ISSN 1884-0760

GRACE TECHNICAL REPORTS

Bidirectionalizing Graph Transformations

Soichiro Hidaka Zhenjiang Hu Kazuhiro Inaba
Hiroyuki Kato Kazutaka Matsuda Keisuke Nakano

GRACE-TR 2010–06 July 2010

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/

Bidirectionalizing Graph Transformations *

Soichiro Hidaka Zhenjiang Hu
Kazuhiro Inaba Hiroyuki Kato
National Institute of Informatics, Japan
{hidaka,hu,kinaba,kato}@nii.ac.jp

Kazutaka Matsuda
Tohoku University, Japan
kztk@kb.ecei.tohoku.ac.jp

Keisuke Nakano
The University of
Electro-Communications, Japan
ksk@cs.uec.ac.jp

Abstract

Bidirectional transformations provide a novel mechanism for synchronizing and maintaining the consistency of information between input and output. Despite many promising results on bidirectional transformations, these have been limited to the context of relational or XML (tree-like) databases. We challenge the problem of bidirectional transformations within the context of graphs, by proposing a formal definition of a well-behaved bidirectional semantics for UnCAL, i.e., a graph algebra for the known UnQL graph query language. The key to our successful formalization is full utilization of both the recursive and bulk semantics of structural recursion on graphs. We carefully refine the existing forward evaluation of structural recursion so that it can produce sufficient trace information for later backward evaluation. We use the trace information for backward evaluation to reflect in-place updates and deletions on the view to the source, and adopt the universal resolving algorithm for inverse computation and the narrowing technique to tackle the difficult problem with insertion. We prove our bidirectional evaluation is well-behaved. Our current implementation is available online and confirms the usefulness of our approach with nontrivial applications.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; E.1 [*Data Structures*]: Graphs and networks

General Terms Design, Languages

Keywords bidirectional transformation, view updating, graph query and transformation, structural recursion

1. Introduction

Bidirectional transformations (Czarnecki et al. 2009; Foster et al. 2005) provide a novel mechanism for synchronizing and maintaining the consistency of information between input and output. They consist of a pair of *well-behaved* transformations: *forward* transformation is used to produce a target view from a source, while the *backward* transformation is used to reflect modification on the view to the source. This pair of forward and backward transformations should satisfy certain bidirectional properties. Bidirectional transformations are indeed pervasive and can be seen in many interesting applications, including the synchronization of replicated data in different formats (Foster et al. 2005), presentation-oriented structured document development (Hu et al. 2008), interactive user interface design (Meertens 1998), coupled software transformation (Lämmel 2004), and the well-known *view updating* mechanism which has been intensively studied in the database community (Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988; Hegner 1990; Lechtenbörger and Vossen 2003).

Despite many promising results on bidirectional transformations, they are limited to the context of relational or XML (tree-like) databases. It remains unresolved (Czarnecki et al. 2009) whether bidirectional transformations can be addressed within the context of *graphs* containing node sharing and cycles. It would be remarkably useful in practice if bidirectional transformation could be applied to graph data structures, because graphs play an irreplaceable role in naturally representing more complex data structures such as those in biological information, WWW, UML diagrams in software engineering (Stevens 2007), and the Object Exchange Model (OEM) used for exchanging arbitrary database structures (Papakonstantinou et al. 1995).

There are many challenges in addressing bidirectional transformation on graphs. First, unlike relational or XML databases, 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 much more complicated than that on trees; naïve computation on graphs would visit the same nodes many times and possibly infinitely. It is particularly difficult to handle insertion in backward transformation because it requires a suitable subgraph to be created and inserted into a proper place in the source.

This paper reports our first solution to the problem of bidirectional graph transformation. We approach this problem by providing a bidirectional semantics for UnCAL, which is a graph algebra for the known graph query language UnQL (Buneman et al. 2000); forward semantics (forward evaluation) corresponds to forward transformation and backward semantics (backward evaluation) corresponds to backward transformation. We choose UnQL/UnCAL as the basis of our bidirectional graph transformation for two main reasons.

- First, UnQL/UnCAL is a graph query language that has been well studied in the database community with a solid foundation and efficient implementation. It has a concise and practical surface syntax based on *select-where* clauses like SQL, and can be easily used to describe many interesting graph transformations.
- Second, and more importantly, graph transformations in UnQL can be automatically mapped to those in terms of *structural recursion* in UnCAL, which can be evaluated in a *bulk* manner (Buneman et al. 2000); 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 significantly contributes to our bidirectionalization, providing a smart way of treating shared nodes and cycles in graphs and of tracing back from the view to the source.

Our main technical contributions are summarized as follows.

- We are, as far as we are aware, the first to have recognized the importance of structural recursion and its bulk semantics in addressing the challenging problem of bidirectional graph transformation, and succeeded in a novel *two-stage* framework of bidirectional graph transformation based on structural recursion. We demonstrate that graph transformations defined in terms of structural recursions (being suitable for optimization as have been intensively studied thus far (Buneman et al. 2000)) make backward evaluation easier.
- We give a formal definition of bidirectional semantics for UnCAL by (1) refining the existing forward evaluation so that it can produce useful trace information for later backward evaluation (Section 4), and (2) using the trace information to reflect in-place updates and deletions on the view to the source, and adopt the narrowing technique to tackle the difficult problem with insertion (Section 5). We prove our bidirectional evaluation is well-behaved.
- We have fully implemented our bidirectionalization presented in this paper and confirmed the effectiveness of our approach through many non-trivial examples, including all those presented in this paper and some typical bidirectional graph transformations in database management and software engineering. More examples and demos are available on our BiG project Web site*.

We consider an operation based approach, which means that the user explicitly provides editing operations in terms of "rename", "delete", and "insert". Currently these operations are treated according to the order specified by users. It might be challenging to produce these operation sequences automatically from the states before and after user's modifications on the view, but it is beyond the scope of this paper.

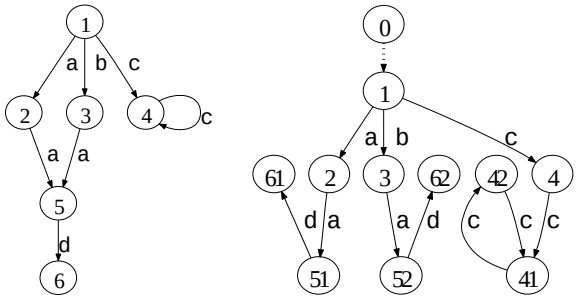
The forward transformations we consider is based on UnCAL, which is bisimulation generic, meaning that the transformation can't distinguish between graphs that are bisimilar. For example, it can't extract "first child of a node". Extending our model to cope with order is included in our future work.

Also note that backward transformation is not bisimulation generic, meaning that two results of updates that are bisimilar do not always lead to bisimilar source. However, this is not necessarily a limitation introduced by our bidirectionalization, since this asymmetry comes from the expressiveness of conditional expression in the original UnCAL graph algebra. Similar argument apply for isomorphic updates.

Outline We start with a brief review of the basic concept of a graph data model and the structural recursion of UnCAL in Section 2. Then, we clarify the bidirectional properties within our context and give an overview of our two-staged framework for bidirectionalizing graph transformations in Section 3. After explaining how to extend the forward evaluation of UnCAL with trace information in Section 4, we give a formal definition of bidirectional semantics for UnCAL and prove that it is well-behaved in Section 5. We discuss implementation issues in Section 6 and related work in Section 7. We conclude the paper in Section 8.

2. UnCAL: A Graph Algebra

We adopted UnCAL (Buneman et al. 2000), a well-studied graph algebra, as the basis of our bidirectional graph transformation. We will briefly review its graph data model and the core of UnCAL.



(a) A Simple Graph (b) An Equivalent Graph

Figure 1. Graph Equivalence Based on Bisimulation

2.1 Graph Data Model

We deal with rooted, directed, and edge-labeled graphs with no order on outgoing edges. They are edge-labeled in the sense that all information is stored on labels of edges while labels of nodes serve only as a unique identifier without a particular meaning. UnCAL 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 $Label$ to denote the set of labels and $\mathcal{M}$ to denote the set of markers.

Formally, a graph G, sometimes denoted by $G_{(V,E,I,O)}$, is a quadruple (V, E, I, O), where V is a set of nodes, $E \subseteq V \times (Label \cup \{\varepsilon\}) \times V$ is a set of edges, $I \subseteq \mathcal{M} \times V$ is a set of pairs of an input marker and the corresponding input node, and $O \subseteq V \times \mathcal{M}$ is a set of pairs of output nodes and associated output markers. For each marker $\&x \in \mathcal{M}$, there is at most one node v such that $(\&x, v) \in I$. The node v is called an *input node* with marker $\&x$ and is denoted by $I(\&x)$. 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 later. 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. In addition, we write $\mathsf{label}(\zeta)$ to denote the label of the edge ζ.

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

Value Equivalence between Graphs Two graphs are value equivalent if they are bisimilar. Please refer to (Buneman et al. 2000) for the complete definition. Informally, graph G_1 is bisimilar to graph G_2 if every node x_1 in G_1 has at least a bisimilar counterpart x_2 in G_2 and vice versa, and if there is an edge from x_1 to y_1 in G_1, then there is a corresponding edge from x_2 to y_2 in G_2 that is a bisimilar counterpart of y_1, and vice versa. Therefore, unfolding a cycle or duplicating shared nodes does not really change a graph.

* `http://www.biglab.org`

† This analogy would choose NFA rather than DFA, since we allow multiple outgoing edges with identical labels from a node.

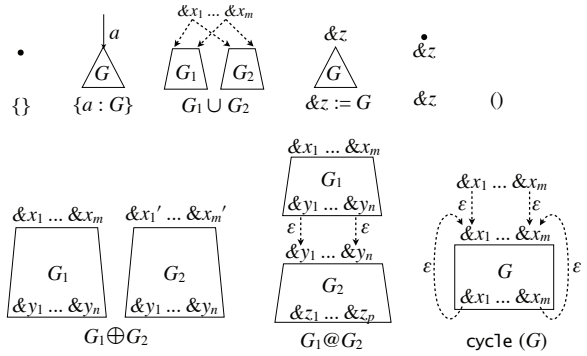


Figure 2. Graph Constructors

This notion of bisimulation is extended to cope with ε-edges. For instance, the graph in Figure 1(b) is value equivalent to the graph in Figure 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.

Graph Constructors Figure 2 summarizes the nine graph constructors that are powerful enough to describe arbitrary (directed, edge-labeled, and rooted) graphs (Buneman et al. 2000):

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

Here, $\{\}$ constructs a root-only graph, $\{a : G\}$ constructs a graph by adding an edge with label $a \in \mathit{Label} \cup \{\varepsilon\}$ pointing to the root of graph G, and $G_1 \cup G_2$ adds two ε-edges from the new root to the roots of G_1 and G_2. Also, $\&x := G$ associates an input marker, $\&x$, to the root node of G, $\&y$ constructs a graph with a single node marked with one output marker $\&y$, and $()$ constructs an empty graph that has neither a node nor an edge. Further, $G_1 \oplus G_2$ constructs a graph by using a componentwise (V, E, I and O) union. $\cup$ differs from $\oplus$ in that $\cup$ unifies input nodes while $\oplus$ does not. $\oplus$ requires input markers of operands to be disjoint, while $\cup$ 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. Newly created nodes have unique identifiers. We will give this creation rule extended for our bidirectionalization in Section 4.1. The definition here is based on graph isomorphism (identical graph construction expressions results in identical graphs up to isomorphism), and they are, together with other operators, also bisimulation generic (Buneman et al. 2000), i.e., bisimilar result is obtained for bisimilar inputs.

Example 1. The graph equivalent to that in Figure 1(a) can be constructed as follows (though not uniquely).

$$
\begin{array}{l}
\&z \mathbin{@} \mathbf{cycle}((\&z := \{\mathtt{a} : \{\mathtt{a} : \&z_1\}\} \cup \{\mathtt{b} : \{\mathtt{a} : \&z_1\}\} \cup \{\mathtt{c} : \&z_2\}) \\
\qquad \oplus (\&z_1 := \{\mathtt{d} : \{\}\}) \\
\qquad \oplus (\&z_2 := \{\mathtt{c} : \&z_2\}))
\end{array}
$$

□

$$
\begin{array}{rcll}
e & ::= & \{\} \mid \{l : e\} \mid e \cup e \mid \&x := e \mid \&y \mid () & \\
 & | & e \oplus e \mid e \mathbin{@} e \mid \mathbf{cycle}(e) & \{\text{ constructor }\} \\
 & | & \$g & \{\text{ graph variable }\} \\
 & | & \mathbf{if}\ l = l\ \mathbf{then}\ e\ \mathbf{else}\ e & \{\text{ conditional }\} \\
 & | & \mathbf{rec}(\lambda(\$l, \$g).e)(e) & \{\text{ structural recursion application }\} \\
l & ::= & a \mid \$l & \{\text{ label } (a \in \mathit{Label}) \text{ and label variable }\}
\end{array}
$$

Figure 3. Core UnCAL Language

For simplicity, we often write $\{a_1 : G_1, \ldots, a_n : G_n\}$ to denote $\{a_1 : G_1\} \cup \cdots \cup \{a_n : G_n\}$.

2.2 The Core UnCAL

UnCAL (Unstructured Calculus) is an internal graph algebra for the graph query language UnQL, and its core syntax is depicted in Figure 3. It consists of the graph constructors, variables, conditionals, and structural recursion. We have already detailed the data constructors, while variables and conditionals are self explanatory. Therefore, we will focus on *structural recursion*, which is a powerful mechanism in UnCAL to describe graph transformations.

A function f on graphs is called a structural recursion if it is defined by the following equations[‡]

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

where the expression e may contain references to variables $\$l$ and $\$g$ (but no recursive calls to f). Since the first and the third equations are common in all structural recursions, we write the structural recursion in UnCAL simply as

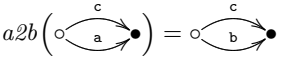
$$f(\$db) = \mathbf{rec}(\lambda(\$l, \$g).e)(\$db).$$

Despite its simplicity, the core UnCAL is powerful enough to describe interesting graph transformation including all graph queries (in UnQL) (Buneman et al. 2000), and nontrivial model transformations (Hidaka et al. 2009). Some simple examples are given below.

Example 2. The following structural recursion $a2b$ replaces edge label `a` with `b` and leaves other labels unchanged.

$$
\begin{array}{l}
a2b(\$db) = \mathbf{rec}(\lambda(\$l, \$g).\ \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{b} : \&^1\}^2 \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\qquad\quad \{\$l : \&^3\}^4)\ (\$db)^5
\end{array}
$$

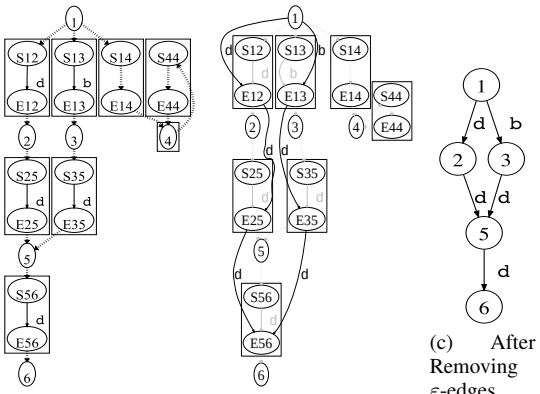
(The superscripts are for identifying code positions, which will be important in Section 4; they can simply be ignored for now.) Here is an instance of an execution:



where $\circ$ denotes the root of the graph. □

[‡] Informally, the meaning of this definition can be considered to be a fixed point (though may not necessarily unique) over the graph, which is again defined by a set of equations using the three constructors $\{\}$, :, and $\cup$. For instance, the graph in Figure 1(a) can be considered to be the fixed point of the following equations:

$$
\begin{array}{lcl}
G_{\mathrm{root}} & = & \{\mathtt{a} : \{\mathtt{a} : G_5\}, \mathtt{b} : \{\mathtt{a} : G_5\}, \mathtt{c} : G_4\} \\
G_5 & = & \{\mathtt{d} : \{\}\} \\
G_4 & = & \{\mathtt{c} : G_4\}.
\end{array}
$$



(a) Before Removing ε-edges (b) Removing ε-edges (c) After Removing ε-edges

Figure 4. Bulk Semantics of Structural Recursion in UnCAL

Example 3. The following structural recursion $a2d\_xc$ replaces all labels `a` with `d` and removes edges labeled `c`.

$$
\begin{array}{l}
a2d\_xc(\$db) = \\
\quad \mathbf{rec}(\lambda(\$l, \$g).\ \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then} \quad \{\mathtt{d} : \&^1\}^2 \\
\qquad\qquad\qquad\ \mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ \{\varepsilon : \&^3\}^4 \\
\qquad\qquad\qquad\ \mathbf{else} \quad\qquad\ \{\$l : \&^5\}^6)\ (\$db)^7
\end{array}
$$

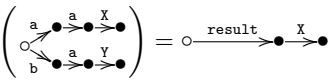
Applying the function $a2d\_xc$ to the graph in Figure 1(a) yields the graph in Figure 4(c). □

Example 4. The following structural recursion *consecutive* extracts subgraphs that can be accessible by traversing two connected edges of the same label.

$$
\begin{array}{l}
consecutive(\$db) = \\
\quad \mathbf{rec}(\lambda(\$l, \$g).\ \mathbf{rec}(\lambda(\$l', \$g').\ \\
\qquad \mathbf{if}\ \$l = \$l'\ \mathbf{then}\ \{\mathtt{result} : \$g'\}^1 \\
\qquad \mathbf{else} \quad\qquad\ \{\}^2 \qquad\qquad)(\$g)^3)(\$db)^4
\end{array}
$$

For example, we have

$$consecutive\left(\ \right) =$$



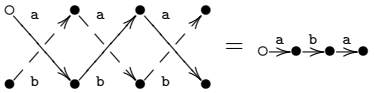
Note that the structural recursive definition of *consecutive* uses graph parameter $\$g'$ to achieve the transformation. Also note that structural recursions are allowed to be nested, and inner recursion can refer to outer variables (as $\$l$ in the example). This enables us to express the *join* of multiple queries. □

Example 5. Although the examples given so far are self-recursive, it is possible to simulate *mutual recursion* by returning graphs with multiple markers. For instance, the following function *abab*

$$
\begin{array}{l}
abab(\$db) = \&z_1\ @\ \mathbf{rec}(\lambda(\$l, \$g). \\
\qquad\qquad \&z_1 := \{\mathtt{a} : \&z_2\} \ \oplus\ \&z_2 := \{\mathtt{b} : \&z_1\})(\$db)
\end{array}
$$

changes all edges of even distances from the root node to `a`, and odd distance edges to `b`. We may consider the markers $\&z_i$ as a mutually recursive call, and *abab* to consist of two mutual recursive functions. The first is $\&z_1$, which, at each edge in the original graph, generates a new `a` edge pointing to the result of $\&z_2$ at the original destination node. The second is $\&z_2$ that generates `b` edges pointing to the result of $\&z_1$ from its destination. The result of the whole expression is defined to be the result of the $\&z_1$ at the root node of the argument graph. The following figure should be helpful. The dashed edges denote the edges that are unreachable from the output root node.

$$abab(\ ) =$$



2.3 Bulk Semantics of Structural Recursion

By allowing ε-edges, we can evaluate a structural recursion in a *bulk* manner. Consider the structural recursion,

$$\mathbf{rec}(\lambda(\$l, \$g).\ e)$$

which is to be applied to an input graph G. In bulk semantics, we apply body e *independently* on every edge (a, G_1) in G where a is the label of the edge and G_1 is the graph that the edge is pointing to, then join the results with ε-edges (as in the @ constructor).

Recall the structural recursion $a2d\_xc$ defined in Example 3. Applying it to the input graph in Figure 1(a) yields the graph in Figure 4(a), where each edge from i to j in the input graph leads to a subgraph containing a graph with an edge from Sij to Eij in the output graph (where the dotted edge denotes an ε-edge), and these subgraphs are connected with ε-edges according to the original shape of the graph. If we eliminate all ε-edges as explained in Section 3.2, we obtain a standard graph in Figure 4(c).

One distinct feature of bulk semantics is that the shape of the input graph is remembered through additional ε-edges, which will be fully utilized in our later bidirectionalization.

3. Overview: Bidirectionalizing UnCAL

It is more challenging to bidirectionalize transformations on graphs than trees, because graphs may contain shared nodes or cycles. We shall demonstrate that the structural recursion in UnCAL can serve as the basis to solve this problem. Although structural recursion was proposed within the context of query optimization, we will show that it plays a crucial role in our bidirectionalization.

3.1 Bidirectional Properties

Bidirectionalization is used to derive backward transformation from forward transformation. We approach the problem of bidirectionalization in graph transformation by providing a bidirectional semantics for UnCAL; forward semantics (forward evaluation) corresponds to forward transformation and backward semantics (backward evaluation) corresponds to backward transformation.

Before giving our bidirectional semantics for UnCAL, let us clarify the bidirectional properties that the forward and backward evaluations should satisfy. Let $\mathcal{F}[\![e]\!]\rho$ denote a forward evaluation (*get*) of expression e under environment ρ to produce a view, and $\mathcal{B}[\![e]\!](\rho, G')$ 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. ρ is a set of mappings with form $\$x \mapsto G$ with a graph (or label) G. The following are two important properties:

$$\frac{\mathcal{F}[\![e]\!]\rho = G}{\mathcal{B}[\![e]\!](\rho, G) = \rho}(\textsc{GetPut})$$

$$\frac{\mathcal{B}[\![e]\!](\rho, G') = \rho'\ \ G' \in \mathrm{Range}(\mathcal{F}[\![e]\!])}{\mathcal{F}[\![e]\!]\rho' = G'}(\textsc{PutGet})$$

The (GETPUT) property states that unchanged view G should give no change on the environment ρ in the backward evaluation, while the (PUTGET) property states that if a view is modified to G' which is in the range of the forward evaluation, then this modification can be reflected to the source such that a forward evaluation will produce the same view G'.

These two properties are essentially the same as those in (Foster et al. 2005). One problem with the (PUTGET) property is that it

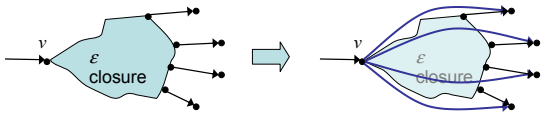


Figure 5. General ε-edge Elimination Procedure

needs to check whether a graph is in the range of forward evaluation, which is difficult to do in practice. To avoid this range checking, we allow the modified view and the view obtained by backward evaluation followed by forward evaluation to differ, but require both views to have the same effect on the original source if backward evaluation is applied.

$$\frac{\mathcal{B}[\![e]\!](\rho, G') = \rho' \quad \mathcal{F}[\![e]\!]\rho' = G''}{\mathcal{B}[\![e]\!](\rho, G'') = \rho'} (\text{WPUTGET})$$

The *get* in our (WPUTGET) can be considered as an amendment of the modified view G' to G''. Certainly, if the (PUTGET) property holds, so does the (WPUTGET).

We say that 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 give a bidirectional evaluation (semantics) for UnCAL, and prove the following theorem, which is a direct consequence of Lemmas 2, 3, and 4 that will be discussed later.

Theorem 1 (Well-behavedness)**.** *Our forward and backward evaluations are well-behaved, provided their evaluations succeed.*

3.2 Two-Stage Bidirectionalization

Recall $a2d\_xc$, which maps the source graph in Figure 1(a) to the view graph in Figure 4(c). The big gap between the source and the view makes it hard to reflect changes on the view to the source. Our idea to bridge this gap was to divide the forward evaluation into two easily handled stages:

- Stage 1: Forward evaluation (in the bulk semantics) with sufficient ε-edges, so that the output graph will have a similar shape to the input graph, making the later backward evaluation easier.
- Stage 2: Elimination of ε-edges to produce a usual view.

For $a2d\_xc$, Stage 1 maps the source graph to the intermediate graph in Figure 4(a), and Stage 2 maps the intermediate graph to the view graph (Figure 4(c)). By doing so, each stage becomes easier to bidirectionalize.

First, let us consider Stage 2. The ε-edge elimination procedure is simple: new edges are added to skip the ε-closure (Figure 5). It is easy to define a well-behaved backward evaluation for this procedure. First, all nodes in the result graph, G_v, exist in the original graph, G_s, so each node in G_v can be traced to G_s. Second, although an edge in G_s may be duplicated in G_v ((E25, d, E56) and (E35, d, E56) in Figure 4(b))[§], each edge in G_v should have a uniquely corresponding edge in G_s. Therefore, adding a new node to G_v corresponds to adding a new node to G_s, and adding a new edge to G_v corresponds to adding a new edge between two corresponding nodes in G_s. Similar correspondence holds for deletions of nodes and edges, and in-place updates of edges.

Next, let us consider Stage 1. One fact worth noting is that after the backward evaluation in Stage 2, the modification to the view in Stage 1 satisfies the *ε-marker preserving property*: (1) No ε-edges are added or deleted, (2) Markers are not added, deleted, or changed and (3) Unreachable parts are not modified. This property is very important in our bidirectionalization, because it not only enforces the nine graph constructors so that they are invertible, but it also makes it easy to bidirectionalize structural recursion because there is a clear correspondence between the input and output graphs.

In the rest of this paper, we will focus on bidirectional graph transformation in Stage 1.

[§] Note that Figure 4(c) does not have this duplication because for this particular graph, it is safe to glue the source and the destination nodes of an ε-edge together. It is unsafe, if and only if, the source has another outgoing edge *and* the destination has another incoming edge. Here, duplication is unavoidable.

4. Traceable Forward Evaluation

An UnCAL expression usually specifies a forward evaluation mapping a graph database (which is just a graph) to a view graph (in Section 2). The main purpose of the present paper is to give *backward evaluation* (*backward semantics*), which specifies how to reflect view updates to the graph database. For this purpose, we have to detect how each node of the view is generated, particularly when it is constructed through connecting input/output markers and removing ε-edges, which are no longer in the view. To make the view more informative, viz., *traceable*, we enrich the original semantics of UnCAL by embedding trace information (like provenance traces (Cheney et al. 2008)) in all nodes of the view that possibly includes ε-edges. In this section, we explain what kind of trace information is embedded in the view, and extend the original semantics for UnCAL expressions to be evaluated into traceable views.

4.1 Traceable Views

A view is obtained by evaluating an UnCAL expression with a database. Every node of the view originates in either a node of the database or a construct in the UnCAL expression, except when the node is generated through a structural recursion with a **rec** construct (in the bulk semantics). Recall that an expression $\mathbf{rec}(\lambda(\$l, \$g).e_1)(e_2)$ is evaluated by binding variables $\$l$ and $\$g$ in e_1 to a part of the evaluation result of e_2. In this case, a node in the view may originate not only in the whole **rec** expression but also a sub-expression in e_2.

A *traceable view* is a view each node of which has information for tracing its origin. The information, called *trace ID*, is defined by

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

where SrcID ranges over identifiers uniquely assigned to all nodes of the database, Pos ranges over code positions in the UnCAL expression, Marker ranges over input/output markers, and Edge stands for $\mathit{TraceID} \times \mathit{Label} \times \mathit{TraceID}$ with a set of labels Label.

We now briefly explain the meaning of each trace ID. Let i be a trace ID of a node u in a traceable view. When i is a node identifier in SrcID, node u originates in the node assigned by i in the database. When i is $\mathsf{Code}\ p\ \&m$ with code position p and input marker $\&m$, node u originates in the subexpression at p in the UnCAL expression. The marker $\&m$ is only required when the subexpression is given by the $\cup$ or **cycle** construct. This is because these constructs yield as many ε-edges as input markers. When i is either $\mathsf{RecN}\ p\ i_0\ \&m$ or $\mathsf{RecE}\ p\ i_0\ (i_1, a, i_2)$, node u is generated through the **rec** construct at the code position p. RecN and RecE stand for what node and edge, respectively, of the argument of the recursion, the node originates in.

Let us explain these cases through an example where the UnCAL expression $a2d\_xc$ in Example 3 is applied to the database G_{src} in Figure 1(a). The traceable view we want can be obtained from the graph G_{view} in Figure 4(a) by assigning trace IDs to all nodes. The trace ID assigned to node 1 in G_{view} is

(RecN 7 1 &) because the node originates in node 1 of G_{src} in $SrcID$, which is used as a part of the argument of the **rec** construct at code position 7 in $a2d\_xc$. The trace ID assigned to node S12 in G_{view} is (RecE 7 (Code 2) $(1, a, 2)$) because the node originates in the a-labeled edge from node 1 to 2 of G_{src} in $Edge$ through the graph constructor {d : _} at code position 2 in the **rec** construct at 7 in $a2d\_xc$. When the argument of the **rec** construct is also a **rec** expression, RecN and RecE in the trace ID are nested like (RecN p (RecE p' ...) ...) and (RecE p (RecE p' ...) (RecN ..., a, RecN ...)).

A traceable view is denoted by a quadruple (V, E, I, O) just like an ordinary UnCAL graph. The only difference is that in traceable views, trace IDs are assigned to all nodes.

4.2 Enriched Forward Semantics

Traceable views can be computed by a simple extension of the original forward semantics of UnCAL so that tracing information is recorded when a node is created. Let e^p denote an UnCAL subexpression e at code position p. We write $\rho(\$x)$ for G when $(\$x \mapsto G) \in \rho$. ρ is naturally used as variable substitution in UnCAL expressions, e.g., $e\rho$ for an expression e. We inductively define the enriched forward semantics $\mathcal{F}[\![e^p]\!]\rho$ for each UnCAL construct of e.

Graph Constructor Expressions. The semantics of graph constructor expressions is straightforward according to the construction in Figure 2. For instance, we have

$$\mathcal{F}[\![\{\}^p]\!]\rho = (\{\mathsf{Code}\ p\}, \emptyset, \{(\&, \mathsf{Code}\ p)\}, \emptyset),$$

which creates a graph having a single node with the trace ID of Code p (indicating the node is constructed by the code at position p), no edges, an input node (the single node itself), and no output nodes. As another example, the semantics for the expression $e_1 \cup e_2$ is defined below to unify two graphs by connecting their input nodes with matching markers using ε-edges:

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

where $\cup^p$ is a union operator for two graphs concerning position p and is defined by

$$\begin{aligned} G_1 \cup^p G_2 = (V \cup V_1 \cup V_2, E \cup E_1 \cup E_2, I, O_1 \cup O_2) \\ \text{where } (V_1, E_1, I_1, O_1) = G_1 \\ (V_2, E_2, I_2, O_2) = G_2 \\ M = \mathsf{inMarker}(G_1) = \mathsf{inMarker}(G_2) \\ V = \{\mathsf{Code}\ p\ \&m \mid \&m \in M\} \\ E = \{(\mathsf{Code}\ p\ \&m, \varepsilon, v) \mid (\&m, v) \in I_1 \cup I_2\} \\ I = \{(\&m, \mathsf{Code}\ p\ \&m) \mid \&m \in M\}. \end{aligned}$$

We omit definitions for other constructor expressions.

Variable. A variable looks up its binding from environment ρ.

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

Condition. The forward semantics of a condition is defined as

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

It first evaluates the conditional expression $l_1 = l_2$, and with the result it evaluates either the **then** branch or the **else** branch.

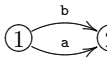
Structural Recursion. The semantics of a structural recursion is given by *bulk semantics* as reviewed in Section 2.3, which can be formally defined by

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

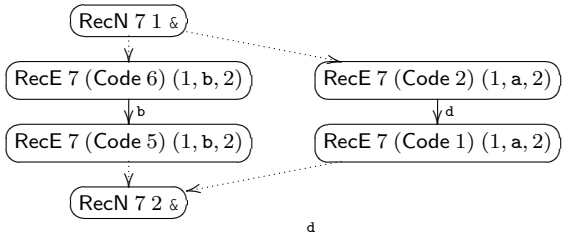
where fwd_eachedge and $\mathsf{compose}_{\mathbf{rec}}$ are defined in Figure 6. Intuitively, fwd_eachedge evaluates the body expression e_{b} at each edge ζ of the argument graph G_{a} obtained by evaluating e_{a} and returns the set of result graphs. Then, $\mathsf{compose}^p_{\mathbf{rec}}$ glues all the graphs together along the structure of G_{a} concerning code position p. Note that $\mathsf{subgraph}(G, \zeta)$ denotes the subgraph to which the edge ζ is pointing in the graph G.

Example 6. We will now illustrate the semantics of **rec** through an example: the structural recursion $a2d\_xc$, which is defined with position information in Example 3, is applied to G_{src} in Figure 1(a), and the traceable view is a graph similar to G_{view} in Figure 4(a).

First, G_{src} is bound to a variable $\$db$. Then, fwd_eachedge generates a set of pairs of an edge and a 'local result' for each edge in G_{src}. The local result is obtained by evaluating the body of **rec** under $\rho = \{\$db \mapsto G_{\mathrm{src}}\} \cup \{\$l \mapsto L, \$g \mapsto G\}$ with the label L of the edge and a subgraph G reachable from the edge. For example, as the local result for edge $(3, \mathsf{a}, 5)$ in G_{src}, edge (Code 2, d, Code 1) with input node Code 2 and output node Code 1 is generated because the subexpression $\{\mathsf{d} : \&^1\}^2$ is used due to $\$l = \mathsf{a}$. The function $\mathsf{compose}^p_{\mathbf{rec}}$ glues all pairs of an edge and a local result after adding RecN or RecE to their nodes. For example, regarding a pair of edge $\zeta = (3, \mathsf{a}, 5)$ and its local result containing edge (Code 2, d, Code 1), the set E_{RecE} contains edge (RecE 7 (Code 2) ζ, d, RecE 7 (Code 1) ζ) where 7 is the code position of the concerned **rec**, while set E_{RecN} contains edge (RecN 7 3 &, ε, RecE 7 (Code 2) ζ) and (RecE 7 (Code 1) ζ, ε, RecN 7 5 &) due to $(\&, \mathsf{Code}\ 2) \in I$ and $(\mathsf{Code}\ 1, \&) \in O$. The former corresponds to the edge from S35 to E35 of G_{view} and the latter corresponds to two edges from 3 to S35 and from E35 to 5 of G_{view}. In this example, E_ε is an empty set since G_{src} has no ε-edges. The sets I_{RecN} and O_{RecN} of input and output nodes are obtained with $I = \{(\&, 1)\}$ and $O = \emptyset$, respectively, which are those of G_{src}. Hence, $I_{\mathsf{RecN}} = \{(\&.\&, \mathsf{RecN}\ 7\ 1\ \&)\}$ and $O_{\mathsf{RecN}} = \emptyset$ because $M = \mathsf{inMarker}(e_{\mathrm{b}}) \cup \mathsf{outMarker}(e_{\mathrm{b}}) = \{\&\}$. Here, "." denotes Skolem function (Buneman et al. 2000) that satisfies $(\&x.\&y).\&z = \&x.(\&y.\&z)$ (associativity) and $\&.\&x = \&x.\& = \&x$ (left and right identity).

More concretely, if the source graph is $s =$ 

, $a2d\_xc(s)$ gives the graph



which is bisimilar to the graph 

. □

5. Backward Evaluation of UnCAL

With traceable views and the ε-marker preserving property (Section 3) on the modification of such views, backward evaluation (in Stage 1) turns out to be simpler for two reasons.

- First, the graph constructors become invertible. For instance, if $G = G_1 \cup G_2$, G is modified to G', but the modification is ε-marker preserving; then, we can follow tracing information, ε-edges, and marker information to *uniquely* decompose G' to G'_1

$$\mathsf{fwd\_eachedge}(G_{(\_,E,\_,\_)}, \rho, e) = \left\{ (\zeta, \mathcal{F}[\![e]\!]\rho_\zeta) \;\middle|\; \zeta \in E,\ \mathsf{label}(\zeta) \neq \varepsilon,\ \rho_\zeta = \rho \cup \{\$l \mapsto \mathsf{label}(\zeta), \$g \mapsto \mathsf{subgraph}(G, \zeta)\} \right\}$$

$$\begin{aligned}
&\mathsf{compose}^p_{\mathbf{rec}}(\mathcal{G}, (V, E, I, O), M) = (V_{\mathsf{RecE}} \cup V_{\mathsf{RecN}}, E_{\mathsf{RecE}} \cup E_{\mathsf{RecN}} \cup E_\varepsilon, I_{\mathsf{RecN}}, O_{\mathsf{RecN}}) \\
&\quad \text{where } V_{\mathsf{RecE}} = \{\mathsf{RecE}\ p\ v\ \zeta \mid (\zeta, (V_\zeta, _, _, _)) \in \mathcal{G}, v \in V_\zeta\} \\
&\qquad E_{\mathsf{RecE}} = \{(\mathsf{RecE}\ p\ u\ \zeta, a, \mathsf{RecE}\ p\ v\ \zeta) \mid (\zeta, (_, E_\zeta, _, _)) \in \mathcal{G}, (u, a, v) \in E_\zeta\} \\
&\qquad V_{\mathsf{RecN}} = \{\mathsf{RecN}\ p\ v\ \&m \mid v \in V, \&m \in M\} \\
&\qquad E_{\mathsf{RecN}} = \{(\mathsf{RecN}\ p\ v\ \&m, \varepsilon, \mathsf{RecE}\ p\ u\ \zeta) \mid \&m \in M, (\zeta = (v, _, _), (_, _, I_\zeta, _)) \in \mathcal{G}, (\&m, u) \in I_\zeta\} \\
&\qquad\qquad \cup \{(\mathsf{RecE}\ p\ u\ \zeta, \varepsilon, \mathsf{RecN}\ p\ v\ \&m) \mid \&m \in M, (\zeta = (_, _, v), (_, _, _, O_\zeta)) \in \mathcal{G}, (u, \&m) \in O_\zeta\} \\
&\qquad E_\varepsilon = \{(\mathsf{RecN}\ p\ v\ \&m, \varepsilon, \mathsf{RecN}\ p\ u\ \&m) \mid (v, \varepsilon, u) \in E, \&m \in M\} \\
&\qquad I_{\mathsf{RecN}} = \{(\&n.\&m, \mathsf{RecN}\ p\ v\ \&m) \mid (\&n, v) \in I, \&m \in M\} \\
&\qquad O_{\mathsf{RecN}} = \{(\mathsf{RecN}\ p\ v\ \&m, \&n.\&m) \mid (v, \&n) \in O, \&m \in M\}
\end{aligned}$$

Figure 6. Core of Forward Semantics of $\mathbf{rec}$ at Code Position p

and G'_2 such that $G'_1 \cup G'_2$ is exactly equivalent to G'. ¶ We will write this decomposition as $\mathsf{decomp}_{G_1 \cup G_2}$ **, and applying it to G' will give (G'_1, G'_2).

- Second, backward evaluation of a structural recursion $\mathbf{rec}(e)$ is reduced to that of its body e (followed by result gluing), because of the bulk semantics of structural recursion.

Backward evaluation greatly depends on what updates are allowed on the view. We allow the following three general updates on our edge-labeled graphs: (1) in-place updates as modification of edge labels, (2) deletion of edges, and (3) insertion of edges or a subgraph rooted at a node. And we accept a sequence of these updates on the view and reflect them to the source. In the rest of this section, we shall explain the respective backward evaluation for these updates on views.

5.1 Reflection of In-place Updates

In this section, we formally define backward semantics for UnCAL, where only in-place updates are considered.

Recall that backward semantics $\mathcal{B}[\![e]\!](\rho, G')$ is used to compute a new environment from the original input environment ρ and the modified view G'. Like forward semantics, backward semantics can be defined inductively over the construction of expression.

5.1.1 Backward Evaluation of Simple Expressions

Graph Constructor Expressions. Since each constructor is revertible and is associated with a decomposition function, we can decompose the views of constructor expressions so as to define the backward semantics *inductively*. For example, we have

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

Unlike Foster et al. (2005), we have variable binding, and therefore multiple environments produced by backward evaluation of the operands are merged by $\uplus_\rho$ defined below, using an approach similar to that in Liu et al. (2007), which deals with variable bindings.

$$\begin{aligned}
&(\rho_1 \uplus_\rho \rho_2) \\
&\quad = \left\{ (\$v \mapsto \mathsf{mg}(G, G_1, G_2)) \;\middle|\; \begin{array}{lcl} (\$v \mapsto G_1) & \in & \rho_1, \\ (\$v \mapsto G) & \in & \rho, \\ (\$v \mapsto G_2) & \in & \rho_2 \end{array} \right\} \\
&\quad \text{where } \mathsf{mg}(G, G_1, G_2) = \begin{cases} G_1 & \text{if } G_2 = G \vee G_1 = G_2 \\ G_2 & \text{if } G_1 = G \\ \textsc{Fail} & \text{otherwise} \end{cases}
\end{aligned}$$

$\uplus_\rho$ unifies each binding by mg. If only the binding on the left hand side is modified ($G_2 = G$), or both are consistently updated ($G_1 = G_2$), then the binding on the left is adopted, and vice versa. If both are updated to different values, it fails, leading to the failure of the entire backward evaluation. Label variable bindings are treated similarly.

We have omited the definitions for other constructor expressions, which can be defined similarly.

Variable. A variable simply updates its binding as

$$\mathcal{B}[\![\$v]\!](\rho, G') = \rho[\$v \leftarrow G'].$$

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

$$\begin{aligned}
&\mathcal{B}[\![\mathbf{if}\ l_1 = l_2\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!](\rho, G') \\
&\quad = \begin{cases} \rho'_1 & \text{if } l_1\rho = l_2\rho \wedge l_1\rho'_1 = l_2\rho'_1 \\ \rho'_2 & \text{if } l_1\rho \neq l_2\rho \wedge l_1\rho'_2 \neq l_2\rho'_2 \\ \textsc{Fail} & \text{otherwise} \end{cases} \\
&\quad \text{where } \rho'_1 = \mathcal{B}[\![e_1]\!](\rho, G') \\
&\qquad\quad \rho'_2 = \mathcal{B}[\![e_2]\!](\rho, G'),
\end{aligned}$$

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 ensure that $l_1 = l_2$ does not change after backward evaluation.

5.1.2 Backward Evaluation of Structural Recursion

Due to the traceable bulk forward evaluation of structural recursion $\mathbf{rec}$ and the ε-marker preserving property that retains similarity in shape between input and output graphs, backward semantics can easily be defined as

$$\begin{aligned}
&\mathcal{B}[\![\mathbf{rec}(\lambda(\$l, \$g).\ e_\mathrm{b})(e_\mathrm{a})]\!](\rho, G') \\
&\quad = \mathsf{merge}(\rho, e_\mathrm{a}, E_\mathrm{a}, \\
&\qquad\quad \mathsf{bwd_eachedge}(G_\mathrm{a}, \rho, e_\mathrm{b}, \mathsf{decomp}_{\mathbf{rec}}(G', E_\mathrm{a}))) \\
&\quad \text{where } G_\mathrm{a} = (_, E_\mathrm{a}, _, _) = \mathcal{F}[\![e_\mathrm{a}]\!]\rho
\end{aligned}$$

This definition is easy to understand if we note duality with the definition of its forward semantics. Backward semantics first decomposes through $\mathsf{decomp}_{\mathbf{rec}}$ the modified result graph G' into pieces of graphs, which is intuitively an inverse operation of $\mathsf{compose}_{\mathbf{rec}}$.

¶ The exact equivalence of two graphs $G_{1(V_1,E_1,I_1,O_1)}$ and $G_{2(V_2,E_2,I_2,O_2)}$, is defined by $V_1 = V_2 \wedge E_1 = E_2 \wedge I_1 = I_2 \wedge O_1 = O_2$.

** It would be more precise to write it as $\mathsf{decomp}_{G_1,\cup,G_2}$ in that the decomposition depends on three arguments.

$$\mathsf{decomp}_{\mathbf{rec}}((V',E',I',O'),E_{\mathrm{a}}) = \left\{ (\zeta,(V'_\zeta,E'_\zeta,I'_\zeta,O'_\zeta)) \;\middle|\; \begin{array}{l} \zeta \in E_{\mathrm{a}}, \mathit{label}(\zeta) \neq \varepsilon, \\ V'_\zeta = \{w \mid (\mathsf{RecE}\ p\ w\ \zeta) \in V'\}, \\ E'_\zeta = \{(w_1,a,\ w_2) \mid (\mathsf{RecE}\ p\ w_1\ \zeta, a, \mathsf{RecE}\ p\ w_2\ \zeta) \in E'\}, \\ I'_\zeta = \{(\&m, w) \mid (\mathsf{RecN}\ p\ v\ \&m, \varepsilon, \mathsf{RecE}\ p\ w\ \zeta) \in E'\}, \\ O'_\zeta = \{(w, \&m) \mid (\mathsf{RecE}\ p\ w\ \zeta, \varepsilon, \mathsf{RecN}\ p\ v\ \&m) \in E'\} \end{array} \right\}$$

$$\mathsf{bwd\_eachedge}(G,\rho,e,\mathcal{G}') = \left\{ (\zeta, \mathcal{B}[\![e]\!](\rho_\zeta, G'_\zeta)) \;\middle|\; (\zeta,G'_\zeta) \in \mathcal{G}',\ \rho_\zeta = \rho \cup \{\$l \mapsto \mathsf{label}(\zeta), \$g \mapsto \mathsf{subgraph}(G,\zeta)\} \right\}$$

$$\mathsf{merge}(\rho, e_{\mathrm{a}}, E_{\mathrm{a}}, \mathcal{R}) = \mathcal{B}[\![e_{\mathrm{a}}]\!](\rho, G'_{\mathrm{a}}) \uplus_\rho \biguplus \left\{ \rho'_\zeta \setminus \{\$l \mapsto \_\} \setminus \{\$g \mapsto \_\} \;\middle|\; (\zeta,\rho'_\zeta) \in \mathcal{R} \right\}$$

$$\begin{array}{lll} \text{where } G'_{\mathrm{a}} & = \left(\bigcup V''_\zeta,\ E_{\mathrm{eps}} \cup \bigcup E''_\zeta,\ I_{\mathrm{a}},\ O_{\mathrm{a}}\right) & \\ E_{\mathrm{eps}} & = \{(u,\varepsilon,v) \mid (u,\varepsilon,v) \in E_{\mathrm{a}}\} & \\ (V''_\zeta, E''_\zeta) & = \left(V'_\zeta \cup \{u\}, E'_\zeta \cup \{(u, \rho'_\zeta(\$l), I'_\zeta(\&))\}\right) & \text{for each } (\zeta,\rho'_\zeta) \in \mathcal{R}, \text{ letting } (u,\_,\_) = \zeta \text{ and } (V'_\zeta,E'_\zeta,I'_\zeta,O'_\zeta) = \rho'_\zeta(\$g) \end{array}$$

Figure 7. Core of Backward Semantics of **rec** at Code Position p

For every non-ε edge $\zeta \in E_{\mathrm{a}}$ in the source argument graph, the decomposition extracts (possibly modified) subpart G'_ζ of G', which originates at the result G_ζ of the forward computation on the edge. Then, in bwd_eachedge, we carry out backward computation of the body expression e_{b} on each edge and compute the updated environment ρ'_ζ. Finally, these environments are merged into the updated environment ρ' of the whole expression. The merge function does two pieces of work. First, by combining the information $\rho'_\zeta(\$l)$ and $\rho'_\zeta(\$g)$ from the updated environments (and ε-edges existing in the edges E_{a} of the source argument graph), it computes the modified argument graph G'_{a}. Then, we inductively carry out backward evaluation on the argument expression e_{a} to obtain another updated environment ρ'_{a}. This ρ'_{a} and all ρ'_ζs are merged into ρ'.

Let us explain in more detail the definition of $\mathsf{decomp}_{\mathbf{rec}}$, which is the key point of the backward evaluation.

The function first extracts from result graph G' nodes V'_ζ and edges E'_ζ that belong to each edge ζ by matching trace ID $\mathsf{RecE}\ p\ \_\ \zeta$. Note that if there are nodes that have been freshly inserted into the view, we also require these nodes to have this structure, so that these nodes are also passed to the backward evaluation of the recursion body. Input and output nodes with marker $\&m$ are recovered by selecting those pointed from/to "hub" nodes having structure $\mathsf{RecN}\ \_\ \_\ \&m$. Top-level constructors of trace ID are erased so that we can inductively compute the backward image from the body expression.

Example 7. Recall the simple example in Example 3 where the source is $s = \textcircled{1} \xrightarrow{\mathtt{b}} \textcircled{2}$ (with a parallel edge labeled a), and $a2d\_xc(s)$ gives the graph G. If the graph G is modified to G' where the edge label b is updated to x, then $\mathcal{B}[\![a2d\_xc]\!](\{\$db \mapsto s\}, G')$ returns binding $\{\$db \mapsto s'\}$ where $s' = \textcircled{1} \xrightarrow{\mathtt{x}} \textcircled{2}$ (with a parallel edge labeled a). Therefore, the in-place update of the change on the view graph is reflected to the source. □

Lemma 2 (Well-behavedness for In-place Updates)**.** *If output graphs are modified by in-place updates on edges, then for any expression e, the two evaluations $\mathcal{F}[\![e]\!]\_$ and $\mathcal{B}[\![e]\!](\_,\_)$ form a well-behaved bidirectional transformation, if they succeed.*

Proof. This statement can be proved by induction on the structure of e. For the base case where e is a variable, it clearly holds. Considering the inductive case, (1) if e is a constructor expression, it holds because each constructor is revertible within our context, (2) if e is a condition, its backward evaluation is reduced to that on either its true branch or its false branch, so the statement holds by induction, and (3) if e is a structural recursion, by bulk semantics, its backward computation is reduced to its body expression, so the statement holds by induction. □

5.2 Reflection of Deletion

Deletion in a view is reflected as deletion of the corresponding part in the source by using trace IDs. Suppose we want to delete the edge labeled d in the view of Example 7. Since both endpoints of the edge have trace IDs of the form $\mathsf{RecE}\ 7\ \_\ (1,\mathtt{a},2)$, we can see that the selected edge has been generated due to the existence of the source edge $(1,\mathtt{a},2)$, which is the "corresponding part" to be deleted in the source.

In general, for a labeled edge $\zeta = (u,a,v)$ with $a \neq \varepsilon$, its corresponding edge $\mathsf{corr}(\zeta)$ is defined as:

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

Here, FAIL means failure on finding the corresponding edge. The first case means that if the edge ζ is a copy of an edge in the source, then ζ itself is the corresponding edge. The second and the third cases are for when ζ is a result of some structural recursion. According to the forward semantics of **rec** in Figure 6, the non-ε edge ζ must have the form $(\mathsf{RecE}\ p\ u\ \zeta', a, \mathsf{RecE}\ p\ v\ \zeta')$ for some p,u,v, and another non-ε edge ζ'. This means that ζ consists of an edge (u,a,v) originating from an evaluation of a recursion-body at ζ'. Hence, for this case, we first recursively trace the corresponding source of (u,a,v), and if this fails, then try that of ζ'. In other cases, corr fails to find the corresponding source, because it must be the case that u has a trace ID of the form $\mathsf{Code}\ \_$, meaning that the edge is not derived from the source but from an UnCAL expression.

Let $\$db$ be the source graph, G_{view} be the view produced by $\mathcal{F}[\![e]\!]\rho$ from a forward computation of expression e with environment ρ, and G'_{view} be a graph from G_{view} with a set of edges $D_{\mathrm{out}} = \{\zeta_1,\ldots,\zeta_n\}$ removed. Our backward evaluation $\mathcal{B}[\![e]\!](\rho, G'_{\mathrm{view}})$ consists of the following three steps.

1. Compute the set of source edges
$$D_{\mathrm{in}} = \{\mathsf{corr}(\zeta_i) \mid \zeta_i \text{ is not an } \varepsilon\text{-edge}\}.$$
2. If $\textsc{Fail} \in D_{\mathrm{in}}$, backward evaluation fails. If it is obtained successfully without failure, compute
$$G'_{\mathrm{src}} = \rho(\$db) - D_{\mathrm{in}},$$
where $G - E$ denotes removal of the edges in the set E from graph G.
3. Return $\rho' = \rho[\$db \leftarrow G'_{\mathrm{src}}]$ as the result if $\mathcal{F}[\![e]\!]\rho' = G'_{\mathrm{view}}$, and fail otherwise.

Lemma 3 (Well-behavedness for Deletion)**.** *If output graphs are modified by edge deletion, then for any expression e, the two evaluations $\mathcal{F}[\![e]\!]_{\_}$ and $\mathcal{B}[\![e]\!](\_,\_)$ form a well-behaved bidirectional transformation, if they succeed.*

Proof. The (GETPUT) property is clear because of the fact that $D_{\mathrm{in}} = \emptyset$ if $D_{\mathrm{out}} = \emptyset$. For the (WPUTGET) property, it holds because the third step actually does this check. □

5.3 Reflection of Insertion

Reflection of insertion is much more complicated than that of inplace-updating and deletion. This is because there are no corresponding edges in the source for the freshly inserted edges in the view, which requires us not only to *create* new information but also to add it to a proper location in the source graph.

Our idea was to use the Universal Resolving Algorithm (URA) (Abramov and Glück 2002), a powerful method of inversion computation, to derive a right inverse of the forward evaluation, and use the distributive property of structural recursion

$$\mathbf{rec}(e)(\$g_1 \cup \$g_2) = \mathbf{rec}(e)(\$g_1) \cup \mathbf{rec}(e)(\$g_2)$$

to properly reflect insertion to the source.

In this section, we shall give our algorithm for this reflection, before we highlight how URA can be used to derive the right inverse.

5.3.1 Insertion Reflection with Right Inverse

We assume the monotonicity of insertion in that an insertion on the view is translated to an insertion on the source rather than other updating operations. The monotonicity comes from the absence of `isEmpty` (Buneman et al. 2000) in our core UnCAL. We only consider insertion on the view graph produced by forward computation of a variable expression or a structural recursion. For the case of a variable, this reflection is done in the same way as in Section 5.1.1. Insertion for structural recursion, the basic computation unit in UnCAL, needs to be carefully designed. In the following, we will focus on structural recursion, omitting other cases for simplicity.

Before giving our reflection algorithm, we should clarify the meaning of right inverse. In general, a function h is said to be a right inverse of f if for any x in the range of f, $f(h(x)) = x$ holds. Within our context, for an expression e and a graph G, $\mathcal{F}^{\circ}[\![e]\!](G)$ is said to be a right inverse computation if it returns ρ' such that $\mathcal{F}[\![e]\!]\rho' = G$.

Now, we will return to our reflection algorithm. Let G_{src} be the source graph, $G_{\mathrm{view}} = \mathcal{F}[\![\mathbf{rec}(e)(\$db)]\!]\rho$, where $\rho = \{\$db \mapsto G_{\mathrm{src}}\}$, and G'_{view} be a graph from G with new edges inserted. Notice that it is sufficient to consider $\$db$ as the argument of $\mathbf{rec}$, because $\$db$ can be bound to other expression. Our backward evaluation $\mathcal{B}[\![\mathbf{rec}(e)(\$db)]\!](\rho, G'_{\mathrm{view}})$ returns ρ as the result if there are no new edges inserted in G_{view}; otherwise, it does the following:

1. Extract the inserted subgraph G' from G'_{view} such that
$$G'_{\mathrm{view}} = G_{\mathrm{view}} \cup G'.$$
2. Compute with right inverse computation:
$$\rho'_1 = \mathcal{F}^{\circ}[\![\mathbf{rec}(e)(\$db)]\!](G').$$
3. Return $\rho'_2 = \{\$db \mapsto G_{\mathrm{src}} \cup \rho'_1(\$db)\}$ as the result.

The first step of extraction is possible provided that insertion happens at the root node[††]. The second step of right inverse computation will be explained in Section 5.3.3. The last step is to update the binding of $\$db$ and return this environment as our result. The following lemma shows the correctness of the algorithm.

[††] Insertions to non-root positions are possible due to bulk semantics that allows similar treatment for every node.

Lemma 4 (Well-behavedness for Insertion)**.** *If output graphs are modified by edge insertion, then for a structural recursion of the form $\mathbf{rec}(e)(\$db)$ where e contains no free variables, then two evaluations $\mathcal{F}[\![e]\!]_{\_}$ and $\mathcal{B}[\![e]\!](\_,\_)$ form a well-behaved bidirectional transformation, if they succeed.*

Proof. First, the (GETPUT) property clearly holds because ρ is returned when no insertions occur. Next, we prove the (WPUTGET) property by using the following calculation.

$$\begin{array}{ll} & \mathcal{F}[\![\mathbf{rec}(e)(\$db)]\!]\rho'_2 \\ = & \quad \{ \text{ partial application } \} \\ & \mathcal{F}[\![\mathbf{rec}(e)(\rho'_2(\$db))]\!]\rho'_2 \\ = & \quad \{ \text{ def. of } \rho'_2 \ \} \\ & \mathcal{F}[\![\mathbf{rec}(e)(G_{\mathrm{src}} \cup \rho'_1(\$db))]\!]\rho'_2 \\ = & \quad \{ \text{ structural recursion property } \} \\ & \mathcal{F}[\![\mathbf{rec}(e)(G_{\mathrm{src}}) \cup \mathbf{rec}(e)(\rho'_1(\$db)))]\!]\rho'_2 \\ = & \quad \{ \text{ forward evaluation } \} \\ & \mathcal{F}[\![\mathbf{rec}(e)(G_{\mathrm{src}})]\!]\rho'_2 \cup \mathcal{F}[\![\mathbf{rec}(e)(\rho'_1(\$db)))]\!]\rho'_2 \\ = & \quad \{ \ e \text{ does not contain free variable } \} \\ & G_{\mathrm{view}} \cup \mathcal{F}[\![\mathbf{rec}(e)(\$db)]\!]\rho'_1 \\ = & \quad \{ \text{ right inversion } \} \\ & G_{\mathrm{view}} \cup G' \end{array}$$ □

It is worth noting that we have simplified our discussion in both the above algorithm and lemma by making it a requirement that e in $\mathbf{rec}(e)(\$db)$ does not contain any free variables. With this requirement, our forward and backward evaluation satisfies the stronger (PUTGET) property. In fact, it is acceptable to relax this condition by allowing e to contain other free variables and the initial ρ contains binding of other variables. Then, right inversion will produce ρ'_1 that will be used to update all variable bindings in addition to $\$db$. In this case, $\mathcal{F}[\![\mathbf{rec}(e)(G_{\mathrm{src}})]\!]\rho'_1$ may produce a graph that is different from the original view G_{view}. In any case, this different graph will not have an additional effect on the source when we apply backward evaluation to this new graph. Therefore, (WPUTGET) always holds.

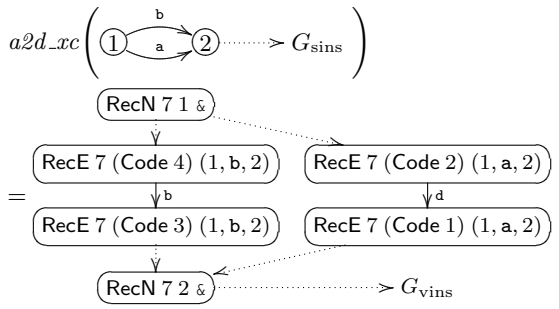
With this idea, we shall propose an algorithm in which (PUTGET) property is satisfied without any additional requirements. The idea is to utilize the Trace ID information, as will be discussed later.

5.3.2 Improving Insertion Reflection

The method above satisfies the (PUTGET) property only if the variables of e are disjoint from the variables bound in the initial environment ρ. However, in general, since a transformation may have multiple variable references, more effort is required to achieve the (PUTGET) property. We tackle the problem by first locating where we insert a graph by using trace IDs, and then applying the URA algorithm (to be described later) to find what graph should be inserted.

Consider the transformation $a2d\_xc$ and the view in Example 6. Suppose we want to insert a graph G_{vins} rooted at the view node v = RecN 7 2 &. Where should some graph be inserted into the source to reflect this insertion? The answer is that we *must* insert a graph rooted at the source node 2 because there would be no edge from v in the view unless there were an edge from 2 in the source according to the bulk semantics of structural recursion. Now, our next task is to find what graph should be inserted under the source

node 2. That is, we hope to find G_{sins} such that the following holds.



URA can help us to find such G_{sins} for G_{vins}. For example, if G_{vins} is $\{\mathtt{b} : \{\}\}$, then URA returns $G_{\text{sins}} = \{\mathtt{b} : \{\}\}$. If G_{vins} is $\{\mathtt{d} : \{\}\}$, then URA returns one of the possibilities, $G_{\text{sins}} = \{\mathtt{a} : \{\}\}$ or $G_{\text{sins}} = \{\mathtt{d} : \{\}\}$, depending on the search method used in URA. According to the soundness and the completeness of URA, the reflection by URA is always correct in the sense that (PUTGET) holds, and moreover URA always returns a G_{sins} if such G_{sins} exists. Of these, soundness is the key to insertion reflection satisfying (PUTGET) for general UnCAL transformations.

In summary, our insertion-reflection algorithm is as follows.

1. Let v be a node under which we want to insert a graph G_{vins}.
2. By using the tr function in Figure 8, we find the source node $u = \mathtt{tr}(v)$ under which we insert a graph to reflect the insertion.
3. Let G'_{view} be a graph obtained from the view by adding ε-edge from v to G_{vins}.
4. We find a graph G_{sins} connected from u by an ε-edge, by applying URA for G'_{view}.
5. We return a graph G'_{src} obtained from the source by adding an ε-edge from u to G_{sins}.

The soundness of the insertion-reflection algorithm is directly derived from the soundness of URA.

Lemma 5 (Soundness of Insertion)**.** *Our insertion-reflection algorithm satisfies* (PUTGET).

Note that we use URA for G'_{view} instead of G_{vins}. Thus, URA rejects any insertion of G_{sins} that violates (PUTGET).

In addition, our insertion-reflection algorithm is *complete* in the sense that, if there exist some source insertions to reflect the view insertion under some conditions, the algorithm will find one of them.

Lemma 6 (Completeness of Insertion)**.** *Let v be a node such that $\mathtt{tr}(v) \neq$ FAIL. For any source graph G, we can insert any graph into its view if there exists a source insertion that reflects the view insertion and v still occurs in the view of the insertion-reflected source.*

Recall that we only consider insertion on the view graph produced by forward computation of a variable expression or a structural recursion, which is expressed by $\mathtt{tr}(v) \neq$ FAIL. This lemma can be proved using the property of trace IDs stating that, to insert a graph rooted at view node v, we must insert a graph rooted at source node $\mathtt{tr}(v)$. By induction on the trace ID of v, we can show that, if there is an edge from v, it must be the case that there is an edge from $\mathtt{tr}(v)$, which is implied by the property of trace IDs. Note that G_{vins} has no edge to the original view. However, this is not a restriction since if there is a crossing edge pointing to a subgraph of the original view, we can duplicate the subgraph and integrate it to G_{vins} so that the edge can be eliminated.

$$\begin{array}{ll} \mathtt{tr}(\mathit{SrcID}) = \mathit{SrcID} & \mathtt{tr}(\mathsf{RecN}\ \_\ v\ \_) = \mathtt{tr}(v) \\ \mathtt{tr}(\mathsf{Code}\ \_\ \_) = \text{FAIL} & \mathtt{tr}(\mathsf{RecE}\ \_\ v\ \_) = \mathtt{tr}(v) \end{array}$$

Figure 8. Tracing Node ID

5.3.3 Right Inverse Computation by URA

Recall that the right inverse computation of an expression e is to take a graph G_{view} and return a ρ such that $\mathcal{F}[\![e]\!]\rho = G_{\text{view}}$. We adopt the *universal resolving algorithm* (URA) (Abramov and Glück 2002), a powerful and general inversion mechanism, to compute ρ. The basic idea behind URA is to search on a *perfect process tree* (Glück and Klimov 1993), which represents all possible computations of an expression, and to find a computation path that produced the result.

Our right inverse computation consists of three steps.

1. It lazily enumerates possible evaluation paths by symbolic computation called *needed narrowing* (Antoy et al. 1994)[‡‡].
2. From the generated evaluation paths, it constructs a table of input/output pairs of computations.
3. If there is a pair in the table whose output is G_{view}, it generates a substitution (environment) from the path and returns it as the result.

Example 8. As a simple example, let us see how we find ρ such that

$$\mathcal{F}[\![a2d\_xc(\$x)]\!]\rho = G_{\text{view}}$$

where $G_{\text{view}} = \{\mathtt{d} : \{\}\}$. We search ρ by symbolic evaluation of $a2d\_xc(\$x)$. To evaluate $a2d\_xc(\$x)$, we unfold $\$x$ and recursively evaluate $a2d\_xc$, i.e., a structural recursion. There are many ways to instantiate $\$x$ such as

$$\$x \mapsto \{\}, \$x \mapsto \{\$l_1 : \$x_1\}, \$x \mapsto \{\$l_1 : \$x_1, \$l_2 : \$x_2\}.$$

If we choose $\$x \mapsto \{\}$, the computation finishes, yielding a table consisting of an input/output pair $(\{\}, \{\})$. Since this table does not contain a pair whose output is G_{view}, we continue searching. Assume that we choose $\$x \mapsto \{\$l_1 : \$x_1\}$. Then $a2d\_xc(\$x)$ is unfolded to $(\mathbf{if}\ \$l_1 = \mathtt{a}\ \mathbf{then}\ \{\mathtt{d} : \&\}\ \mathbf{else}\ (\mathbf{if}\ \$l_1 = \mathtt{c}\ \mathbf{then}\ \{\varepsilon : \&\}\ \mathbf{else}\ \{\$l_1 : \&\}))\ @\ a2d\_xc(\$x_1)$. As evaluation gets stuck here because of a free variable $\$l_1$ in the **if** condition, we find a suitable $\$l_1$ to resume the evaluation. If we choose $\$l_1 \mapsto \mathtt{a}$, then the expression is reduced to $\{\mathtt{d} : \&\}\ @\ a2d\_xc(\$x_1)$ and input/output pair $(\{\mathtt{a} : \{\}\}, \{\mathtt{d} : \{\}\})$ is obtained by choosing $\$x_1 \mapsto \{\}$. Since $G_{\text{view}} = \{\mathtt{d} : \{\}\}$, we gather all bindings along this computation and return the following environment as the result.

$$\{\$x \mapsto \{\mathtt{a} : \{\}\}\}$$

Figure 9 shows part of a perfect process tree in our right-inverse computation: the left is the tree and the right is a table of a pair of input/output graph templates (it is more general than a pair of input/output graph instances, as we discussed above). Note this tree is a variant of SLD-resolution trees (Glück and Sørensen 1994). □

To use URA effectively for our right inverse computation of UnCAL, we define a *small-step semantics* for UnCAL such that a perfect process tree can be constructed though these small steps. The only non-standard feature of this semantics is that we use memoization to avoid infinite loops probably caused by cycles in the source graph (See Appendix C for details). In addition, we provide a Dijkstra-searching strategy to enumerate all the possible

[‡‡] The same notion is called *driving* (Glück and Klimov 1993; Glück and Sørensen 1994) in (Abramov and Glück 2002).

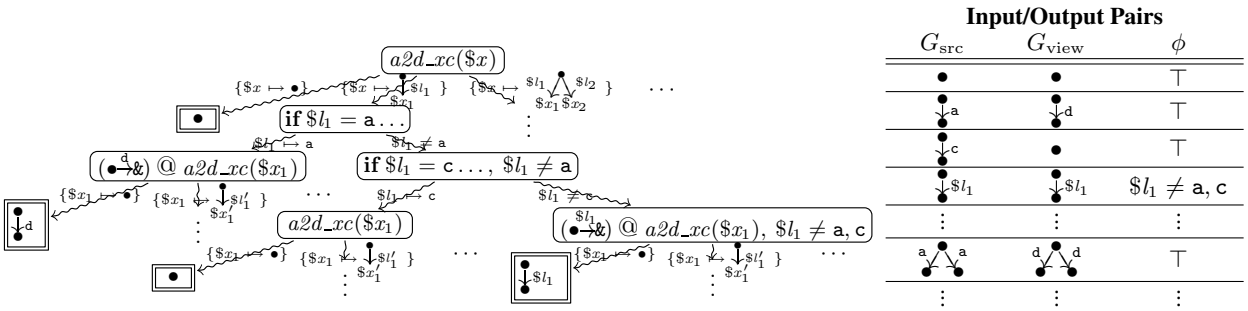


Figure 9. URA for $a2d\_xc$ and Enumerated Input/Output Pairs with Constraints (nodes without branching have been contracted)

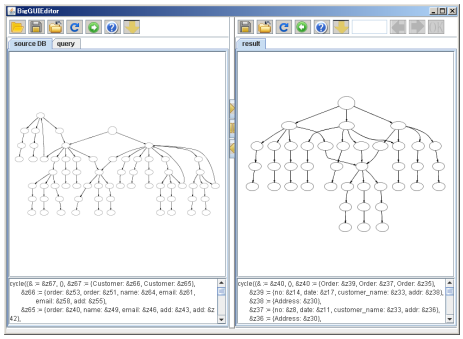


Figure 10. GUI of our Bidirectional Graph Transformation System (left is a source and right is a view, and UnCAL transformation can be run forward and backward)

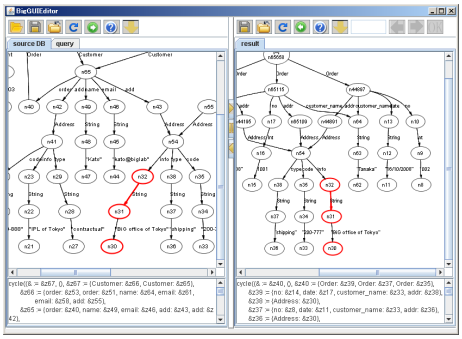


Figure 11. Trace Information Display (one can choose set of edges in either source or view and correspondence can be shown in both according to our trace IDs)

evaluation paths so that a solution can always be found if one exists. The two heuristics we use to design the cost function are:

- We use a (weighted) size of graphs (to be inserted into the source) as a cost function in the Dijkstra-search.
- For the weighted size, the depth (the length of the path) has more weight than the width (the number of paths). This strategy works nicely for *consecutive* in Example 4.

Moreover, we show that a suitable binding to continue evaluation of conditional expressions can easily be found for our core UnCAL, because the conditional part of a conditional expression is in the simple form of $a_1 = a_2$.

6. Implementation and Experiments

The prototype system has been implemented and is available on our BiG project Website. It has a GUI for users to modify source and view graphs, execute UnQL queries (which can be automatically transformed to UnCAL) or UnCAL programs both forward and backward, and see tracing information between the source and view graphs. Figures 10 and 11 have two snapshots of the system. In addition to all the examples in Buneman et al. (2000) and in this paper, we have tested three non-trivial examples, demonstrating its usefulness in software engineering and database management.

- *Customer2Order*: A case study in the textbook on model-driven software development (Pastor and Molina 2007).
- *PIM2PSM*: A typical example of transforming a platform independent object model to a platform specific object model.
- *Class2RDB*; A non-trivial benchmark application for testing the power of model transformation languages (Bezivin et al. 2005).

All of these have demonstrated the effectiveness of our approach in practical applications.

In our implementation, we carefully treat ε-edges and unreachable parts introduced during operations related to markers, and retrieval of edges or nodes of interest, which greatly affect performance. Poor treatment would hinder large-scale UnQL queries to evaluate in bidirectional mode[§§] in a reasonable amount of time. Speed-up of several orders of magnitude has been achieved since our initial implementation due to the above and the following optimizations.

Reduction in number of ε-edges As mentioned in the UnQL paper (Buneman et al. 2000), ε-edges are generously generated during evaluation, especially in **rec**. This slows the evaluation process due to the increase in input size. Removing ε-edges during evaluation has no harm on forward semantics because of bisimulation equivalence. However, since ε-edges play an important role in backward evaluation, they are not freely omitted in our bidirectional settings. Moreover, a straightforward implementation of the removal algorithm (Buneman et al. 2000) may introduce additional edges, which may harm backward evaluation. Toward prudently removing ε-edges that are suitable for backward evaluation, our ε-removal algorithm glues source and destination nodes of ε as long as bisimulation equivalence is not violated.

Pruning of unreachable nodes @ and **rec** may leave unreachable nodes if some input and output nodes are left unconnected due to mismatched markers. This mismatch happens typically during projection of graph components $\&z_i$ by idiom $\&z_i$ @ G and in the case of @ in the definition of **rec** in $\mathbf{rec}(\lambda(\$l, \$g).e_{\mathrm{b}})(\{a : G\}) =$

[§§] Note that we preserve every result of forward computation in the bidirectional mode.

$e_{\rm b}(a, G)$ @ $\mathbf{rec}(e_{\rm b})(G)$. A typical use of $\mathbf{rec}$ includes $e_{\rm b}$ with no output marker, where actual recursion below the first level from the input nodes — subgraphs that are not reachable by traversing only one edge — is not necessary because the @ does not connect the first and second arguments due to the lack of matching markers. This opens up optimization opportunities, i.e., not to evaluate $e_{\rm b}$ for the subgraphs below. The improvement in performance was significant for forward evaluation.

Optimization by fusion transformation Note that the backward evaluation of $\mathbf{rec}(e_1)(\mathbf{rec}(e_2)(e_3))$, a composition of structural recursions, requires to generate intermediate result of backward transformation, which is very expensive. This can be avoided by fusing the two structural recursions into one. We have implemented this based on the fusion rule (Buneman et al. 2000): if $e_1(a, G)$ does not depend on G then $\mathbf{rec}(e_1)(\mathbf{rec}(e_2)(e_3)) = \mathbf{rec}(\mathbf{rec}(e_1) \circ e_2)(e_3)$. With auxiliary rewriting rules such as e_1 @ $e_2 = e_1$ for e_1 that produces no output nodes, 30% and 50% of CPU time reductions are respectively achieved for forward and backward execution in Customer2Order composed with selection, 30% and 65% reductions for simpler examples that appeared in the evaluation for unidirectional transformation (Hidaka et al. 2009). These experiments are for in-place updates, but similar reduction could be achieved for other updates.

7. Related Work

Bidirectional transformation has been discussed as view updating problem in the database community. Bancilhon and Spyratos (1981) proposed a general approach to the view updating problem. They introduced an elegant solution based on the concept of a constant complement view that captures the information in the view but not in the original database. Their idea was not only applied to relational databases (Hegner 1990; Lechtenbörger and Vossen 2003) but also to tree structures (Matsuda et al. 2007). Constant complement views satisfy very strong bidirectional properties at the sacrifice of the number of reflectable updates. Although such strong properties are nice for some applications (Hegner 1990), they are too strong for our purpose, i.e., model transformation in software engineering. Recent work by Fegaras (2010) propagates updates on XML views created from relational databases. It supports duplicates but detects view side effects at both compile and run time. Staworko et al. (2010) proposed an update propagation for XML views in native XML databases. Their scenarios are very simple in that view definitions are projection of the source, and update operations are restricted to the insertion and deletion of nodes.

In the area of programming languages, view updating has been studied as *bidirectional transformation*. Foster et al. (2005) proposed the first linguistic approach to solving this problem. They developed some domain specific languages to support the development of bidirectional transformation on strings and trees. Bohannon et al. (2006) applied these techniques to relational databases, making use of functional dependencies in relations to correctly propagate updates. However, their approach is limited to strings, trees and relations, and is difficult to apply to graph transformation due to graph-specific features such as circularity and sharing.

Within the context of software engineering, there has been several works on bidirectional model (graph) transformation (Ehrig et al. 2005; Jouault and Kurtev 2005; OMG 2005; Schürr and Klar 2008; Stevens 2007), which can deal with kinds of graph structures. However, they lack a clear formal bidirectional semantics and there has not yet been any powerful method of bidirectionalization that can be used to automatically derive backward model transformations from forward model transformations, so that both transformations can form a consistent bidirectional model transformation.

The concept of structural recursion is not new and has been studied in both the database (Breazu-Tannen et al. 1991) and the functional programming communities (Sheard and Fegaras 1993). However, most of these have focused on structural recursion over lists or trees instead of graphs. Examples include the higher order function *fold* (Sheard and Fegaras 1993) in ML and Haskell, and the generic computation pattern called *catamorphism* in programming algebras (Bird and de Moor 1996). UnCAL (Buneman et al. 2000) demonstrates that the idea of structural recursion can be extended to graphs, but the original focus was on the optimization of query fusion rather than bidirectionalization.

Our work was greatly inspired by interesting work on efficient graph querying (Buneman et al. 2000; Sheng et al. 1999). Unlike trees, graphs involve subtle issues on their representation and equivalence. The use of bisimulation and structural recursion in (Buneman et al. 2000) opens a new way of building a framework for both declarative and efficient graph querying with high modularity and composability. This motivated us to extend the framework from graph querying to graph transformation and apply it to model transformation (Hidaka et al. 2009). This work is a further step in this direction to extend it from unidirectional model transformation to bidirectional model transformation.

8. Concluding Remarks

This paper reports our first attempt toward solving the challenging problem of bidirectional transformation on graphs. We show that structural recursion on graphs and its unique bulk semantics play an important role not only in query optimization, which has been recognized in the database community, but also in automatic derivation of backward evaluation, which has not been recognized thus far. As far as we are aware, the bidirectional semantics of UnCAL proposed in this paper is the first complete language-based framework for general graph transformations.

Future work includes extending the framework from unordered graphs to ordered graphs, introducing graph schemas to provide structural information for more efficient bidirectional computation, an efficient algorithm for checking updatability, and more practical applications of the system for bidirectional model transformation in software engineering.

Acknowledgments

We thank Mary Fernandez who kindly provided us with the SML source codes of an UnQL system. We thank Fritz Henglein and James Cheney, and anonymous reviewers for their thorough comments on earlier versions of the paper. The research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from the National Institute of Informatics, Grant-in-Aid for Scientific Research (B) No. 22300012, Grant-in-Aid for Scientific Research (C) No. 20500043, and Encouragement of Young Scientists (B) of the Grant-in-Aid for Scientific Research No. 20700035.

References

S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In *The Essence of Computation*, pages 269–295, 2002.

S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *POPL 1994*, pages 268–279, 1994.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

J. Bezivin, B. Rumpe, and T. L. Schürr A. Model transformation in practice workshop announcement. In *MoDELS Satellite Events 2005*, pages 120–127, 2005.

R. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.

A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS 2006*, pages 338–347, 2006.

V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *DBPL 1991*, pages 9–19, 1991.

P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.

J. Cheney, U. A. Acar, and A. Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT 2009*, pages 260–283, 2009.

U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. Presented at *MTiP 2005*. `http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf`, 2005.

L. Fegaras. Propagating updates through xml views using lineage tracing. In *ICDE 2010*, pages 309–320, 2010.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL 2005*, pages 233–246, 2005.

R. Glück and A. V. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA 1993*, pages 112–123, 1993.

R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In *PLILP 1994*, pages 165–181, 1994.

G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

S. J. Hegner. Foundations of canonical update support for closed database views. In *ICDT 1990*, pages 422–436, 1990.

S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *SAC 2009*, pages 468–475, 2009.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

F. Jouault and I. Kurtev. Transforming models with ATL. In *MoDELS Satellite Events 2005*, pages 128–138, 2005.

R. Lämmel. Coupled Software Transformations (Extended Abstract). In *SET 2004*, Nov. 2004.

J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, 2003.

D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *PEPM 2007*, pages 21–30, 2007.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP 2007*, pages 47–58, 2007.

L. Meertens. Designing constraint maintainers for user interaction. `http://www.cwi.nl/~lambert`, June 1998.

OMG. MOF QVT final adopted specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE 1995*, pages 251–260, 1995.

O. Pastor and J. C. Molina. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

A. Schürr and F. Klar. 15 years of triple graph grammars. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, pages 411–425. Springer-Verlag, 2008.

T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA 1993*, pages 233–242, Copenhagen, June 1993.

L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE 1999*, pages 572–581, 1999.

S. Staworko, I. Boneva, and B. Groz. The view update problem for XML. In *EDBT Workshops (Updates in XML)*, 2010.

P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MoDELS 2007*, pages 1–15, 2007.

A. Formal Semantics of Traceable Forward Evaluation

We formally give the semantics of traceable forward evaluation that generates traceable views. It is defined in the same way as the original semantics for UnCAL (Buneman et al. 2000). The only difference is that we have to assign a trace ID to every node in the view. The forward semantics $\mathcal{F}[\![e^p]\!]\rho$ for an UnCAL expression e and a variable binding environment ρ is inductively defined on the structure of e. We give the semantics for UnCAL constructs other than $\{\}$, $\cup$, variable reference, **if** and **rec** which are given in Section 4.2.

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.

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

$\boldsymbol{\{l : e\}}$ It prepends an edge on top of the root of the second operand graph.

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

$\boldsymbol{e_1 \oplus e_2}$ It performs a componentwise union like $\cup$, except that no ε-edges are involved.

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

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

$$\begin{aligned}&G_1 \oplus G_2 = (V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2)\\ &\quad\text{where}\ \ (V_1, E_1, I_1, O_1) = G_1\\ &\qquad\quad\ (V_2, E_2, I_2, O_2) = G_2\\ &\qquad\quad\ \mathsf{inMarker}(G_1) \cap \mathsf{inMarker}(G_2) = \emptyset\end{aligned}$$

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

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

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

$$\begin{aligned}&G_1 \mathbin{@^p} G_2 = (V_1 \cup V_2, E_1 \cup E_@ \cup E_2, I_1, O_2)\\ &\quad\text{where}\ \ (V_1, E_1, I_1, O_1) = G_1\\ &\qquad\quad\ (V_2, E_2, I_2, O_2) = G_2\\ &\qquad\quad\ E_@ = \{(u, \varepsilon, v)\\ &\qquad\qquad\qquad\ \mid (u, \&m) \in O_1, (\&m, v) \in I_2\}\end{aligned}$$

Note that @ may introduce unreachable parts in the right operand due to unmatched input/output nodes.

$\mathbf{cycle}(e)$ It forms cycles by connecting matching input/output nodes with ε-edges as in @, and for each input node, it creates a new input node marked identically and connects them to the original input nodes with ε-edges. Unmatched output markers remain.

$$\mathcal{F}[\![\mathbf{cycle}(e)]\!]\rho = \mathbf{cycle}^p(\mathcal{F}[\![e]\!]\rho),$$

where $\mathbf{cycle}^p$ is a cycle operator for a graph concerning position p. A graph $\mathbf{cycle}^p(G)$ is defined by

$$\begin{aligned}\mathbf{cycle}^p(G) &= (V \cup V', E \cup E' \cup E'', I', O')\\ \text{where } (V, E, I, O) &= G\\ V' &= \{\mathsf{Code}\ p\ \&m \mid (\&m, u) \in I\}\\ E' &= \{(\mathsf{Code}\ p\ \&m, \varepsilon, u) \mid (\&m, u) \in I\}\\ E'' &= \{(u, \varepsilon, v) \mid (u, \&m) \in O, (\&m, v) \in I\}\\ I' &= \{(\&m, \mathsf{Code}\ p\ \&m) \mid (\&m, v) \in I\}\\ O' &= \{(u, \&m) \in O \mid (\&m, v) \notin I\}\end{aligned}$$

$\&\boldsymbol{m} := \boldsymbol{e}$ 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 ".".

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

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

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

B. Formal Semantics of Backward Evaluation for In-place Updates

This section gives formal backward semantics using trace IDs assigned during forward evaluation. Semantics for **if**, variable reference and **rec** are omitted since they are already given in Section 5.1.

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

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

A label constant similarly accepts no modification.

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

$\{\boldsymbol{l} : \boldsymbol{e}\}$ 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').

$$\begin{aligned}\mathcal{B}[\![\{l : e\}^p]\!](\rho, G') &= \mathcal{B}[\![l]\!](\rho, a') \uplus_\rho \mathcal{B}[\![e]\!](\rho, G_2')\\ \text{where } a &= l\rho\\ G_2 &= \mathcal{F}[\![e]\!]\rho\\ (a', G_2') &= \mathsf{decomp}_{\{a:pG_2\}}(G')\end{aligned}$$

Here, the decomposition function is defined as follows:

$$\begin{aligned}&\mathsf{decomp}_{\{a_1:pG_2\}}(G') =\\ &\quad (a_1', (V' \setminus \{r'\}, E' \setminus \{e'\}, \{(\&, v)\}, O'))\\ &\text{where } (V_2, E_2, \{(\&, v)\}, O_2) = G_2\\ &\quad (V', E', \{(\&, r')\}, O') = G'\\ &\quad \zeta' = \text{the unique edge in } E' \text{ of the form } (r', a_1', v).\end{aligned}$$

The modified view G' is decomposed into its unique root edge $\zeta' = (r', a_1', v)$ and the rest of the graph rooted at v. If G' has more than one edges 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.

$\boldsymbol{e_1} \cup \boldsymbol{e_2}$ It decomposes (decomp) the view G' using forward results of operands e_1 and e_2, and inductively conducts backward computation on the operands.

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

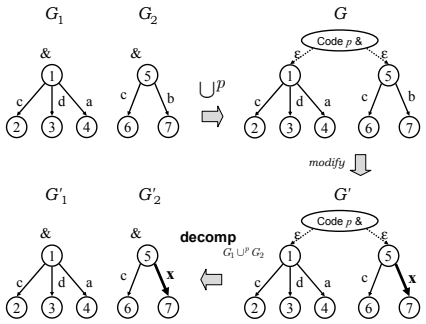


Figure 12. Example for Bidirectional Computation of Union

decomp is defined below. $G_1 \setminus G_2$ denotes component (V, I, E, O) wise set difference.

$$\begin{aligned}&\mathsf{decomp}_{G_1 \cup^p G_2}(G') =\\ &\quad \big(\mathsf{xreachable}(G_1', G_1), \mathsf{xreachable}(G_2', G_2)\big)\\ &\text{where } (V', E', I', O') = G'\\ &\quad (V_i, E_i, I_i, O_i) = G_i\\ &\quad G_i' = \mathsf{reachable}((V', E', I_i, O'))\\ &\text{satisfying}\\ &\quad M = \mathsf{inMarker}(G_1) = \mathsf{outMarker}(G_2)\\ &\quad \forall \&m \in M, (\&m, r') \in I', (r', \varepsilon, v') \in E' :\\ &\quad\quad (\&m, v') \in I_1 \cup I_2\end{aligned}$$

$$\begin{aligned}&\mathsf{unreachable}(G) = G \setminus \mathsf{reachable}(G)\\ &\mathsf{xreachable}(G', G) =\\ &\quad (V^{r\prime} \cup V^u, E^{r\prime} \cup E^u, I^{r\prime} \cup I^u, O^{r\prime} \cup O^u)\\ &\text{where } (V^{r\prime}, E^{r\prime}, I^{r\prime}, O^{r\prime}) = \mathsf{reachable}(G')\\ &\quad (V^u, E^u, I^u, O^u) = \mathsf{unreachable}(G)\end{aligned}$$

Let us exemplify $\mathsf{decomp}_{G_1 \cup^p G_2}(G')$ using Figure 12 in which the label b of edge $(5, \mathtt{b}, 7)$ is modified to x on the view. Recall that in forward semantics $\cup^p$ connects input nodes of the operands with ε-edges. We remove these edges from the input nodes first. Then the original input nodes 1 and 5 are restored using the input nodes of the original graphs G_1 and G_2, and determine (possibly modified) operands by collecting reachable parts from these nodes¶¶. Since the modified edge is reachable from the input node of G_2', the modification belongs to the second operand. For well-behavedness, decomp fails if ε-edges from the input nodes are changed. Note that the forward computation could have glued the two input nodes (1 and 5 in Figure 12) together, but it would make splitting of the view difficult, since both of the operands can be reachable from the glued node. As for trace IDs, this operator does not introduce them except for the input nodes. So no destructing operation is conducted for nodes other than the input nodes. One can easily verify that this operation is reversible. decomp ensures that the input node of the modified graph G' is an origin of a bunch of ε-edges, whose destination node came from the root node of either the original G_1 or the G_2. decomp for other operators are defined similarly to make the operation reversible.

$\boldsymbol{e_1} \oplus \boldsymbol{e_2}$ It is like $\cup$, except that no ε-edge is involved.

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

¶¶ Reachable parts from a given node can be computed by traversing edges from that node. reachable() starts from input node of given graph instead of a given particular node.

$\boldsymbol{e_1 \mathbin{@} e_2}$ 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.

$$\begin{aligned}
&\mathcal{B}[\![(e_1 @ e_2)^p]\!](\rho, G') = \mathcal{B}[\![e_1]\!](\rho, G'_1) \uplus_\rho \mathcal{B}[\![e_2]\!](\rho, G'_2)\\
&\quad \text{where } (G'_1, G'_2) = \mathsf{decomp}_{G_1 @^p G_2}(G')\\
&\qquad\quad G_1 = \mathcal{F}[\![e_1]\!]\rho\\
&\qquad\quad G_2 = \mathcal{F}[\![e_2]\!]\rho\\
&\mathsf{decomp}_{G_1 @^p G_2}(G')\\
&= (\mathsf{xreachable}(G'_1, G_1), \mathsf{xreachable}(G'_2, G_2)) \text{ where}\\
&(V_i, E_i, I_i, O_i) = G_i\\
&(V', E', I', O') = G'\\
&E_@ = \{(u, \varepsilon, v) \mid (u, \&m) \in O_1, (\&m, v) \in I_2\}\\
&G'_i = \mathsf{reachable}((V', E' \setminus E_@, I_i, O_i))
\end{aligned}$$

$\mathbf{cycle}(\boldsymbol{e})$ It removes the ε-edges introduced in the forward evaluation and restores the original I/O nodes.

$$\begin{aligned}
&\mathcal{B}[\![\mathbf{cycle}(e)]\!](\rho, G') = \mathcal{B}[\![e]\!](\rho, G'_2)\\
&\quad \text{where } (V', E', I', O') = G'\\
&\qquad (V, E, I, O) = \mathcal{F}[\![t]\!]\rho\\
&\qquad V' = \{(\mathsf{Code}\ p\ \&m\ u) \mid (\&m, u) \in I\}\\
&\qquad E' = \{(\mathsf{Code}\ p\ \&m\ u, \varepsilon, u) \mid (\&m, u) \in I\}\\
&\qquad E_{\mathbf{cycle}} = \{(u, \varepsilon, v) \mid (u, \&m) \in O, (\&m, v) \in I\}\\
&\qquad G'_2 = (V \setminus V', E \setminus E' \setminus E_{\mathbf{cycle}}, I, O)
\end{aligned}$$

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

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

C. Small-Step Semantics of UnCAL

This section gives a complete definition of the small-step semantics of UnCAL corresponding to the traced evaluation semantics in Section 4. As we mentioned in the paper, small-step semantics is obtained in a straightforward way except where we need to care about shareness and cycles and introduce further computation for **rec**. We basically follow the recursive semantics of UnCAL (Buneman et al. 2000). Unlike the recursive semantics in (Buneman et al. 2000), ours creates the nodes in V_{RecN} in Figure 6 whenever **rec** traverses the node, and whether **rec** traverses these nodes depends on whether they have been visited or not. Additionally, the transformation result of each edge must be escaped by using RecE, as in the bulk semantics. This escaping also follows the recursive semantics because the escaping may traverse a graph before **rec** does, e.g., in the evaluation of $\mathbf{rec}(\lambda(\_,\_).\$g)(\{\mathsf{a} : \{\}\})$; thus, in later narrowing, the escaping narrows variables as well as **rec** does.

In advancing to the formal definition of the small-step evaluation, we extend the definition of the UnCAL expression as:

$$\begin{aligned}
e ::= &\ \ldots\\
&\mid G\\
&\mid v|_G\\
&\mid \mathbf{cnrec}^{M_\mathrm{O}}_{p,v,M}(e_1, \ldots, e_n)\\
&\mid \mathbf{esc}_{p,\zeta}(e)\\
&\mid \mathbf{cnesc}^{M_\mathrm{O}}_{p,v,\zeta}(e_1, \ldots, e_n; a_1, \ldots, a_n),
\end{aligned}$$

where

- G is a graph value appearing in the evaluation, i.e., a quadruple (V, E, I, O),
- $v|_G$ is used internally in **rec** and **esc**, which intuitively means a node v in G just being traversed,
- $\mathbf{cnrec}^{M_\mathrm{O}}_{p,v,M}$ is used to create V_{RecN} nodes as in Figure 6 and then connect each evaluation result from each edge as in the bulk semantics, where M_O is the set of output markers put on v that is used to keep track of the output markers assigned to node v in the input graph of **rec**,
- $\mathbf{esc}_{p,\zeta}$ is used to escape the evaluation result of **rec** by using $\mathsf{RecE}\ p\ \_\ \zeta$, and
- $\mathbf{cnesc}^{M_\mathrm{O}}_{p,v,\zeta}$, a counterpart of **cnrec** for **esc**, is used to create a $\mathsf{RecE}\ p\ v\ \zeta$ node and then connect it to each escaped result from each edge by using an edge labeled a_i, where M_O is similar to that of **cnrec**.

C.1 Small-Step Semantics

Since now a graph value is an (extended) expression, we can define small-step semantics by reduction sequence $e_1 \longrightarrow e_2 \longrightarrow \cdots \longrightarrow G$, where our small-step evaluation $\longrightarrow$ is given in the form of

$$\mathcal{C}[e] \longrightarrow \mathcal{C}[e'] \qquad \text{if} \qquad e \to e',$$

which reads expression $\mathcal{C}[e]$ is reduced to $\mathcal{C}[e']$ via the reduction of e to e'. Here, $\mathcal{C}$ is an *evaluation context* that indicates the redex of the expression, e is the redex of expression $\mathcal{C}[e]$, and $e \to e'$ is the *small-step evaluation relation* that reads redex e is reduced to e'. It is important to separate the redex and the evaluation context from the expression because they are used in later (needed) narrowing; in general, substituting a value for a variable in the non-redex position may change the semantics. In the following, we present the definition of $\mathcal{C}$ and $\to$.

C.1.1 Evaluation Context

The call-by-value evaluation context is defined as

$$\begin{aligned}
\mathcal{C} ::= &\ \Box \mid \mathbf{cycle}(\mathcal{C}) \mid \&m := \mathcal{C} \mid \mathcal{C}\ op\ e \mid G\ op\ \mathcal{C} \mid \{a : \mathcal{C}\}\\
&\mid \mathbf{rec}(\lambda(\$l, \$g).e)(\mathcal{C}) \mid \mathbf{esc}_{p,\zeta}(\mathcal{C})\\
&\mid \mathbf{cnrec}^{M_\mathrm{O}}_{p,v,M}(G_1, \ldots, G_{i-1}, \mathcal{C}, e_{i+1}, \ldots, e_n)\\
&\mid \mathbf{cnesc}^{M_\mathrm{O}}_{p,v,\zeta}(G_1, \ldots, G_{i-1}, \mathcal{C}, e_{i+1}, \ldots, e_n; a_1, \ldots, a_n),
\end{aligned}$$

where op represents a binary graph constructor in UnCAL, i.e., $\cup$, $\oplus$, and @. For an expression e and an evaluation context $\mathcal{C}$, $\mathcal{C}[e]$ denotes the term obtained by replacing a hole $\Box$ in $\mathcal{C}$ with e. Any evaluation context has exactly one $\Box$ and any expression can be written in the form of $\mathcal{C}[e]$, which means that any expression has a unique redex.

C.1.2 Small-Step Evaluation Relation

Figure 13 shows the small-step evaluation relation $e \to e'$. The definition of **rec/esc** basically follows the equation of structural recursion in the form

$$\begin{aligned}
&f(\{a_1 : G_1, \ldots, a_n : G_n\})\\
&\quad = (e(a_1, G_1) \mathbin{@} f(G_1)) \cup \cdots \cup (e(a_n, G_n) \mathbin{@} f(G_n)).
\end{aligned}$$

We use the above equation for the simplicity of both memoization and narrowing. Unlike the original equations of structural recursion where a recursion traverses a set of edges, function f above traverses a node of a graph. The actual definition of **rec/esc** is a bit more complicated than the above because their small-step semantics must coincide with the traced evaluation semantics and must handle a node with input/output markers; the rule of **cnrec/cnesc** is responsible to this. In the following, we first explain the rules of **rec** and **cnrec**, and then those of **esc** and **cnesc**.

The definition of **rec** consists of the three rules: the first one for a graph that may have multiple root nodes, and the second and third ones for a graph with a single root. Intuitively, the first rule partitions a graph to single-root graphs by extracting the reachable part from each input node, and then proceeds computation of **rec**. Formally, the definition just follows the equation below that holds

$$\textbf{if } a_1 = a_2 \textbf{ then } e_1 \textbf{ else } e_2 \to e_1 \qquad (\text{if } a_1 = a_2)$$
$$\textbf{if } a_1 = a_2 \textbf{ then } e_1 \textbf{ else } e_2 \to e_2 \qquad (\text{if } a_1 \neq a_2)$$

$$\textbf{rec}(\lambda(\$l, \$g).e)(G_{(\_,\_,I,\_)})^p \to \bigoplus\nolimits_{\&m \in I} \big(\&m := \textbf{rec}(\lambda(\$l, \$g).e)(I(\&m)|_G)^p\big)$$

$$\textbf{rec}(\lambda(\$l, \$g).e)(v|_G)^p \to \textbf{cnrec}^{v|_G\,M_{\mathrm{O}}}_{p,v,M_e}()$$
(if v has already been traversed by **rec** at p)

$$\textbf{rec}(\lambda(\$l, \$g).e)(v|_G)^p \to \textbf{cnrec}^{M^{v|_G}_{\mathrm{O}}}_{p,v,M_e}(e_1, \ldots, e_n)$$
(if v has not yet been traversed by **rec** at p)
$$e_i = \begin{cases} \textbf{esc}_{p,\zeta_i}(e[\$l \mapsto a_i, \$g \mapsto G_i])^{q_i} & (\text{if } a_i \neq \varepsilon) \\ \quad @\ \textbf{rec}(\lambda(\$l, \$g).e)(v_i|_G)^p & \\ \textbf{rec}(\lambda(\$l, \$g).e)(v_i|_G)^p & (\text{if } a_i = \varepsilon) \end{cases}$$
where $(\_, a_i, v_i) = \zeta_i$
$G_i = \mathsf{subgraph}(G, \zeta_i)$
$q_1, \ldots, q_n$ are fresh code positions
v has the out edges $\zeta_1, \ldots, \zeta_n$.

$$\textbf{cnrec}^{M_{\mathrm{O}}}_{p,v,M}(G_1, \ldots, G_n) \to (V', E', I', O')$$
$V' = V_{\mathsf{RecN}} \cup V_1 \cup \cdots \cup V_n$
$E' = E_{\mathsf{RecN}} \cup E_1 \cup \cdots \cup E_n$
$I' = \{(\&m, \mathsf{RecN}\ p\ v\ \&m) \mid \&m \in M\}$
$O' = O_{\mathsf{RecN}} \cup O_1 \cup \cdots \cup O_n$
$V_{\mathsf{RecN}} = \{\mathsf{RecN}\ p\ v\ \&m \mid \&m \in M\}$
$E_{\mathsf{RecN}} = \{(\mathsf{RecN}\ p\ v\ \&m, \varepsilon, I_i(\&m)) \mid \&m \in M\}$
$O_{\mathsf{RecN}} = \{(\mathsf{RecN}\ p\ v\ \&m, \&n.\&m) \mid \&m \in M, \&n \in M_{\mathrm{O}}\}$
$(V_i, E_i, I_i, O_i) = G_i$

$$\textbf{esc}_{p,\zeta}(G_{(\_,\_,I,\_)})^q \to \bigoplus\nolimits_{\&m \in I} \big(\&m := \textbf{esc}_{p,\zeta}(I(\&m)|_G)^q\big)$$

$$\textbf{esc}_{p,\zeta}(v|_G)^q \to \textbf{cnesc}^{M^{v|_G}_{\mathrm{O}}}_{p,v,\zeta}()$$
(if v has already been traversed by **esc** at q)

$$\textbf{esc}_{p,\zeta}(v|_G)^q \to \textbf{cnesc}^{M^{v|_G}_{\mathrm{O}}}_{p,v,\zeta}(e_1, \ldots, e_n; a_1, \ldots, a_n)$$
(if v has not been traversed by **esc** at q)
$e_i = \textbf{esc}_{p,\zeta}(v_i|_G)^q$
$(\_, a_i, v_i) = \zeta_i$
v has the out-edges $\zeta_1, \ldots, \zeta_n$.

$$\textbf{cnesc}^{M_{\mathrm{O}}}_{p,v,\zeta}(G_1, \ldots, G_n; a_1, \ldots, a_n) \to (V', E', I', O')$$
$V' = \{v_{\mathsf{RecE}}\} \cup V_1 \cup \cdots \cup V_n$
$E' = \{(v_{\mathsf{RecE}}, a_i, I_i(\&)) \mid 1 \leq i \leq n\} \cup E_1 \cup \cdots \cup E_n$
$I' = \{(\&, v_{\mathsf{RecE}})\}$
$O' = \{(v_{\mathsf{RecE}}, \&m) \mid \&m \in M_{\mathrm{O}}\} \cup O_1 \cup \cdots \cup O_n$
$(V_i, E_i, I_i, O_i) = G_i$
$v_{\mathsf{RecE}} = \mathsf{RecE}\ p\ v\ \zeta$

where
$M_e = \mathsf{inMarker}(e) \cup \mathsf{outMarker}(e)$
$M^{v|_G}_{\mathrm{O}} = \{\&m \mid (v, \&m) \in O\}$ with $(\_, \_, \_, O) = G$

Figure 13. Small-Step Evaluation Rules for UnCAL: Rules for graph constructors have been omitted because they are obvious.

for **rec** in the traced semantics.

$$\textbf{rec}(\lambda(\$l, \$g).e)(G_{(V,E,I,O)}) = \bigoplus_{\&m \in G} (\&m := \textbf{rec}(\lambda(\$l, \$g).e)(G_{\&m})) \qquad (*)$$
where $G_{\&m} = (V, E, \{(\&, I(\&m))\}, O)$

Since the single-root graph reachable from a given node in a graph can be identified with the node itself in the graph, we use $I(\&m)|_G$ instead of extracting the reachable part from $I(\&m)$. This not only simplifies the definition of the first rule but also plays an important role in the later narrowing where a narrowing variable can be seen as a node of which edges are not fixed yet.

The second rule and the third rule of **rec** use memo to find whether a node has been traversed by **rec** or not. If the node has not yet been traversed by **rec**, the evaluation rule

$$\textbf{rec}(\lambda(\$l, \$g).e)(v|_G)^p \to \textbf{cnrec}^{M^{v|_G}_{\mathrm{O}}}_{p,v,M_e}(e_1, \ldots, e_n)$$
(if v has not yet been traversed by **rec** at p)
$$e_i = \begin{cases} \textbf{esc}_{p,\zeta_i}(e[\$l \mapsto a_i, \$g \mapsto G_i])^{q_i} & (\text{if } a_i \neq \varepsilon) \\ \quad @\ \textbf{rec}(\lambda(\$l, \$g).e)(v_i|_G)^p & \\ \textbf{rec}(\lambda(\$l, \$g).e)(v_i|_G)^p & (\text{if } a_i = \varepsilon) \end{cases}$$
where $(\_, a_i, v_i) = \zeta_i$
$G_i = \mathsf{subgraph}(G, \zeta_i)$
$q_1, \ldots, q_n$ are fresh code positions
v has the out edges $\zeta_1, \ldots, \zeta_n$.

proceeds to evaluating on each edge connected from the node, memoizes the traversed node, and then, by using **cnrec**, connects the resulted graph. In addition to p, v and M_e, M_{O} is passed to the evaluation of **cnrec** because whether the V_{RecN} nodes have output markers depends on whether v has output markers. We need not care input markers here because the first rule already remove them. Recall the correspondence between the first rule and Equation (*). The introduced **esc** expressions have fresh code positions $q_1, \ldots, q_n$ for the memoization that will be performed in the evaluation of **esc**. Note that, in the evaluation of **rec** body e, $\$g$ will be bound to G_i instead of $v|_G$ because $v|_G$ is not a graph nor a value***. If the node has already been traversed by **rec**, the evaluation rule

$$\textbf{rec}(\lambda(\$l, \$g).e)(v|_G)^p \to \textbf{cnrec}^{v|_G\,M_{\mathrm{O}}}_{p,v,M_e}()$$
(if v has already been traversed by **rec** at p)

just returns V_{RecN} nodes. Note that we only do a membership test for memo instead of looking it up because V_{RecN} nodes generated by **cnrec** work as a corresponding value for the entry v in the memo.

The rule of **cnrec** creates the V_{RecN} nodes, and connects them to each evaluation result G_i from each edge of v. The set of edges E_{RecN} represents this connection. Then, the rule puts the input marker $\&m$ to a node $\mathsf{RecN}\ p\ v\ \&m$ in V_{RecN} for the further recursive connection. Note that the rule does not create any input markers corresponding to I_{RecN} because of the relationship between the first rule of **rec** and Equation (*).

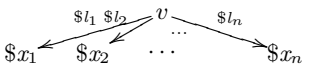
The rules of escaping expression $\textbf{esc}_{p,\zeta}$ is defined so that, when the memo is empty, $\textbf{esc}_{p,\zeta}(G)$ evaluates to the graph obtained from G by replacing each node v by $\mathsf{RecE}\ p\ v\ \zeta$. Similar to **rec**, the definition of **esc** consists of the three rules. We only explain the second rule because the other rules are similar to **rec**. The second rule

$$\textbf{esc}_{p,\zeta}(v|_G)^q \to \textbf{cnesc}^{M^{v|_G}_{\mathrm{O}}}_{p,v,\zeta}(e_1, \ldots, e_n; a_1, \ldots, a_n)$$
(if v has not been traversed by **esc** at q)
$e_i = \textbf{esc}_{p,\zeta}(v_i|_G)^q$
$(\_, a_i, v_i) = \zeta_i$
v has the out-edges $\zeta_1, \ldots, \zeta_n$.

*** It is sure that we can treat $v|_G$ as a value. However, it requires a large amount of changes on the evaluation rules, although the change is straightforward.

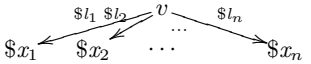
$(\mathcal{C}[e], \phi) \rightsquigarrow_\emptyset (\mathcal{C}[e'], \phi)$ if $e \to e'$

$(\mathcal{C}[\mathbf{if}\ \$l{=}l\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2], \phi) \rightsquigarrow_\theta (\mathcal{C}[e_1], \phi)\theta \quad \theta{=}\{\$l{\mapsto}l\}$
$(\mathcal{C}[\mathbf{if}\ \$l{=}l\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2], \phi) \rightsquigarrow_\emptyset (\mathcal{C}[e_2], \phi \cap (\$l \neq l))$
$(\mathcal{C}[\mathbf{if}\ a{=}\$l\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2], \phi) \rightsquigarrow_\theta (\mathcal{C}[e_1], \phi)\theta \quad \theta{=}\{\$l{\mapsto}a\}$
$(\mathcal{C}[\mathbf{if}\ a{=}\$l\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2], \phi) \rightsquigarrow_\emptyset (\mathcal{C}[e_2], \phi \cap (\$l \neq a))$

$(\mathcal{C}[\mathbf{rec}(\lambda(\$l, \$g).e)(\$x|_G)^p], \phi) \rightsquigarrow_\theta (\mathcal{C}\theta[e'], \phi\theta)$ (for arbitrary n)
$\theta = \{\$x \mapsto v|_{G_{\mathrm{extend}}}\}, \quad \mathbf{rec}(\lambda(\$l, \$g).e\theta)(v|_{G\theta}) \to e'$
$G_{\mathrm{extend}} =$ 

$\$x_1, \ldots, \$x_n, \$l_1, \ldots, \$l_n$ are fresh variables
$v \in \mathit{SrcID}$ is a fresh node id.

$(\mathcal{C}[\mathbf{rec}(\lambda(\$l, \$g).e)(\$x|_G)^p], \phi) \rightsquigarrow_\theta (\mathcal{C}\theta[e'], \phi\theta)$
$\theta = \{\$x \mapsto v|_G\}, \quad \mathbf{rec}(\lambda(\$l, \$g).e\theta)(v|_{G\theta}) \to e'$
the $\mathbf{rec}$ at p has already traversed v.

$(\mathcal{C}[\mathbf{esc}_{p,\zeta}(\$x|_G)^q], \phi) \rightsquigarrow_\theta (\mathcal{C}\theta[e'], \phi\theta)$ (for arbitrary n)
$\theta = \{\$x \mapsto v|_{G_{\mathrm{extend}}}\}, \quad \mathbf{esc}_{p,\zeta}(v|_{G\theta})^q \to e'$
$G_{\mathrm{extend}} =$ 

$\$x_1, \ldots, \$x_n, \$l_1, \ldots, \$l_n$ are fresh variables.
$v \in \mathit{SrcID}$ is a fresh node id.

$(\mathcal{C}[\mathbf{esc}_{p,\zeta}(\$x|_G)^q], \phi) \rightsquigarrow_\theta (\mathcal{C}\theta[e'], \phi\theta)$
$\theta = \{\$x \mapsto v|_G\}, \quad \mathbf{esc}_{p,\zeta}(v|_{G\theta})^q \to e'$
the $\mathbf{rec}$ at p has already traversed v.

Figure 14. Narrowing Rules for UnCAL

proceeds to evaluation of each graph connected from the node, memoizes the traversed node, and then, by using **cnesc**, connects the resulted graph. Unlike **rec**, **esc** passes the labels $a_1, \ldots, a_n$ to **cnesc** for the simplicity of **cnesc**. Note that we cannot use ε-edges in the connection of the escaped results because of the correspondence to the trace semantics. We believe that now the definition of **cnesc** should be intuitive for readers.

C.2 Narrowing

Narrowing rules, which are used to construct perfect process trees in URA, are obtained from the small-step semantics in a straightforward way except where we need to clarify the definition of narrowing variables. Merely treating a narrowing variable as just a syntactic object $\$x$ is unsatisfactory; a graph value (V, E, I, O) cannot contain a syntactic object $\$x$. Instead, we treat them as a special kind of nodes by extending the definition of trace IDs as:

$$\begin{aligned} \mathit{TraceID} ::= {} & \ldots \\ \mid {} & \$x, \end{aligned}$$

under which a graph can contain narrowing variables.

A narrowing variable will be instantiated to some kind of graph G that is identical to $v|_G$ for some v, instead of an arbitrary graph that may contain multiple roots or may contain unreachable part. Node-by-node substitution $\{\$x \mapsto v|_G\}$ represents the replacement of node $\$x$ with v in G, and application $G'\{\$x \mapsto v|_G\}$ intuitively means a graph obtained from the componentwise union of G and G' by replacing every node $\$x$ with v. Note that the definition may introduce a loop; e.g., for $G' = (\{\$x\}, \emptyset, \{(\&, \$x)\}, \emptyset)$ and $G = (\{1, \$x\}, \{(1, \mathtt{a}, \$x)\}, \emptyset, \emptyset)$, we have

$$G'\{\$x \mapsto 1|_G\} = (\{1\}, \{(1, \mathtt{a}, 1)\}, \{(\&, 1)\}, \emptyset).$$

Similar to the usual substitutions, it is easy to define composition of node-by-node substitutions and to extend applications of them to expressions.

Now, we define narrowing rules. Figure 14 shows the formal definition of the narrowing relation $\rightsquigarrow$. Since narrowing at **if** induces constraints on variables, each narrowing step is written as $(e, \phi) \rightsquigarrow_\theta (e', \phi')$, where θ is a substitution used in narrowing. The rules find the most general unifier to proceed with the evaluation when we encounter a free variable in redex. Since free variables may appear within the evaluation context, the narrowing rules involve $\mathcal{C}$. Narrowing for **rec/esc** is defined for arbitrary n so that the argument variable of **rec/esc** can be instantiated to a node with an arbitrary number of outgoing edges. Another important point in narrowing is the second rule of **rec/esc**. To make loops by narrowing, we replace a narrowing variable with an already-traversed node. Note that ϕ may contain constraints of the form $\$l_1 \neq \$l_2$ due to the second rule of **if**. Such constraints appear in the perfect process tree of *consecutive* in Example 4. Our narrowing rules have been borrowed from (Abramov and Glück 2002; Glück and Klimov 1993), and those of needed narrowing (Antoy et al. 1994) for call-by-value small-step semantics. The soundness and completeness of needed narrowing are discussed in (Antoy et al. 1994).