(I) & \Rightarrow\ \mathsf{O}(\&m) \end{bmatrix}_{i \in |G.\mathrm{B}(u)|}$$

where $\mathbf{cycle}^p$ is a cycle operator for a graph concerning position p.

Variable and Condition. They have the following obvious definitions.

$$\mathscr{F}[\![(\$v)^p]\!]\rho = \rho(\$v) \qquad \mathscr{F}[\![(\mathbf{if}\, l_1 = l_2\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2)^p]\!]\rho = \begin{cases} \mathscr{F}[\![e_1]\!]\rho & \text{if } l_1\rho = l_2\rho \\ \mathscr{F}[\![e_2]\!]\rho & \text{otherwise.} \end{cases}$$

rec$(\lambda(\$l, \$g).e_{\mathbf{b}})(e_{\mathbf{a}})$ The semantics of a structural recursion is given by *bulk semantics* as defined in Section 3. Following [HHI+10], we define the enriched forward semantics of structural recursion by two auxiliary functions $\mathsf{compose}_{\mathbf{rec}}$ and fwd_eachedge:

$$\begin{aligned}\mathscr{F}[\![(\mathbf{rec}(\lambda(\$l, \$g).e_{\mathrm{b}})(e_{\mathrm{a}}))^p]\!]\rho &= \mathsf{compose}^p_{\mathbf{rec}}(\mathsf{fwd\_eachedge}(G_{\mathrm{a}}, \rho, e_{\mathrm{b}}), G_{\mathrm{a}}, M) \\ \text{where } M &= \mathrm{inMarker}(e_{\mathrm{b}}) \cup \mathrm{outMarker}(e_{\mathrm{b}}) \\ G_{\mathrm{a}} &= \mathscr{F}[\![e_{\mathrm{a}}]\!]\rho\end{aligned}$$

The function fwd_eachedge evaluates the body expression e_{b} for each edge of the argument graph G_{a} obtained by evaluating e_{a} and the set of result graphs. The function $\mathsf{compose}_{\mathbf{rec}}$ glues all of those results together along the structure of G_{a}. These functions are defined by

$$
\begin{aligned}
&\mathsf{fwd\_eachedge}(G,\rho,e) \;=\; \{(v,i,v_i,\mathscr{F}[\![\bar{e}]\!]\rho_{v,i}) \\
&\qquad \mid\; v \in G.\mathrm{V},\; x \in G.\mathrm{B}(v),\; i \in |x|,\; \mathsf{E}\,(l_i,\; v_i) = x.i,\; \rho_{v,i} = \rho \cup \{\$l \mapsto l_i, \$g \mapsto G|_v\}\} \\
&\mathsf{compose}^{p}_{\mathbf{rec}}(\mathscr{G},G,M) \;=\; (V_{\mathsf{RecE}} \cup V_{\mathsf{RecN}}, B_{\mathsf{RecE}} \cup B_{\mathsf{RecN}}, I_{\mathsf{RecN}}) \\
&\quad \text{where } V_{\mathsf{RecE}} \;=\; \{\mathsf{RecE}\; p\; w\; v\; i \;\mid\; (v,i,\_,G_{v,i}) \in \mathscr{G},\; w \in G_{v,i}.\mathrm{V}\} \\
&\qquad\quad B_{\mathsf{RecE}} \;=\; \left\{ \mathsf{RecE}\; p\; w\; v\; i \mapsto \begin{bmatrix} x.j = \mathsf{E}\,(l_j,\; w_j) \;\Rightarrow\; \mathsf{E}\,(l_j,\; \mathsf{RecE}\; p\; w_j\; v\; i) \\ \quad = \mathsf{O}\,(\&m) \quad \Rightarrow\; \mathsf{E}\,(\varepsilon,\; \mathsf{RecN}\; p\; v_i\; \&m) \end{bmatrix}_{j \in |x|} \right. \\
&\qquad\qquad \left. \mid\; (v,i,v_i,G_{v,i}) \in \mathscr{G},\; w \in G_{v,i}.\mathrm{V},\; x = G_{v,i}.\mathrm{B}(w) \right\} \\
&\qquad\quad V_{\mathsf{RecN}} \;=\; \{\mathsf{RecN}\; p\; v\; \&m \;\mid\; v \in G.\mathrm{V},\; \&m \in M\} \\
&\qquad\quad B_{\mathsf{RecN}} \;=\; \{\mathsf{RecN}\; p\; v\; \&m \mapsto \\
&\qquad\qquad\qquad \begin{bmatrix} x.i = \mathsf{E}\,(l_i,\; v_i) \;\Rightarrow\; \mathsf{E}\,(\varepsilon,\; \mathsf{RecE}\; p\; G_{v,i}.\mathrm{I}(\&m)\; v\; i) \;\text{ where } (v,i,\_,G_{v,i}) \in \mathscr{G} \\ \quad = \mathsf{O}\,(\&n) \quad \Rightarrow\; \mathsf{O}\,(\&m \cdot \&n) \end{bmatrix}_{i \in |x|} \\
&\qquad\qquad\qquad \mid\; v \in G.\mathrm{V},\; x = G.\mathrm{B}(v),\; \&m \in M\} \\
&\qquad\quad I_{\mathsf{RecN}} \;=\; \{\&n \cdot \&m \mapsto \mathsf{RecN}\; p\; v\; \&m \;\mid\; G.\mathrm{I}(\&n) = v,\; \&m \in M\}
\end{aligned}
$$

4.3 Backward Semantics

Thanks to the bulk semantics defined in Section 3 and trace information generated at enriched forward evaluation, the similarity in shape between input and output graphs are retained. As in the previous work [HHI+10], the graph constructors become invertible, and backward evaluation of $\mathbf{rec}(e)$ is reduced to that of its body e. We again divide 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.

4.3.1 Backward Semantics for Edge Renaming

Backward semantics is, similarly to the forward semantics, defined inductively, i.e., backward evaluation of an UnCAL$^{\circ}$ expression is computed by combining results of backward evaluations of the operand expressions. We first define backward semantics of simple expressions, followed by that of structural recursion.

Graph Constructor Expressions. $[]$, $\&y$ and $()$ construct constant graphs in the forward computation. Therefore, for the backward computation, they accept no modification on the result view.

$$
\begin{aligned}
\mathscr{B}[\![[]^p]\!](\rho,G') &= \rho \qquad \text{if } G' = \mathscr{F}[\![[]^p]\!]\rho \\
\mathscr{B}[\![\&m^p]\!](\rho,G') &= \rho \qquad \text{if } G' = \mathscr{F}[\![\&m^p]\!]\rho \\
\mathscr{B}[\![()^p]\!](\rho,G') &= \rho \qquad \text{if } G' = \mathscr{F}[\![()^p]\!]\rho
\end{aligned}
$$

A label constant similarly accepts no modification.

$$
\mathscr{B}[\![\mathtt{a}]\!](\rho,a') = \rho \qquad \text{if } a' = \mathtt{a}
$$

$[l : e]$ Backward computation 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').

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

Here, the decomposition function is defined as follows:

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

The modified view G' is decomposed into its unique root branch $\zeta' = [\mathsf{E}\,(a_1',\ v)\,]$ 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.

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

$$\begin{aligned}&\mathscr{B}[\![(e_1 +\!\!+ e_2)^p]\!](\rho,G') = \mathscr{B}[\![e_1]\!](\rho,G_1') \uplus_\rho \mathscr{B}[\![e_2]\!](\rho,G_2') \\ &\quad \text{where } (G_1,G_2) = (\mathscr{F}[\![e_1]\!]\rho, \mathscr{F}[\![e_2]\!]\rho) \qquad (G_1',G_2') = \mathsf{decomp}_{G_1 +\!\!+ G_2}(G') \\ &(\rho_1 \uplus_\rho \rho_2)(\$v) \;=\; \mathsf{mg}(\rho(\$v), \rho_1(\$v), \rho_2(\$v)) \\ &\quad \text{where } \mathsf{mg}(G,G_1,G_2) = \begin{cases} G_1 & \text{if } G_2 = G \vee G_1 = G_2 \\ G_2 & \text{if } G_1 = G \\ \textsc{Fail} & \text{otherwise} \end{cases}\end{aligned}$$

decomp is defined below. $G_1 \setminus G_2$ denotes $(G_1.\mathrm{V} \setminus G_2.\mathrm{V}, G_1.\mathrm{B} \setminus G_2.\mathrm{B}, G_1.\mathrm{I} \setminus G_2.\mathrm{I})$ where $G_1.\mathrm{B} \setminus G_2.\mathrm{B}$ and $G_1.\mathrm{I} \setminus G_2.\mathrm{I}$ respectively denote removal of bindings in $G_2.\mathrm{B}$ and $G_2.\mathrm{I}$ from those in $G_2.\mathrm{B}$ and $G_2.\mathrm{I}$. Simple union of graphs $G_1 \cup G_2$ is defined by $(G_1.\mathrm{V} \cup G_2.\mathrm{V}, G_1.\mathrm{B} \cup G_2.\mathrm{B}, G_1.\mathrm{I} \cup G_2.\mathrm{I})$.

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

$$\begin{aligned}&\mathsf{unreachable}(G) = G \setminus \mathsf{reachable}(G) \\ &\mathsf{xreachable}(G',G) = \mathsf{reachable}(G') \cup \mathsf{unreachable}(G)\end{aligned}$$

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

$$\begin{aligned}
&\mathscr{B}[\![(e_1 \oplus e_2)^p]\!](\rho, G') = \mathscr{B}[\![e_1]\!](\rho, G'_1) \uplus_\rho \mathscr{B}[\![e_2]\!](\rho, G'_2) \\
&\quad \text{where } G_i \qquad\qquad = \mathscr{F}[\![e_i]\!]\rho \\
&\qquad\quad (G'_1, G'_2) \qquad = \mathsf{decomp}_{G_1 \oplus G_2}(G') \\
&\qquad\quad \mathsf{decomp}_{G_1 \oplus G_2}(G') = \mathsf{decomp}_{G_1 \cup G_2}(G') \\
&\qquad\qquad\qquad\qquad\quad \text{without } \textit{satisfying} \text{ condition}
\end{aligned}$$

$e_1 \mathbin{@} e_2$ 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 because these parts are part of ordinary computation (computation on reachable parts) before discarding by the @ operator.

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

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

$$\begin{aligned}
&\mathscr{B}[\![\&m := e]\!](\rho, G') = \mathscr{B}[\![e]\!](\rho, G'_1) \\
&\quad \text{where } G'_1 \qquad = (V', B', I'_1) \\
&\qquad\quad (V', E', I') = G' \\
&\qquad\quad I'_1 \qquad = \{(\&x \mapsto v) \mid (\&m.\&x \mapsto v) \in I'\}
\end{aligned}$$

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

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

Variable. A variable simply updates its binding 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 backward evaluation of a condition is defined by

$$\mathscr{B}[\![\textbf{if}\, l_1 = l_2\, \textbf{then}\, e_1\, \textbf{else}\, e_2]\!](\rho, G') = \begin{cases} \rho_1' & \text{if } l_1\rho = l_2\rho \wedge l_1\rho_1' = l_2\rho_1' \\ \rho_2' & \text{if } l_1\rho \neq l_2\rho \wedge l_1\rho_2' \neq l_2\rho_2' \\ \textsc{Fail} & \text{otherwise} \end{cases}$$
$$\text{where } (\rho_1', \rho_2') = (\mathscr{B}[\![e_1]\!](\rho, G'), \mathscr{B}[\![e_2]\!](\rho, G'))$$

which is reduced to the backward evaluation of e_1 if $l_1 = l_2$ holds, and to the backward evaluation of e_2 otherwise. To guarantee well-behavedness, we further ensure that $l_1 = l_2$ does not change after backward evaluation. Because the result of $l_1 = l_2$ may be influenced indirectly by backward evaluation of the bodies e_1 and e_2, if they update variable bindings and the value of label variables in the condition is produced from the updated bindings. We check this by returning the condition $l_1 = l_2$ in addition to the variable bindings, and check that condition at the expression binding the label variables. However, to simplify presentation, we omit this passing around of conditions and checking in the following.
Backward Evaluation of Structural Recursion. Backward evaluation is defined by

$$\begin{aligned} &\mathscr{B}[\![\textbf{rec}(\lambda(\$l, \$g).\, e_\mathrm{b})(e_\mathrm{a})]\!](\rho, G') \\ &\quad = \mathsf{merge}(\rho, e_\mathrm{a}, B_\mathrm{a}, \mathsf{bwd\_eachedge}(G_\mathrm{a}, \rho, e_\mathrm{b}, \mathsf{decomp}_{\textbf{rec}}(G', B_\mathrm{a}))) \\ &\quad \text{where } G_\mathrm{a} = (\_, B_\mathrm{a}, \_) = \mathscr{F}[\![e_\mathrm{a}]\!]\rho, \end{aligned}$$

where $\mathsf{decomp}_{\textbf{rec}}$ performs the inverse operation of $\mathsf{compose}^{p}_{\textbf{rec}}(\mathscr{G}, G, M)$ by decomposing the entire updated view into the updated views of the body e_b using bulk semantics. In bwd_eachedge, we carry out backward computation of e_b on each edge to obtain the updated environment $\rho'_{v,i}$. Finally, merge combines these environments to produce the updated environment ρ' of the whole expression. merge first merge $\rho_{v,i}(\$l)$ and $\rho'_{v,i}(\$g)$ for each binding of $\$l$ and $\$g$ to compute G'_a, which is the updated view of the argument expression e_a. Unlike unordered semantics, we cannot just simply unify each $\rho'_{v,i}(\$g).\mathrm{B}$ or $\{v \mapsto [\mathsf{E}(\rho_{v,i}(\$l),\ \rho'_{v,i}(\$g).\mathrm{I}(\&))]\}$ as sets, since there are overlapping originating nodes among them, in addition to possible overlap between the edge for $\rho_{v,i}(\$l)$ and graph $\rho'_{v,i}(\$g)$ in the presence of cycles. We let the backward evaluation fails if these overlapped parts are inconsistent. We first unify $\rho'_{v,i}(\$g)$ with this check, and unify $\{v \mapsto [\mathsf{E}(\rho_{v,i}(\$l),\ \rho'_{v,i}(\$g).\mathrm{I}(\&))]\}$, maintaining the order in the original branches in B_a (the result is stored in B'). If the original label in B_a was ε, then we just restore the ε edge. Output marker of the input is also recovered. If the edge is in the prior graph (graphs created by merging $\rho'(\$g)$), which means every edge from the originating node is in the prior graph, then they must coincide with the edge in the prior graph. If the edge is not in the prior graph, then we add all edges from the originating node together (elements of B'). Above unification produces G'_a. Then, we inductively carry out backward evaluation on e_a to obtain another updated environment ρ'_a. This ρ'_a and all $\rho'_{v,i}$s (excluding binding of $\$l$ and $\$g$) are merged to produce the result of entire backward evaluation.

$$\mathsf{decomp}_{\mathbf{rec}}((V',B',I'),B_{\mathrm{a}}) =$$
$$\left\{ (\zeta,(V'_\zeta,B'_\zeta,I'_\zeta)) \;\middle|\; \begin{array}{l} B_{\mathrm{a}}(v)=b, i\in|b|, \mathsf{E}(l,\ v_i)=b.i, \zeta=(v,i,v_i), \\ V'_\zeta=\{w \mid (\mathsf{RecE}\ p\ w\ v\ i)\in V'\}, \\ B'_\zeta(w)=\left[\begin{array}{ll} s.j=\mathsf{E}(l_j,\ \mathsf{RecE}\ p\ w_j\ v\ i) & \Rightarrow\ \mathsf{E}(l_j,\ w_j) \\ \quad=\mathsf{E}(\varepsilon,\ \mathsf{RecN}\ p\ v_i\ \&m) & \Rightarrow\ \mathsf{O}(\&m) \end{array}\right]_{j\in|B'(\mathsf{RecE}\ p\ w\ v\ i)|} \\ \qquad\qquad s=B'(\mathsf{RecE}\ p\ w\ v\ i) \\ I'_\zeta=\{\&m\mapsto w \mid B'(\mathsf{RecN}\ p\ v\ \&m)=s, j\in|s|, \mathsf{E}(\varepsilon,\ \mathsf{RecE}\ p\ w\ v\ i)=s.j\} \end{array} \right\}$$
$$\mathsf{bwd\_eachedge}(G,\rho,e,\mathscr{G}'_{v,i}) = \{(\zeta,\mathscr{B}[\![e]\!](\rho_{v,i},G'_{v,i}))$$
$$\quad \mid (\zeta,G'_{v,i})\in\mathscr{G}', (v,i,v_i)=\zeta, \mathsf{E}(a,\ v_t)=G.\mathrm{B}(v).i, \rho_{v,i}=\rho\cup\{\$l\mapsto a, \$g\mapsto G|_{v_t}\}\}$$
$$\mathsf{merge}(\rho,e_{\mathrm{a}},B_{\mathrm{a}},\mathscr{R}) = \mathscr{B}[\![e_{\mathrm{a}}]\!](\rho,G'_{\mathrm{a}}) \uplus_\rho \biguplus\{\rho'_{v,i}\setminus\{\$l\mapsto\_\}\setminus\{\$g\mapsto\_\} \mid ((v,i,v_i),\rho'_{v,i})\in\mathscr{R}\}$$
$$\text{where}\ \ G'_{\mathrm{a}} = \left(V'\cup\bigcup\rho'_{v,i}(\$g).\mathrm{V}, B'\cup\bigcup\rho'_{v,i}(\$g).\mathrm{B},\ I_{\mathrm{a}}\right)$$
$$V' = \cup\{\{v\}\cup\{w \mid \mathsf{E}(l,\ w)\in s\} \mid (v\mapsto s)\in B'\}$$
$$B'(v) = \left[\begin{array}{lll} B_{\mathrm{a}}(v).j=\mathsf{E}(\varepsilon,\ v_j) & \Rightarrow & \mathsf{E}(\varepsilon,\ v_j) \\ \quad=\mathsf{E}(l,\ v_j) & \Rightarrow & \mathsf{E}(\rho_{v,j}(\$l),\ v_j) \\ \quad=\mathsf{O}(\&m) & \Rightarrow & \mathsf{O}(\&m) \end{array}\right]_{j\in|B_{\mathrm{a}}(v)|}$$

4.4 Backward Semantics for Edge Deletion

Similarly to the treatment in the previous work, we reflect deletion of an edge in the view as deletion of the corresponding edge in the 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.

$$\begin{array}{ll} \mathsf{corr}((u,i)) = (u,i) & \text{if } u\in \mathit{SrcID} \\ \mathsf{corr}((\mathsf{RecE}\ p\ u\ v\ i,j)) & \\ \qquad = \begin{cases} \mathsf{corr}((u,j)) & \text{if } \mathsf{corr}((u,j))\neq \textsc{Fail} \\ \mathsf{corr}((v,i)) & \text{if } \mathsf{corr}((u,j))= \textsc{Fail} \end{cases} & \\ \mathsf{corr}(\zeta) = \textsc{Fail} & \text{otherwise.} \end{array}$$

FAIL means failure on finding the corresponding edge.

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 of 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 this removal should take place at a time, since if multiple branches of a node are to be deleted, one removal of branch will change the relative position of the following (sibling) branches.

Finally, we check if $\mathscr{F}[\![e]\!]\rho[\$db\leftarrow G'_{\mathrm{src}}]$ is bisimilar to G'_{view}, and returns $\rho[\$db\leftarrow G'_{\mathrm{src}}]$ if the bisimulation test succeeds.

It is straightforward to show that the backward semantics for edge deletion is well behaved, since the final check does (WPUTGET) property.

4.5 Backward Semantics for Insertion

The method used in the previous work for handling insertion can be adopted here. We briefly discuss this adoption.

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.

$$\mathsf{tr}(\mathit{SrcID}) = \mathit{SrcID} \qquad \mathsf{tr}(\mathsf{RecN}\ \_\ v\ \_) = \mathsf{tr}(v)$$
$$\mathsf{tr}(\mathsf{Code}\ \_\ \_) = \textsc{Fail} \qquad \mathsf{tr}(\mathsf{RecE}\ \_\ v\ \_\ \_) = \mathsf{tr}(v)$$

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) [AG02], where G'_{view} is obtained by adding εedge 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.

4.6 Well-behavedness

Now we turn to the well-behavedness of the bidirectional transformation defined above, and briefly sketch the proof.

Proof. This statement can be proved by induction on the structure of e, in a way similar to that in [HHI+10]. Note that we perform aggressive check for branch changing behavior of **if** for this well-behavedness. □

5 Conclusion

In this paper, we report our ongoing work on extension of the previous method for bidirectionalizing graph transformations from unordered graphs to ordered ones, and demonstrate that this is possible by defining a new graph transformation language UnCAL$^{\text{o}}$, a novel definition of bisimulation relation on ordered graphs with ε, and a bulk and bidirectional semantics of the graph language UnCAL$^{\text{o}}$. This paves a new way towards practically solving the open problem of unidirectional and bidirectional transformation over ordered graphs.

In our definition of structured recursion, we implicitly assume that the order of children does not change during transformation. This assumption, however, rules out transformations which reorder the children. For instance, the current UnCAL$^{\text{o}}$ cannot define a transformation that reverses the children. In the future, we wish to relax this limitation introducing a new graph constructor that is similar to the append constructor $+\!\!+$ but can swap the order of two operand graphs. As another important future work, we will study how to efficiently implement the new bidirectional semantics of UnCAL$^{\text{o}}$, and evaluate it with practical applications.

Bibliography

[AG02] S. M. Abramov, R. Glück. Principles of Inverse Computation and the Universal Resolving Algorithm. In *The Essence of Computation*. Pp. 269–295. 2002.

[BFP+08] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, A. Schmitt. Boomerang: resourceful lenses for string data. In Necula and Wadler (eds.), *POPL '08: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. Pp. 407–419. ACM, 2008.

[BFS00] P. Buneman, M. F. Fernandez, D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases* 9(1):76–110, 2000.

[BJ10] Bart, Jacobs. From Coalgebraic to Monoidal Traces. *Electronic Notes in Theoretical Computer Science* 264(2):125 – 140, 2010. Proceedings of the Tenth Workshop on Coalgebraic Methods in Computer Science (CMCS 2010).

[BS81] F. Bancilhon, N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems* 6(4):557–575, 1981.

[CFH+09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *International Conference on Model Transformation (ICMT 2009)*. Pp. 260–283. LNCS 5563, Springer, 2009.

[FGM+05] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. Pp. 233–246. 2005.

[Heg90] S. J. Hegner. Foundations of Canonical Update Support for Closed Database Views. In *ICDT '90: Proceedings of the Third International Conference on Database Theory*. Pp. 422–436. Springer-Verlag, London, UK, 1990.

[HHI+10] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano. Bidirectionalizing Graph Transformations. In *ACM SIGPLAN International Conference on Functional Programming*. Pp. 205–216. ACM, 2010.

[HMT08] Z. Hu, S.-C. Mu, M. Takeichi. A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations. *Higher-Order and Symbolic Computation* 21(1-2):89–118, 2008.

[Läm04] R. Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*. Nov. 2004.

[Mee98] L. Meertens. Designing Constraint Maintainers for User Interaction. June 1998. http://www.cwi.nl/~lambert.

[MHN+07] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, M. Takeichi. Bidirectionalization Transformation based on Automatic Derivation of View Complement Functions. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. Pp. 47–58. ACM Press, Oct. 2007.

[RSGV09] E. L. Robertson, L. V. Saxton, D. V. Gucht, S. Vansummeren. Structural Recursion as a Query Language on Lists and Ordered Trees. *Theory of Computing Systems* 44(4):590–619, 2009.

[Rut00] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249(1):3 – 80, 2000.

[SR11] D. Sangiorgi, J. Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011.

[VHMW10] J. Voigtländer, Z. Hu, K. Matsuda, M. Wang. Combining Syntactic and Semantic Bidirectionalization. In *ACM SIGPLAN International Conference on Functional Programming*. Pp. 181–192. ACM, 2010.

[Voi09] J. Voigtländer. Bidirectionalization for free! (Pearl). In *POPL '09: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. Pp. 165–176. ACM, New York, NY, USA, 2009.

[WGW11] M. Wang, J. Gibbons, N. Wu. Incremental updates for efficient bidirectional transformations. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. ICFP '11, pp. 392–403. ACM, New York, NY, USA, 2011.