ISSN 1884-0760

GRACE TECHNICAL REPORTS

Bidirectionalizing Structural Recursion on Graphs

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

GRACE-TR 2009–03 August 2009

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 Structural Recursion on Graphs

Soichiro Hidaka Zhenjiang Hu Kazuhiro Inaba Hiroyuki Kato
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
{hidaka,hu,kinaba,kato}@nii.ac.jp

Kazutaka Matsuda
The University of Tokyo/JSPS Research Fellow
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
kztk@ipl.t.u-tokyo.ac.jp

Keisuke Nakano
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi
Tokyo 182-8585, Japan
ksk@cs.uec.ac.jp

August 31, 2009

Abstract

The bidirectional transformation problem has been attracting more and more attention in the programming language community. Despite many promising results about bidirectional transformation on linear strings or tree-like data structures, it remains as an open problem whether it is possible to design a language that can support practical development of bidirectional transformations on *graphs*. In this paper, we propose the first language-based (linguistic) solution towards solving this challenging problem. We approach this problem by giving a well-behaved bidirectional semantics for structural recursion (on graphs), the most essential construct in UnCAL which is the underlying graph algebra for the known UnQL graph query language. In particular, we carefully refine the existing forward evaluation of structural recursion so that it can produce useful trace information for later backward evaluation, and extending the bulk semantics of structural recursion from forward evaluation to backward evaluation. We have formally proved the well-behavedness of our bidirectional semantics, fully implemented bidirectional transformation engine for UnQL, and confirmed the effectiveness of our approach through many non-trivial examples including typical transformation in database and software engineering.

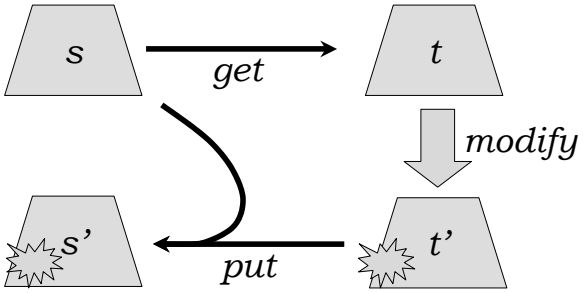


Figure 1: Bidirectional Transformation

1 Introduction

Bidirectional transformation (Foster et al. 2005; Czarnecki et al. 2009) is characterized by a pair of transformations, a forward transformation *get* and a backward transformation *put* as in Figure 1. The forward transformation *get* is used to produce from a source s to a view t, while the backward transformation *put* is to reflect modification on the view back to the source. Practical examples of bidirectional transformation include 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 transformation, the data that can be dealt with are limited to linear strings or tree-like data structures. It remains as an open problem (Czarnecki et al. 2009) whether it is possible to design a language that can support practical development of bidirectional transformations on *graphs* which contains node sharing and cycles. It would be remarkably useful in many applications if bidirectional transformation can be applied to graph data structures, because graphs play an irreplaceable role in representing more complex data structures such as models (e.g., UML diagrams) in model-driven software development (Stevens 2007), and Object Exchange Model (OEM) for exchanging arbitrary database structures (Papakonstantinou et al. 1995).

There are many challenges in designing a language for bidirectional transformation on graphs. First, unlike strings and trees, there is no unique way to represent, construct, and decompose a general graph, and this requires more precise definition of *equivalence* between two graphs. Second, graphs have sharing nodes and cycles, which makes forward computation much more complicated than that on trees (let alone to say about backward

computation), because naïve computation on graphs would visit the same nodes many times and possibly infinitely often.

In this paper, we give the first language-based (linguistic) solution to the problem of bidirectional graph transformation. We approach this problem by providing a bidirectional semantics for an existing graph query language UnQL (Buneman et al. 2000). We choose UnQL as the basis of our bidirectional graph transformation for the following two reasons.

- First, UnQL is a graph querying language that has been intensively studied in the database community with solid foundation and efficient implementation based on graph algebra. It has a concise and powerful surface syntax based on *select-where* clauses like SQL, and can be easily used to describe many interesting graph transformations.

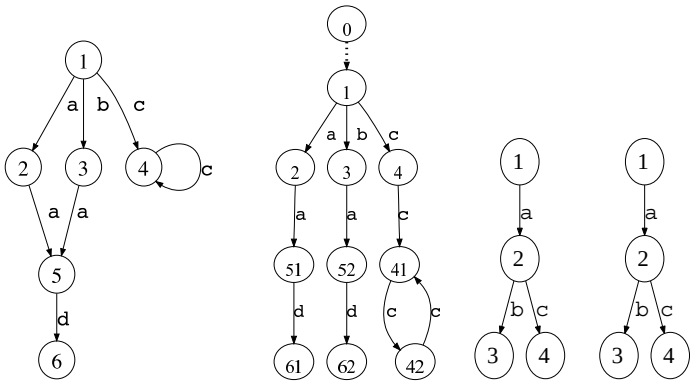
- Second, and more importantly, graph transformations in UnQL are *structured* in the sense that any transformation can be expressed in terms structural recursion, which can be evaluated in a *bulk* way (Buneman et al. 2000); a structural recursion is evaluated by first processing *in parallel* on all edges of the input graph and then combining the results. This feature significantly contributes to our bidirectionalization because it helps us to trace a corresponding source from its result.

Our technical contributions are summarized as follows.

- We are, as far as we are aware, the first who recognize the importance of structural recursion and its bulk semantics in addressing the challenging problem of bidirectional graph transformation, and succeeded in a general graph transformation framework based on structural recursion. We show that graph transformations described with structured recursion are not only suitable for optimization as intensively studied so far (Buneman et al. 2000), but also make backward evaluation easier.

- We give a formal definition of bidirectional semantics for structural recursion (Section 4), by (1) refining the existing forward evaluation so that it can produce useful trace information for later backward evaluation, and (2) extending the bulk semantics of structural recursion from forward evaluation to backward evaluation. And we prove that our bidirectional semantics is well-behaved. The success of bidirectionalizing non-trivial UnQL shows practical power of our approach.

- We have fully implemented our bidirectionalization presented in this paper and confirmed the effectiveness of our approach through many non-trivial examples including typical transformations in database management and software engineering (Section 5). More examples and demos are available in our project web site `http://www.biglab.org`.



(a) A Simple Graph (b) An Equivalent Graph (c) (d) ($\sim 2(c)$)

Figure 2: Graph Equivalence Based on Bisimulation

The rest of this paper is organized as follows. We start with a brief overview of the basic idea of graph data model and graph query language UnQL in Section 2, before we show how an UnQL query program can be automatically transformed to structural recursion in its underlying graph algebra UnCAL in Section 3. Then we provide bidirectional semantics for UnCAL and prove that the bidirectional semantics is well-behaved in section 4. Section 5 discusses our implementation and experimental results, Section 6 summarizes related work, and Section 7 concludes the paper.

2 Graph Transformation in UnQL

We start with a brief overview of the graph data model and unidirectional graph transformation in UnQL (Buneman et al. 2000), a graph query language to be bidirectionalized in this paper.

2.1 Graph Data Model

External Graph Representation Graphs in UnQL are rooted and directed cyclic graphs with no order between outgoing edges. They are edge-labelled in the sense that all information is stored as labels on edges and the labels on nodes serve as a unique identifier and have no particular meaning. Figure 2(a) gives a small example of a directed cyclic graph with six nodes and seven edges. In text, it is represented by

$$\begin{aligned} G &= \{\mathtt{a} : \{\mathtt{a} : G_1\}, \mathtt{b} : \{\mathtt{a} : G_1\}, \mathtt{c} : G_2\} \\ G_1 &= \{\mathtt{d} : \{\}\} \\ G_2 &= \{\mathtt{c} : G_2\} \end{aligned}$$

where the notation $\{l_1 : G_1, \ldots, l_n : G_n\}$ denotes a set representing a graph which contains n edges with labels $l_1, \ldots, l_n$, each l_i pointing to a graph G_i, and the empty set $\{\}$ denotes a graph with a single node. Two graphs G_1 and G_2 can be merged using set union operation $G_1 \cup G_2$. This graph model is set-based in the sense that $\{\mathtt{a} : \{\}, \mathtt{a} : \{\}\}$ and $\{\mathtt{a} : \{\}\}$ represent the same graph. In addition, the ε-edge is allowed to represent shortcut of two nodes, and works like ε-transition in automaton. For instance, we have

$$\{\varepsilon : \{\mathtt{a} : G_1\}, \varepsilon : \{\mathtt{b} : G_2\}\} = \{\mathtt{a} : G_1, \mathtt{b} : G_2\}.$$

Internal Graph Representation While the external graph representation suffices for users to consider when writing graph transformation, the internal graph representation is designed for internal implementation and semantics description as in Section 4. Different from the external representation, nodes in the internal representation may be marked with *input* and *output marker*, which are used as an interface for composition of graphs.

Now we describe the formal definition and notations of graphs. These notations will be used to specify the bidirectional semantics of UnCAL in Section 4. Let *Label*, $\mathcal{X}$ and $\mathcal{Y}$ be a (possibly infinite) set of labels, a set of input markers, and a set of output markers, respectively. A graph G is denoted by a quadruple (V, E, I, O), where V is a set of nodes, $E \subseteq V \times \mathit{Label}_\epsilon \times V$ with $\mathit{Label}_\epsilon = \mathit{Label} \cup \{\epsilon\}$ is a set of edges, $I \subseteq \mathcal{X} \times V$ is a set of pairs of input markers and corresponding input node, and $O \subseteq V \times \mathcal{Y}$ is a set of pairs of output nodes and associated output marker. For each marker $\&x \in \mathcal{X}$, there is at most one node v such that $(\&x, v) \in I$. The node v is called *input node* with marker $\&x$ and denoted by $I(\&x)$. In contrast, more than one node can be marked with an identical output marker. They are called *output nodes*.

Intuitively, input nodes can be regarded as root nodes of the graph. In other words, although the external representation limits a graph to be singly rooted, internally we deal with multiple roots. For singly rooted graphs, we implicitly use the default marker `&` to indicate its single root. An output node can be regarded as a "context-hole" of graphs where an input node with the same marker is plugged later. Indeed, the "vertical graph composition" operator $G_1 \mathbin{@} G_2$ is defined to plug the input nodes of G_2 into their corresponding output nodes in G_1.

For example, in internal representation, the simple graph in Figure 2(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 $O = \{\}$. This graph is not really typical, as it has no ε-edge, only one input node, and no output node. We will see more examples later.

Graph Equivalence *Graph bisimulation* defines value equivalence between graph instances. The intuition is that two graphs are value equivalent

if they are equal when viewed as sets of paths from the root. Informally, we say that there is a simulation from graph G_1 to graph G_2 if every node x_1 in G_1 has a counterpart x_2 in G_2, and if there is an edge form 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 counterpart of y_1. In UnQL data model, graph equivalence of two graphs requires the correspondence of input and output markers between them as well.

For instance, the graph in Figure 2(b) is value equivalent to the graph in Figure 2(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.

Note also that sets of paths from the root do not always represent value equivalence. The graph in Figure 2(c) is not value equivalent to the graph in Figure 2(d) although they are represented by an identical set of paths $\{a.b, a.c\}$ from the root.

As a remark, the notion of bisimulation is useful because it allows variation in representing semantically equivalent graphs. It has been shown that a graph transformation defined in UnQL preserves bisimilarity (Buneman et al. 2000). If two graphs G_1 and G_2 are bisimilar, $f(G_1)$ and $f(G_2)$ are bisimilar for any transformation f in UnQL.

2.2 UnQL

UnQL has a convenient **select-where** construct like SQL to extract information from a graph and building a new graph with the information. Figure 3 shows an abstract syntax of UnQL. We omit the detailed explanation of the language syntax, which can be found in (Buneman et al. 2000). Rather we illustrate the important features through some examples. Variables in UnQL are prefixed with \$ in this paper. For all examples below, we assume a variable $\$db$ is bound to the graph in Figure 2(a).

2.2.1 The select-where Construct

A query of the form **select** T **where** $B_1, \ldots, B_n$ extracts information from graphs based on the binding and/or Boolean conditions $B_1, \ldots, B_n$ and constructs a graph according to the T.

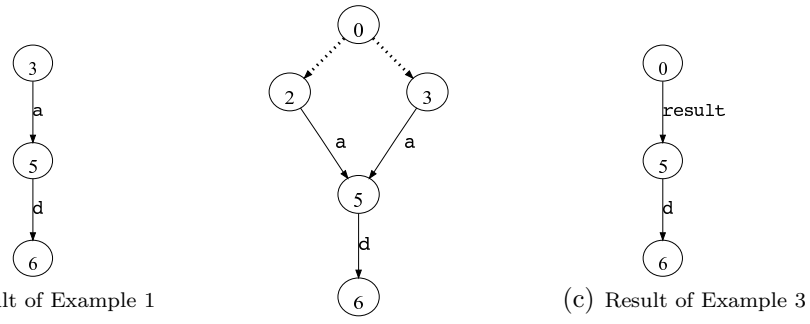
Example 1. The following query returns subgraphs that are pointed by `b` from the root of $\$db$.

$$\textbf{select } \$G \textbf{ where } \{\texttt{b} : \$G\} \textbf{ in } \$db$$

This query first matches the graph pattern $\{\texttt{b} : \$G\}$ against the graph $\$db$ and gets bindings for $\$G$, and then produces the result according to the **select** part. Figure 4(a) shows the result of this query. □

$$
\begin{array}{lll}
\text{(query)} & Q & ::= \ \textbf{select}\ T\ \textbf{where}\ B, \ldots, B \\
\text{(template)} & T & ::= \ \{L : T, \ldots, L : T\} \ \mid \ \$G \ \mid \ Q \\
 & & \mid \ T \cup T \ \mid \ f(\$G) \\
 & & \mid \ \textbf{if}\ L = L\ \textbf{then}\ T\ \textbf{else}\ T \\
 & & \mid \ \textbf{let sfun}\ f\ \{Lp : Gp\}\ = T \\
 & & \qquad\qquad \mid \ f\ \{Lp : Gp\}\ = T \\
 & & \qquad\qquad\quad \ldots \\
 & & \qquad\quad \textbf{sfun}\ f'\ \{Lp : Gp\} = T \\
 & & \qquad\qquad \mid \ f'\ \{Lp : Gp\} = T \\
 & & \qquad\qquad\quad \ldots \\
 & & \qquad\quad \ldots \\
 & & \textbf{in}\ T \\
\text{(condition)} & B & ::= \ Gp\ \textbf{in}\ \$G \ \mid \ L = L \ \mid \ L \neq L \\
\text{(label)} & L & ::= \ \$l \ \mid \ \texttt{a} \\
\text{(label pattern)} & Lp & ::= \ \$l \ \mid \ Rp \\
\text{(graph pattern)} & Gp & ::= \ \$G \ \mid \ \{Lp : Gp, \ldots, Lp : Gp\} \\
\text{(regular} & Rp & ::= \ \texttt{a} \ \mid \ \texttt{\_} \ \mid \ Rp.Rp \\
\text{path pattern)} & & \mid \ (Rp|Rp) \ \mid \ Rp? \ \mid \ Rp* \\
\end{array}
$$

Figure 3: Syntax of UnQL



(a) Result of Example 1

(b) Result of Example 2

(c) Result of Example 3

Figure 4: Result Graphs of Query Examples on Figure 2(a)

Example 2. The following query has multiple conditions in the **where** part and construction of graphs in the **select** part.

$$\begin{array}{l} \textbf{select } \$G_1 \cup \$G_2 \textbf{ where } \{\texttt{b} : \$G_1\} \textbf{ in } \$db, \\ \qquad\qquad\qquad\qquad\quad \{\texttt{a} : \$G_2\} \textbf{ in } \$db \end{array}$$

It returns all subgraphs that are pointed by either `b` or `a` from the root. Figure 4(b) shows the result of this query. □

2.2.2 Regular Path Patterns

Regular path patterns, similar to XPath, are provided for concisely expressing "deep" queries against a graph.

Example 3. Consider the following query.

$$\begin{array}{l} \textbf{select } \{\texttt{result} : \$G_1\} \\ \textbf{where } \{\_*.(\texttt{a}|\texttt{b}) : \$G_1\} \textbf{ in } \$db, \\ \qquad\quad \{\$l : \$G_2\} \textbf{ in } \$G_1, \\ \qquad\quad \$l \neq \texttt{a} \end{array}$$

It extracts all subgraphs $\$G_1$ according to the regular path `_`$*.(\texttt{a}|\texttt{b})$ (i.e., any path ended with an edge labelled `a` or `b`), keeps those subgraphs that do not contain an edge of `a` from their root, and glues the results with new edges labelled `result`. It returns all subgraphs that are pointed by either `b` or `a` from the root. Figure 4(c) shows the result of this query. □

2.2.3 Structural Recursion

Structural recursion is powerful to define various functions to manipulate graphs. A structural recursion f on graphs is a very simple recursive computation scheme on graphs defined by

$$\begin{array}{lcl} f\ \{\} & = & \{\} \\ f\ \{\$l : \$G\} & = & t(\$l, \$G) \odot f(\$G) \\ f\ (\$G_1 \cup \$G_2) & = & f(\$G_1) \cup f(\$G_2) \end{array}$$

where $\odot$ is a given binary operator and the term $t(\$l, \$G)$ does not contain recursive call to f. Different choice of $\odot$ defines different function. Function f is homomorphic in the sense that application result of a union of two graphs coincides with a union of application result of them. Since the first and the third equations are the same for any definition, we may omit them and simplify the above definition as:

$$\textbf{sfun } f\ \{\$l : \$G\} = t(\$l, \$G) \odot f(\$G).$$

Note that structural recursion is similar to the familiar higher-order function *map* in functional programming language, because it basically

manipulates each edge of the graph in parallel. Note also that structural recursion has a syntactic restriction that no return value of a function should be fed to another function as its input. For instance, a function definition

$$\textbf{sfun}\ \ f\ \{\$l : \$G\} = t(\$l, \$G) \odot f(g(\$G))$$

is invalid because $g\ (\$G)$ is fed to function f.

This restriction guarantees that the recursion always terminates.

Example 4. As a simple example, we use the following structural recursion to replace all labels `a` by `d` and delete the edges labelled `c` for an input graph.

$$\begin{aligned}\textbf{sfun}\ \ a2d\_xc\ \{\$l : \$G\} = {} & \textbf{if}\ \$l = \texttt{a}\ \textbf{then}\ \{\texttt{d} : a2d\_xc(\$G)\} \\ & \textbf{else if}\ \$l = \texttt{c}\ \textbf{then}\ a2d\_xc(\$G) \\ & \textbf{else}\ \{\$l : a2d\_xc(\$G)\}\end{aligned}$$ □

A natural extension of the above structural recursion is to allow mutual recursion as shown in Figure 3. For example, a mutual recursive definition

$$\begin{aligned}\textbf{sfun}\ \ & h\ \{\texttt{b} : \$G\}\ = \{\texttt{b} : a2e(\$G)\} \\ \mid\ & h\ \{\$l : \$G\} = h(\$G) \\ \textbf{sfun}\ \ & a2e\ \{\texttt{a} : \$G\}\ = \{\texttt{e} : a2e(\$G)\} \\ \mid\ & a2e\ \{\$l : \$G\} = \{\$l : a2e(\$G)\}\end{aligned}$$

defines a function h which erases all edges until it reaches a `b`. After that it copies the graph, but replace every `a` with an `e`.

2.3 A Practical Example: Customer2Order

As a more practical example, we consider a transformation from customers' graph to orders' graph, which is adapted from a similar example in the textbook on model-driven software development (Pastor and Molina 2007). It will serve as one of the running examples of this paper.

Figure 5 gives a simple graph representing customers' information. Remember that all information should be stored on labels of edges in our graph model. All numbers in nodes have no particular meaning. The graph has a root pointing to two customers, each having a name, some email addresses, several addresses of different types (e.g. shipping or contractual customer address). A customer can have many customer orders.

Now consider how to generate from the customers' graph a graph that represents those information of those orders that have type of "shipping", such that its root points to all the orders and each order contains order information of the date, the order number, the customer name, and the address to which the goods should be delivered. This transformation can be

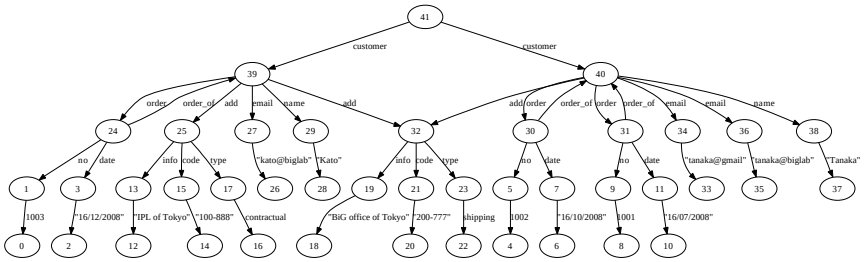


Figure 5: Cyclic Graph g_{cus} Representing Customer-Centric Database

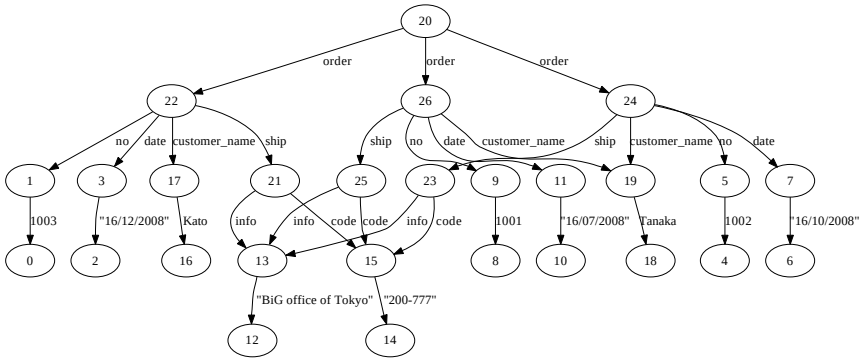


Figure 6: Graph g_{ord} Representing Order-Centric Database

expressed in UnQL as follows, and generates the graph in Figure 6.

$$
\begin{array}{ll}
\textbf{select} & \{\texttt{order}: \{\texttt{date}: \$\mathit{date}, \\
& \quad \texttt{no}: \$\mathit{no}, \\
& \quad \texttt{customer\_name}: \$\mathit{name}, \\
& \quad \texttt{addr}: \$a\}\} \\
\textbf{where} & \{\texttt{customer.order}: \$o\}\ \textbf{in}\ \$\mathit{customer\_db}, \\
& \{\texttt{order\_of}: \$c,\ \texttt{date}: \$\mathit{date},\ \texttt{no}: \$\mathit{no}\}\ \textbf{in}\ \$o, \\
& \{\texttt{add}: \$a,\ \texttt{name}: \$\mathit{name}\}\ \textbf{in}\ \$c, \\
& \{\texttt{type}: \texttt{shipping}\}\ \textbf{in}\ \$a
\end{array}
$$

3 UnCAL: An Internal Graph Algebra

UnQL is convenient and powerful but is not suitable to discuss bidirectional semantics on UnQL due to its complicated language constructs such as regular path pattern. To resolve this problem, we use the idea in (Buneman et al. 2000) to map UnQL to UnCAL, and then discuss bidirectional semantics on UnCAL. UnCAL is an internal graph algebra of a graph query

$$
\begin{array}{lll}
T ::= & \{\} & \text{(* one-node graph *)} \\
| & \{L : T\} & \text{(* graph with a single edge from root *)} \\
| & T \cup T & \text{(* union of two graphs *)} \\
| & \&x := T & \text{(* label the root node with input marker } x \text{ *)} \\
| & \&y & \text{(* graph with output marker } y \text{ *)} \\
| & () & \text{(* empty graph *)} \\
| & T \oplus T & \text{(* disjoint union *)} \\
| & T \mathbin{@} T & \text{(* append of two graphs *)} \\
| & \mathbf{cycle}(T) & \text{(* graph with cycles *)} \\
| & \$G & \text{(* variable reference *)} \\
| & \mathbf{if}\ L = L\ \mathbf{then}\ T\ \mathbf{else}\ T & \text{(* conditional *)} \\
| & \mathbf{rec}(\lambda(\$l, \$G).T)(T) & \text{(* application of structural recursion *)} \\
L ::= & \$l & \text{(* label variable reference *)} \\
| & \mathtt{a} & \text{(* label } (a \in \mathit{Label}) \text{ *)}
\end{array}
$$

Figure 7: Core UnCAL Language

language UnQL. An UnQL query program written by users is translated into well-defined UnCAL expressions. In this section, after explaining how to represent structural recursion in UnCAL with new graph constructors, we show that any UnQL program can be automatically mapped to structural recursions in UnCAL.

3.1 Structural Recursion in UnCAL

The core of the UnCAL language is summarized in Figure 7. In addition to the graph constructors, UnCAL provides structural recursion, a general way to manipulate graphs.

Graph Constructors There are nine data constructors which are used to build arbitrary graphs (Buneman et al. 2000).

- $\{\}$ (one-node graph): it constructs a graph with a single node without edges.
- $\{l : G\}$ (one-edge-connected graph): it constructs a graph with the root pointing to the root of the graph G through the edge l.
- $G_1 \cup G_2$ (union of graphs): it unifies two graphs by creating a new root and connect it to the roots of G_1 and G_2 using ε-edges.
- $\&x := G$ (graph with input marker): it adds some input marker to the root of G.
- $\&y$ (output node): it constructs a graph with a single node marked with one output marker.

- () (empty graph): it constructs an empty graph which has neither node nor edge.
- $G_1 \oplus G_2$ (disjoint union of graphs): it constructs a graph by componentwise union.
- $G_1 \mathbin{@} G_2$ (append of graphs): it appends two graphs by connecting the output nodes of G_1 with corresponding input nodes of G_2 with ε-edges.
- **cycle**(G) (cyclic graph): it connects the input nodes with the output nodes of G to form cycles.

The formal definition of the semantics of these constructors can be found in Section 4. It is worth noting that this set of constructors are powerful enough to describe any unordered graphs.

Example 5. The graph in Figure 2(a) can be constructed as follows (though not uniquely).

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

We will use the following two abbreviations: $\{l_1 : G_1, \ldots, l_n : G_n\}$ for $\{l_1 : G_1, \} \cup \ldots \cup \{l_n : G_n\}$ and $(t_1, \ldots, t_n)$ for $t_1 \oplus \ldots, \oplus t_n$. Thus, the above UnCAL program becomes

$$\begin{aligned}\&z @ \mathbf{cycle}(&\&z := \{\mathtt{a} : \{\mathtt{a} : \&z_1\}, \mathtt{b} : \{\mathtt{a} : \&z_1\}, \mathtt{c} : \&z_2\}, \\ &\&z_1 := \{\mathtt{d} : \{\}\}, \\ &\&z_2 := \{\mathtt{c} : \&z_2\}).\end{aligned}$$

□

It is worth remarking that not all these constructors are required to transform UnQL queries to UnCAL terms. In fact, the constructor **cycle** is not required.

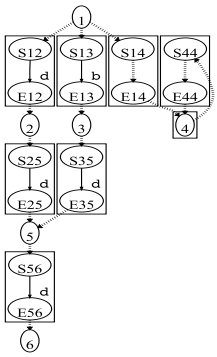
Structural recursion in UnCAL Any structural recursion in UnQL

$$\mathbf{let}\ \ \mathbf{sfun}\ \ f\ \{\$l : \$G\} = t \odot f(\$G)\ \ \mathbf{in}\ \ f(t')$$

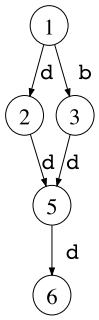
can be described by **rec** as

$$\&z_1 \mathbin{@} (\mathbf{rec}(\lambda(\$l, \$G).(\&z_1 := t \odot \&z_1))(t').$$

Generally, all branches in the definition of **sfun** have to be translated into **if** branches in UnCAL.



(a) Before removing ε-edges



(b) After removing ε-edges

Figure 8: Bulk Semantics in UnCAL

Example 6. The UnQL structural recursion $a2d\_xc$ given in Example 4 above is represented by

$$\&z_1 \mathbin{@} (\mathbf{rec}(\lambda(\$l, \$G').\, \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ (\&z_1 := \{\mathtt{d} : \&z_1\}) \\ \mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ (\&z_1 := \{\varepsilon : \&z_1\}) \\ \mathbf{else}\ (\&z_1 := \{\$l : \&z_1\}))(\$G))$$ □

For mutually defined functions, we can merge them into one **rec** construct by the tupling transformation (Hu et al. 1997).

3.2 Bulk Semantics of Structural Recursion

Structural recursion has two equivalent semantics under the graph bisimulation: *bulk* semantics and *recursive* semantics. The former is the prominent feature of structural recursion, whereas the latter is the usual recursive semantics with memoization. Informally, the bulk semantics for $\mathbf{rec}(\lambda(\$l, \$G).t)(G')$ is that $\lambda(\$l, \$G).t$ is applied independently on all edges of graph G', then the results are joined together with ε-edges (as in the @ operation).

Example 7. Consider to apply the structural recursive function $a2d\_xc$ to the graph in Figure 2(a). Applying the function to each edge from i to j gives a subgraph containing a graph with an edge from Sij to Eij (where the dotted edge denotes an ε-edge), then marking the root with an input marker, and finally joining the nodes with ε-edges according to the original graph shape and input/output markers yields the graph in Figure 8(a), which is equivalent to the graph in Figure 8(b) if we remove all ε-edges. □

3.3 From UnQL to UnCAL

Every UnQL expression can be mapped to structural recursion in UnCAL, which has been shown in (Buneman et al. 2000). We first informally explain

this mapping transformation, and then clarify the property of the obtained UnCAL programs that will serve as the target of our bidirectionalization in Section 4.

As we have shown that structural recursion in UnQL can be mapped to that in UnCAL before, to show that every UnQL expression can be mapped to structural recursion in UnCAL, it is sufficient to show that the select-where expression can be expressed in terms of structural recursion (in UnQL). This can be achieved by three steps. First, we can unnest patterns in the where-clause such that each pattern is in the simple form of $\{Lp : \$G\}\,\mathbf{in}\,\$G$ with the following three rules.

$$
\begin{array}{l}
\mathbf{where}\ \{Lp_1 : Gp_1, ..., Lp_n : Gp_n\}\,\mathbf{in}\,\$G \\
\quad \longrightarrow \mathbf{where}\ \{Lp_1 : Gp_1\}\,\mathbf{in}\,\$G, \ldots, \{Lp_n : Gp_n\}\,\mathbf{in}\,\$G \\
\mathbf{where}\ \{Lp : \mathtt{a}\}\,\mathbf{in}\,\$G \\
\quad \longrightarrow \mathbf{where}\ \{Lp : \$G'\}\,\mathbf{in}\,\$G, \{\mathtt{a} : \$G''\}\,\mathbf{in}\,\$G' \\
\mathbf{where}\ \{Lp : Gp\}\,\mathbf{in}\,\$G \\
\quad \longrightarrow \mathbf{where}\ \{Lp : \$G'\}\,\mathbf{in}\,\$G, Gp\,\mathbf{in}\,\$G'
\end{array}
$$

Note that $\$G'$ and $\$G''$ are fresh variable names in the above rules.

Then, an UnQL query with simple patterns (not regular path patterns) can be translated into structural recursions with condition in UnQL.

$$
\begin{array}{l}
\mathbf{select}\ T\ \mathbf{where}\ \ \longrightarrow T \\
\mathbf{select}\ T\ \mathbf{where}\ \{Lp : \$G_1\}\,\mathbf{in}\,\$G_2, \mathit{rest} \\
\quad \longrightarrow \mathbf{let\,sfun}\ f\ \{Lp : \$G_1\} = \mathbf{select}\ T\ \mathbf{where}\ \mathit{rest} \\
\qquad \mathbf{in}\ f\ (\$G_2) \\
\mathbf{select}\ T\ \mathbf{where}\, L_1 = L_2, \mathit{rest} \\
\quad \longrightarrow \mathbf{if}\ L_1 = L_2\ \mathbf{then\,select}\ T\ \mathbf{where}\ \mathit{rest}\ \mathbf{else}\ \{\}
\end{array}
$$

When the patterns are regular path patterns, they are translated into structural recursions. The idea of translation is to express a regular path pattern as an NFA (Non-deterministic Finite Automaton), associate a function to each state, and produce a mutually defined structural recursion according to the transition of the NFA.

As a simple example, consider the regular path pattern `_*.a.b`, an equivalent non-deterministic automaton[1] has five states and the following transitions :

$$
s_1 \xrightarrow{Any} s_4, s_1 \xrightarrow{Any} s_5, s_1 \xrightarrow{a} s_3, s_3 \xrightarrow{b} s_2, s_4 \xrightarrow{a} s_3,
$$

$$
s_5 \xrightarrow{Any} s_4, s_5 \xrightarrow{Any} s_5, s_5 \xrightarrow{a} s_3
$$

[1]There are several equivalent automata, of course.

The initial state is s_1 and the terminal state is s_2. So, the following mutual structural recursion can be defined.

$$\begin{array}{ll}
\textbf{sfun}\ f_1\{\mathtt{a} : \$G\} & = f_3(\$G) \\
\quad \mid f_1\{\$l : \$G\} & = f_4(\$G) \cup f_5(\$G) \\
\textbf{sfun}\ f_2\{\$l : \$G\} & = \{\} \\
\textbf{sfun}\ f_3\{\mathtt{b} : \$G\} & = f_2(\$G) \cup \$G \\
\quad \mid f_3\{\$l : \$G\} & = \{\} \\
\textbf{sfun}\ f_4\{\mathtt{a} : \$G\} & = f_3(\$G) \\
\quad \mid f_4\{\$l : \$G\} & = \{\} \\
\textbf{sfun}\ f_5\{\mathtt{a} : \$G\} & = f_3(\$G) \\
\quad \mid f_5\{\$l : \$G\} & = f_4(\$G) \cup f_5(\$G)
\end{array}$$

In general mutually defined recursive functions are translated into a single recursive function(Hu et al. 1997). For UnQL/UnCAL, the new single recursive function is defined using markers. Each marker corresponds to a function that is mutually defined with others. Output markers are used instead of recursive calls in the body of the function. This new function returns a tuple of results of each function, represented by markers, that is mutually defined with others.

For example, consider the following mutual recursive definition shown in Section 2.2;

$$\begin{array}{ll}
\textbf{sfun}\ \ h\ \{\mathtt{b} : \$G\} & = \{\mathtt{b} : a2e(\$G)\} \\
\quad \mid\ h\ \{\$l : \$G\} & = h(\$G) \\
\textbf{sfun}\ \ a2e\ \{\mathtt{a} : \$G\} & = \{\mathtt{e} : a2e(\$G)\} \\
\quad \mid\ a2e\ \{\$l : \$G\} & = \{\$l : a2e(\$G)\}
\end{array}$$

This function is translated into single recursive function as follows;

$$\textbf{sfun}\ \ f\ \{\$l : \$G\} = g(\$l, \$G)\ @\ \ f(\$G)$$

where $g(\$l, \$G)$ is defined as follows;

$$\begin{array}{ll}
g(\mathtt{a}, \$G) & = ((\&z_1 := \&z_1) \quad\ \oplus (\&z_2 := \{\mathtt{e} : \&z_2\})) \\
g(\mathtt{b}, \$G) & = ((\&z_1 := \{\mathtt{b} : \&z_2\}) \oplus (\&z_2 := \{\mathtt{b} : \&z_2\})) \\
g(\$l, \$G) & = ((\&z_1 := \&z_1) \quad\ \oplus (\&z_2 := \{\$l : \&z_2\}))
\end{array}$$

Note that markers $\&z_1$ and $\&z_2$ correspond to functions h and $a2e$, respectively. Recall that an output node vcan be regarded as a "context-hole" of grapshs and append of graphs ($t_1 @ t_2$) plugs the input nodes in t_2 into their corresponding output nodes in t_1. Recall also that disjoint union ($t_1 \oplus t_2$) performs componentwize union of t_1 and t_2.

As shown in Section 3.1, the above form of single structural recursive function f is descrived by **rec** as

$$\begin{array}{rl}
\textbf{rec}(\lambda(\$l, \$G).\ \textbf{if}\ \$l = \mathtt{a}\ \textbf{then} & ((\&z_1 := \&z_1) \quad\ \oplus (\&z_2 := \{\mathtt{e} : \&z_2\})) \\
\textbf{else if}\ \$l = \mathtt{b}\ \textbf{then} & ((\&z_1 := \{\mathtt{b} : \&z_2\}) \oplus (\&z_2 := \{\mathtt{b} : \&z_2\})) \\
\textbf{else} & ((\&z_1 := \&z_1) \quad\ \oplus (\&z_2 := \{\$l : \&z_2\}))
\end{array}$$

Note that append operator @ used in the form using **sfun** is dropped.

Lemma 1 (Target UnCAL)**.** *The mapping algorithm from UnQL to UnCAL produces an UnCAL expression with the following syntactic properties.*

- *For recursion application* $\mathbf{rec}(\lambda(\$l, \$G).t)(t')$*, the argument* t' *should be a variable, which implies no intermediate graphs.*
- *For disjoint union* $(t_1 \oplus t_2)$*,* t_1 *and* t_2 *should be in the form of* $\&x := t'$*.*
- *For append* $(t_1 \ @ \ t_2)$*, the left operand should be an output marker and the right operand should be an application of structural recursion* $(\&y \ @ \mathbf{rec}(\lambda(\$l, \$G).\ T)(T))$*.*
- *No* $\mathbf{cycle}(t)$ *expression should appear.*

4 Bidirectionalization of UnCAL

In this section, we show that an UnCAL program can not only be evaluated forwardly as usual, but also be evaluated backwardly to reflect updates from the result to the source. We shall give a bidirectional semantics for each construct of UnCAL. Note that the backward semantics mentioned first can cope with general in-place updating and deletion, but not insertion. We extend the semantics of **if** and **rec** to cope with insertion in Section 4.5, and finally prove well-behavedness of our bidirectional semantics.

4.1 Bidirectional Properties

Forward evaluation (often called *get* in the literature) of an UnCAL term t computes a result graph $G = [\![t]\!]^\rho$ (*view*), under a variable binding environment ρ (*input*) which is a mapping from variables to graphs. Backward evaluation (or *put*) goes backward. Given an original input environment ρ and a possibly modified view graph G', it computes an updated environment $\rho' = \langle\!\langle t \rangle\!\rangle^\rho_{G'}$.

The following two important properties define the well-behavedness of a pair of forward and backward evaluations, which are essentially the same as those in *lenses* (Foster et al. 2005).

$$[\![t]\!]^\rho = G \quad \text{implies} \quad \langle\!\langle t \rangle\!\rangle^\rho_G = \rho \qquad (\textsc{GetPut})$$

$$\langle\!\langle t \rangle\!\rangle^\rho_{G'} = \rho' \quad \text{implies} \quad [\![t]\!]^{\rho'} = G' \qquad (\textsc{PutGet})$$

The (GetPut) property says that no change on the view graph should give no change on the environment, while the (PutGet) property says that the backward computation computes a new environment ρ' from G' in such a way that applying the forward computation under ρ' again should give the same G'.

4.2 Embedding Trace Information in Structured Node IDs

Different from unidirectional computation, the forward evaluation in the context of bidirectional computation should keep trace information for later backward evaluation. Our forward evaluation rules will refine (extend) the original semantics of UnCAL. Basically, each graph constructor expression is straightforwardly evaluated to the graph as it denotes. However, not only constructing the output graph structure, we also embed some "trace" information in each node of the output graph to guarantee the correct backward evaluation. The nodes of the output graph are identified by what we call the *Structured IDs* that describe where the nodes came from.

Consider the upper part of Figure 9, which demonstrates evaluation of $G_1 \cup G_2$ (a union of two graphs). For later backward evaluation, we need to decompose the result graph into two while keeping the original correspondence. And this is difficult because our graphs are unordered. Our idea is to assign structured IDs to the nodes of the output graph so that together with information of the original inputs G_1 and G_2 the output graph can be decomposed.

Formally, the structured ID is defined as follows.

$$
\begin{aligned}
\mathit{StructuredID} ::= {} & \mathit{OriginalID} \\
\mid {} & \mathsf{InC}\ \mathit{CodePos} \\
\mid {} & \mathsf{InC}_{\cup}\ \mathit{CodePos}\ \mathit{Marker} \\
\mid {} & \mathsf{Hub}_{\mathit{CodePos}}\ \mathit{StructuredID}\ \mathit{Marker} \\
\mid {} & \mathsf{FrE}_{\mathit{CodePos}}\ \mathit{StructuredID}\ \mathit{Edge}
\end{aligned}
$$

where $\mathit{Edge} = \mathit{StructuredID} \times \mathit{Label} \times \mathit{StructuredID}$, *CodePos* is the set of position identifiers that are uniquely assigned to all syntactic nodes of the UnCAL program, and *OriginalID* is the set of identifiers that are uniquely assigned to all nodes of the input graph. All the structured IDs are to denote the output nodes[2] of UnCAL transformations. The ID ($\mathsf{InC}\ p$) denotes an output node created from a constant expression like $\{\}$ or $\{a : \{\}\}$. The value p is the unique identifier assigned to each syntactic node of UnCAL expressions. For example, there are two syntactic nodes in the constant expression $\{a : \{\}\}$: the whole expression itself and the subexpression $\{\}$. They are assumed to have distinct position identifiers, say, p_1 and p_2 respectively, and the corresponding output nodes are named distinctly as ($\mathsf{InC}\ p_1$) and ($\mathsf{InC}\ p_2$). The ID ($\mathsf{InC}_{\cup}\ p\ \&m$) is similar to ($\mathsf{InC}\ p$) but also indexed with the marker $\&m$. This type of ID is used for representing an output graph of an union-expression, as explained later. Last but not least, the IDs of the from ($\mathsf{FrE}_p\ i\ e$) and ($\mathsf{Hub}_p\ v\ m$) are used for constructing the output nodes of the structural recursion: **rec**. Recall that the bulk semantics of the structural recursion first evaluates the recursion body at every edge e

[2]Input/output here denotes input/output of transformations. We say "input/output *marker* node" for node with markers.

of the input graph, and then joins the results at the point of input/output markers. Now, suppose the evaluation of the body expression at the edge e generated an output node with ID i. We augment such output node with the ID $(\mathsf{FrE}_p\ i\ e)$ where p is the position of the **rec** expression itself. The ID $(\mathsf{Hub}_p\ v\ \&m)$ denotes the output node used for joining the results of bulk evaluation of structural recursion at the position p.

It is worth noting that our assignment of structured IDs makes an ID independent of actual evaluation order. In this sense, the ID assignment strategy is *functional* and side-effect free. This fact helps to simplify the our bidirectional semantics.

4.3 Issues on Backward Evaluation

Similarly to the usual *put*, the backward semantics $\rho' = \langle\!\langle t \rangle\!\rangle^{\rho}_{G'}$ requires the original input environment ρ as well as the modified target G'. The original input is used for decomposing the target so as to define the backward semantics inductively. For example, to compute $\langle\!\langle t_1 \cup t_2 \rangle\!\rangle^{\rho}_{G'}$, we first decompose G' to two parts G'_1 and G'_2 and then inductively compute $\rho'_1 = \langle\!\langle t_1 \rangle\!\rangle^{\rho}_{G'_1}$ and $\rho'_2 = \langle\!\langle t_2 \rangle\!\rangle^{\rho}_{G'_2}$. For this decomposition, we need the original, unmodified version of G'_1 and G'_2, i.e., $G_1 = [\![t_1]\!]^{\rho}$ and $G_2 = [\![t_2]\!]^{\rho}$. This is the first reason why we take the original ρ as the input here.

Another reason is to use the original environment for merging the updated environments. In general, multiple environment produced by backward evaluation are merged by $\uplus_{\rho}$, with original environment ρ considered. Conflicts are resolved in this operator as follows: if no difference between the original environment and the result of backward evaluations is detected, it simply returns the original environment. Otherwise, the result of backward evaluation takes precedence, provided that (possibly multiple) result(s) are consistent with the original environment. It fails if none of the condition hold.

4.4 Formal Definition of Bidirectional Semantics

In this section, we give formal definition of bidirectional semantics for UnCAL, where only in-place updating and deletion is considered. We will extend the bidirectional semantics so that insertion can be coped with in Section 4.5.

4.4.1 Bidirectional Evaluation of Simple Expressions

We propose a set of rules for describing both forward and backward evaluations of expressions in UnCAL, guaranteeing that the forward and backward evaluations satisfy the bidirectional properties. We start by showing how graph constructors can be computed forwardly and backwardly thanks

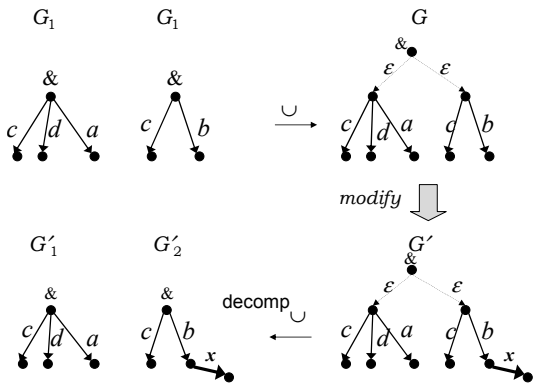


Figure 9: Example for Bidirectional Computation of Union

to the ε-edges and the structured IDs. Then we show that this bidirectional evaluation of graph constructors can be extended to bidirectional evaluation of UnCAL expressions except for the structural recursions. Finally, we tackle the problem of bidirectional semantics for structural recursions. In the following definitions, p denotes position identifier.

Nullary constructors {} (single node graph), &*y* (a node with output marker) and () (empty graph) construct constant graphs in their forward computation. For backward computation, they are constant and hence accept no modification on the result view.

$$\begin{aligned}
[\![\{\}]\!]^{\rho} &= (\{\mathsf{InC}\ p\}, \emptyset, \{(\&, \mathsf{InC}\ p)\}, \emptyset) \\
\langle\!\langle\{\}\rangle\!\rangle^{\rho}_{G'} &= \rho \qquad \text{if } G' = [\![\{\}]\!]^{\rho}
\end{aligned}$$

$$\begin{aligned}
[\![\&m]\!]^{\rho} &= (\{\mathsf{InC}\ p\}, \emptyset, \{(\&, \mathsf{InC}\ p)\}, \{\mathsf{InC}\ p, \&m\}) \\
\langle\!\langle\&m\rangle\!\rangle^{\rho}_{G'} &= \rho \qquad \text{if } G' = [\![\&m]\!]^{\rho}
\end{aligned}$$

$$\begin{aligned}
[\![()]\!]^{\rho} &= (\emptyset, \emptyset, \emptyset, \emptyset) \\
\langle\!\langle()\rangle\!\rangle^{\rho}_{G'} &= \rho \qquad \text{if } G' = [\![()]\!]^{\rho}
\end{aligned}$$

{_:_} prepends an edge on top of the root of the second operand graph in its forward computation. 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').

$$
\begin{array}{l}
[\![\{t_1 : t_2\}]\!]^\rho \quad = \\
\quad (\{\mathsf{InC}\ p\} \cup V_2, E_1 \cup E_2, \{(\&, \mathsf{InC}\ p)\}, O_2) \\
\quad\ \text{where}\ \ E_1 \qquad\qquad\ = \{(\mathsf{InC}\ p, l_1, I_2(\&))\} \\
\qquad\qquad (V_2, E_2, I_2, O_2) = [\![t_2]\!]^\rho \\
\qquad\qquad l_1 \qquad\qquad\ = [\![t_1]\!]^\rho
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle \{t_1 : t_2\} \rangle\!\rangle^\rho_{G'} = \langle\!\langle t_1 \rangle\!\rangle^\rho_{l_1'} \uplus_\rho \langle\!\langle t_2 \rangle\!\rangle^\rho_{G_2'} \\
\quad \text{where}\ \ l_1 \qquad\ = [\![t_1]\!]^\rho \\
\qquad\qquad G_2 \qquad\ = [\![t_2]\!]^\rho \\
\qquad\qquad (l_1', G_2') = \mathsf{decomp}_{\{l_1 : G_2\}}(G')
\end{array}
$$

Here, the decomposition function is defined as follows:

$$
\begin{array}{l}
\mathsf{decomp}_{\{l_1 : G_2\}}(G') = \\
\quad (l_1', (V' \setminus \{r'\}, E' \setminus \{e'\}, \{(\&, v)\}, O')) \\
\quad \text{where}\ \ (V_2, E_2, \{(\&, v)\}, O_2) = G_2 \\
\qquad\qquad (V', E', \{(\&, r')\}, O') = G' \\
\qquad\qquad e' = \text{ the unique edge in } E' \\
\qquad\qquad\qquad\quad \text{of the form } (r', l_1', v).
\end{array}
$$

The modified output graph G' is decomposed into its unique root edge $e' = (r', l_1', v)$ and the rest of the graph rooted at v. If G' have 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.

$\cup$ unifies two graphs by connecting input nodes in two graphs with matching markers using ε-edges in forward computation. Its backward computation removes these edges and restores the operand graphs while taking modifications to them into account. Consider an example in Figure 9. The edge labeled x (bold arrow) is added on the view. Backward computation removes these ε-edges to restore original input nodes, and compute modified operands by collecting reachable parts from each of the input nodes[3]. The added edge is reachable from the input node of G_2', so the modification belongs to the second operand. For well-behavedness, backward computation checks if ε-edges from the input nodes are changed, and fails if that is the case. Note that the forward computation could have been glued the two input nodes together, but doing so would make splitting of the view difficult, since both of the

[3]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.

operands can be reachable from the glued node.

$$
\begin{aligned}
[\![t_1 \cup t_2]\!]^\rho \quad &= (V, E_\mathrm{u} \cup E_1 \cup E_2, I_\mathrm{u}, O_1 \cup O_2) \\
\text{where } V &= V_\mathrm{u} \cup V_1 \cup V_2 \\
(V_1, E_1, I_1, O_1) &= [\![t_1]\!]^\rho \\
(V_2, E_2, I_2, O_2) &= [\![t_2]\!]^\rho \\
M &= \mathsf{inMarkers}(t_1) = \mathsf{inMarkers}(t_2) \\
V_\mathrm{u} &= \{\mathsf{InC}_\cup\ p\ \&m \mid \&m \in M\} \\
E_\mathrm{u} &= \{(\mathsf{InC}_\cup\ p\ \&m, \varepsilon, v) \mid (\&m, v) \in I_1\} \\
&\quad \cup\ \{(\mathsf{InC}_\cup\ p\ \&m, \varepsilon, v) \mid (\&m, v) \in I_2\} \\
I_\mathrm{u} &= \{(\&m, \mathsf{InC}_\cup\ p\ \&m) \mid \&m \in M\}
\end{aligned}
$$

$$
\begin{aligned}
\langle\!\langle t_1 \cup t_2 \rangle\!\rangle^\rho_{G'} &= \langle\!\langle t_1 \rangle\!\rangle^\rho_{G'_1} \uplus_\rho \langle\!\langle t_2 \rangle\!\rangle^\rho_{G'_2} \\
\text{where } G_1 &= [\![t_1]\!]^\rho \\
G_2 &= [\![t_2]\!]^\rho \\
(G'_1, G'_2) &= \mathsf{decomp}_{G_1 \cup G_2}(G')
\end{aligned}
$$

The decomposition function ensures that the root 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. $G_1 \setminus G_2$ denotes componentwise set difference.

$$
\begin{aligned}
&\mathsf{decomp}_{G_1 \cup G_2}(G') = \\
&\quad \big(\mathsf{xreachable}(G'_1, G_1), \mathsf{xreachable}(G'_2, G_2)\big) \\
&\quad \text{where } (V', E', I', O') = G' \\
&\qquad (V_i, E_i, I_i, O_i) = G_i \\
&\qquad G'_i = \mathsf{reachable}((V', E', I_i, O')) \\
&\quad \text{satisfying} \\
&\qquad \mathsf{inMarkers}(G_1) = \mathsf{inMarkers}(G_2) \\
&\qquad \forall \&m \in \mathsf{inMarkers}(G_1), (\&m, r') \in I', \\
&\qquad (r', \varepsilon, v') \in E' : \\
&\qquad\quad (\&m, v') \in I_1 \cup I_2
\end{aligned}
$$

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

$\oplus$ performs componentwise union in its forward computation. For backward computation, it is like $\cup$, except that no ε-edge is involved.

$$\begin{aligned}
&[\![t_1 \oplus t_2]\!]^\rho = (V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2, O_1 \cup O_2) \\
&\quad \text{where } (V_1, E_1, I_1, O_1) = [\![t_1]\!]^\rho \\
&\qquad\quad (V_2, E_2, I_2, O_2) = [\![t_2]\!]^\rho
\end{aligned}$$

$$\begin{aligned}
&\langle\!\langle t_1 \oplus t_2 \rangle\!\rangle^\rho_{G'} = \langle\!\langle t_1 \rangle\!\rangle^\rho_{G'_1} \uplus_\rho \langle\!\langle t_2 \rangle\!\rangle^\rho_{G'_2} \\
&\text{where } G_i = [\![t_i]\!]^\rho \\
&\qquad (G'_1, G'_2) = \mathsf{decomp}_{G_1 \oplus G_2}(G') \\
&\qquad \mathsf{decomp}_{G_1 \oplus G_2}(G') = \mathsf{decomp}_{G_1 \cup G_2}(G') \\
&\qquad\qquad \text{without } \textit{satisfying} \\
&\qquad\qquad \text{condition}
\end{aligned}$$

@ appends two graphs by connecting the output nodes of the left operand with corresponding input nodes of the right operand with ϵ-edges in its forward computation. Currently, we allow this operator to be used only for the projection of one component of the result of structural recursion by $\&z_i @ \mathbf{rec}(t_b)(t_a)$. Note that @ may introduce unreachable part in the second argument, because of unmatched I/O nodes. 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. Note that users operating on the view do not change these unreachable parts since these parts are invisible for the users.

$$\begin{aligned}
&[\![\&m @ t_2]\!]^\rho = (V, E_@ \cup E_2, \{(\&, \mathsf{InC}\ p)\}, O_2) \\
&\quad \text{where } V = \{\mathsf{InC}\ p\} \cup V_2 \\
&\qquad\quad V_1 = \{\mathsf{InC}\ p\} \\
&\qquad\quad (V_2, E_2, I_2, O_2) = [\![t_2]\!]^\rho \\
&\qquad\quad E_@ = \{(\mathsf{InC}\ p, \varepsilon, v) \mid (\&m, v) \in I_2\}
\end{aligned}$$

$$\begin{aligned}
&\langle\!\langle \&m @ t_2 \rangle\!\rangle^\rho_{G'} = \langle\!\langle t_2 \rangle\!\rangle^\rho_{G'_2} \\
&\quad \text{where } (V', E', I', O') = G' \\
&\qquad\quad (V_2, E_2, I_2, O_2) = [\![t_2]\!]^\rho \\
&\qquad\quad E_@ = \{(\mathsf{InC}\ p, \varepsilon, v) \mid (\&m, v) \in I_2\} \\
&\qquad\quad G'_2 = (V' \setminus \{\mathsf{InC}\ p\}, E' \setminus E_@, I_2, O_2)
\end{aligned}$$

(:=) distributes the marker on the left operand to each of the input marker of the graph in the right operand, using the Skolem function "." that satisfies $(\&x.\&y).\&z = \&x.(\&y.\&z)$ (associativity) and $\&.\&x = \&x.\& = \&x$ (left and right identity) in its forward computation. Backward computation, "peals off" the marker on the left hand side from each

of the input markers in G' at the front.

$$\begin{aligned}
[\![\&m := t_1]\!]^\rho &= (V_1, E_1, I, O_1) \\
&\quad \text{where } (V_1, E_1, I_1, O_1) = [\![t_1]\!]^\rho \\
&\qquad I = \{(\&m.\&x, v) \mid (\&x, v) \in I_1\}
\end{aligned}$$

$$\begin{aligned}
\langle\!\langle \&m := t_1 \rangle\!\rangle^\rho_{G'} &= \langle\!\langle t_1 \rangle\!\rangle^\rho_{G'_1} \\
&\quad \text{where } G'_1 = (V', E', I'_1, O') \\
&\qquad (V', E', I', O') = G' \\
&\qquad I'_1 = \{(\&x, v) \mid (\&m.\&x, v) \in I'\}
\end{aligned}$$

(**$v**) looks up variable binding from environment ρ. Backward computation updates the binding.

$$\begin{aligned}
[\![\$v]\!]^\rho &= \rho(\$v) \\
\langle\!\langle \$v \rangle\!\rangle^\rho_{G'} &= \rho[\$v \leftarrow G']
\end{aligned}$$

(**if**) first evaluates the conditional expression b and by the result it evaluates either the **then** branch or the **else** branch in the forward computation. For the backward computation, modification to the environment as a result of modification to the view may change the branching behavior (result of b). In order to make sure that well-behavedness still holds, backward semantics chooses the branch in which a result of another forward computation on the condition b agrees. To cope with possible non-determinism where both branches agree, the branch taken in the forward computation takes precedence. If neither of the branches agree, the backward computation fails.

$$\begin{aligned}
[\![\textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2]\!]^\rho &= [\![t_i]\!]^\rho \\
&\quad \text{where } \; i = (\text{if } [\![b]\!]^\rho = \mathsf{true} \text{ then 1 else 2})
\end{aligned}$$

$$\begin{aligned}
\langle\!\langle \textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2 \rangle\!\rangle^\rho_{G'} &= \rho'_{i'} \\
&\quad \text{where } \; \rho'_1 = \langle\!\langle t_1 \rangle\!\rangle^\rho_{G'} \\
&\qquad \rho'_2 = \langle\!\langle t_2 \rangle\!\rangle^\rho_{G'} \\
&\qquad i \;\; = (\text{if } [\![b]\!]^\rho = \mathsf{true} \text{ then 1 else 2}) \\
&\qquad i' \;\; = \begin{cases} i & \text{if } \; [\![b]\!]^{\rho'_i} = [\![b]\!]^\rho \\ 3 - i & \text{if } \; [\![b]\!]^{\rho'_{3-i}} \neq [\![b]\!]^\rho \end{cases}
\end{aligned}$$

It is worth noting that we have the following properties on the internal graph representation when modification is applied on the view: (1) No ε-edges are added or deleted.; (2) Markers are not added, deleted or changed.; and (3) Unreachable parts are not modified. These properties are important for (GetPut) and (PutGet) properties to hold. Besides, Property (2) is a natural consequence of the rule for the output marker constructor $\&y$.

$$\mathsf{fwd\_eachedge}(\rho, t_b, G_a{=}(\_, E_a, \_, \_)) = \left\{ (e, [\![t_b]\!]^{\rho_e}) \;\middle|\; \begin{array}{l} e \in E_a,\ \mathit{label}(e) \neq \varepsilon, \\ \rho_e = \rho\{\$l \mapsto \mathit{label}(e), \$g \mapsto \mathit{subgraph}(G_a, e)\} \end{array} \right\}$$

$$\mathsf{compose}_{\mathbf{rec}}(\mathcal{G}, (V_a, E_a, I_a, O_a), M) = (V_{\mathrm{body}} \cup V_{\mathrm{hub}}, E_{\mathrm{body}} \cup E_{\mathrm{spoke}} \cup E_{\mathrm{eps}}, I, O)$$

$$\begin{array}{rl}
\text{where } V_{\mathrm{body}} & = \{\mathsf{FrE}_p\ e\ v \mid (e, (V_e, _, _, _)) \in \mathcal{G}, v \in V_e\} \\
E_{\mathrm{body}} & = \{(\mathsf{FrE}_p\ e\ u, l, \mathsf{FrE}_p\ e\ v) \mid (e, (_, E_e, _, _)) \in \mathcal{G}, (u, l, v) \in E_e\} \\
V_{\mathrm{hub}} & = \{\mathsf{Hub}_p\ v\ \&m \mid v \in V_{\mathrm{a}}, \&m \in M\} \\
E_{\mathrm{spoke}} & = \{(\mathsf{Hub}_p\ v\ \&m, \varepsilon, \mathsf{FrE}_p\ e\ u) \mid \\
& \qquad \&m \in M, (e = (v, _, _), (_, _, I_e, _)) \in \mathcal{G}, (\&m, u) \in I_e\} \\
& \cup \{(\mathsf{FrE}_p\ e\ u, \varepsilon, \mathsf{Hub}_p\ v\ \&m) \mid \\
& \qquad \&m \in M, (e = (_, _, v), (_, _, _, O_e)) \in \mathcal{G}, (u, \&m) \in O_e\} \\
E_{\mathrm{eps}} & = \{(\mathsf{Hub}_p\ v\ \&m, \varepsilon, \mathsf{Hub}_p\ u\ \&m) \mid (v, \varepsilon, u) \in E_{\mathrm{a}}, \&m \in M\} \\
I & = \{(\&n.\&m, \mathsf{Hub}_p\ v\ \&m) \mid (\&n, v) \in I_{\mathrm{a}}, \&m \in M\} \\
O & = \{(\mathsf{Hub}_p\ v\ \&m, \&n.\&m) \mid (v, \&n) \in O_{\mathrm{a}}, \&m \in M\}
\end{array}$$

Figure 10: Core of the Forward Semantics of **rec** at Code Position p

Property (1) is required for the consistent decomposition of targets. Property (3) is required to avoid unnecessary failure of backward computation by interference of unreachable parts on reachable parts. They are also natural from the user's point of view since these components (ε-edge, markers and unreachable parts) are all invisible to users.

4.4.2 Bidirectional Evaluation of Structural Recursion

In this section, we give bidirectional semantics for structural recursion **rec**. For forward computation, **rec** expressions are interpreted in *bulk semantics* as follows:

$$\begin{array}{l}
[\![\mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}})]\!]^{\rho} = \\
\qquad \mathsf{compose}_{\mathbf{rec}}(\mathsf{fwd_eachedge}(\rho, t_b, G_a), G_a, M) \\
\quad \text{where } M = \mathsf{inMarker}(t_{\mathrm{b}}) \cup \mathsf{outMarker}(t_{\mathrm{b}}) \\
\qquad\ \ G_a = [\![t_{\mathrm{a}}]\!]^{\rho}
\end{array}$$

where $\mathsf{fwd\_eachedge}$ and $\mathsf{compose}_{\mathbf{rec}}$ is defined in Figure 10. Intuitively, $\mathsf{fwd\_eachedge}$ evaluates the expression t_b at each edge e of the argument graph G_a and returns the set of result graphs. Then, $\mathsf{compose}_{\mathbf{rec}}$ glues all the graphs together along the structure of the input G_a.

Forward evaluation consists of the following steps. (1) Argument expression t_a is computed to produce graph G_a. (2) By the $\mathsf{fwd\_eachedge}$ auxiliary function, at every non-ε-edge e of G_a, new binding ρ_e is created so that $\$l$ points to the label of e (denoted by $\mathit{label}(e)$) and $\$g$ points to the subgraph (denoted by $\mathit{subgraph}(G_a, e)$) that is reachable from the target node of e. (3) Forward computation of the body expression t_b is conducted for each of these bindings. (4) By the $\mathsf{compose}_{\mathbf{rec}}$ auxiliary function, the results G_e are combined to produce the final result.

$$\mathsf{bwd\_eachedge}(\rho, t_b, \mathcal{G}') = \left\{ (e, \langle\!\langle e_b \rangle\!\rangle^{\rho_e}_{G'_e}) \;\middle|\; (e, G'_e) \in \mathcal{G',}\ \rho_e = \rho\{\$l \mapsto \mathit{label}(e), \$g \mapsto \mathit{subgraph}(e)\} \right\}$$

$$\mathsf{decomp}_{\mathbf{rec}}(\rho, t_b, (V', E', I', O'), G_a, M) = \left\{ (e, G'_e) \;\middle|\; \begin{array}{l} e \in E_a, \mathit{label}(e) \neq \varepsilon, \\ \rho_e = \rho\{\$l \mapsto \mathit{label}(e), \$g \mapsto \mathit{subgraph}(G_a, e)\} \\ V'_e = \{w \mid (\mathsf{FrE}_p\ e\ w) \in V'\}, \\ E'_e = \{(w_1, l, w_2) \mid (\mathsf{FrE}_p\ e\ w_1, l, \mathsf{FrE}_p\ e\ w_2) \in E'\}, \\ I'_e = \{(\&m, w) \mid (\mathsf{Hub}_p\ v\ \&m, \varepsilon, \mathsf{FrE}_p\ e\ w) \in E'\}, \\ O'_e = \{(w, \&m) \mid (\mathsf{FrE}_p\ e\ w, \varepsilon, \mathsf{Hub}_p\ v\ \&m) \in E'\}, \\ G'_e = (V'_e, E'_e, I'_e, O'_e) \end{array} \right\}$$

where $(V_a, E_a, I_a, O_a) = G_a$
$\mathcal{G} = \mathsf{fwd\_eachedge}(\rho, t_b, (V_a, E_a, I_a, O_a))$

$$\mathsf{merge}(\rho, t_a, (V_a, E_a, I_a, O_a), \mathcal{R}) = \langle\!\langle t_a \rangle\!\rangle^{\rho}_{G'_a} \uplus_\rho \biguplus \{\rho'_e \setminus \{\$l \mapsto \_\} \setminus \{\$g \mapsto \_\} \mid (e, \rho'_e) \in \mathcal{R}\}$$

where
$E_{\mathrm{eps}} = \{(u, \varepsilon, v) \mid (u, \varepsilon, v) \in E_a\}$
$(V''_e, E''_e) = (V'_e \cup \{u\}, E'_e \cup \{(u, \rho'_e(\$l), \mathsf{root}(I'_e))\})$
for each $e = (u, a, v) \in E_a$ with $a \neq \varepsilon$,
$(e, \rho'_e) \in \mathcal{R}, (V'_e, E'_e, I'_e, O'_e) = \rho'_e(\$g)$
$G'_a = (\bigcup V''_e, E_{\mathrm{eps}} \cup \bigcup E''_e, I_a, O_a)$

Figure 11: Core of the Backward Semantics of **rec** at Code Position p

The composition by $\mathsf{compose}_{\mathbf{rec}}$ produces two types of nodes. One is V_{body}, which is the set of nodes v of of each graph fragments G_e to be combined. Note that, however, in order to keep the traceability, we augment the structured node ID of v as $\mathsf{FrE}_p\ e\ v$ recording the edge code ID p of the **rec** expression and the input edge e where the node is created. Of course, in the case of nested **rec** expressions, the ID v from the result of body expression themselves may be structured already. Similarly, the set of edges E_{body} consists of edges from G_e with structured ID information added. The other type of nodes is V_{hub}, which is used as connecting points of G_e's. For instance, let $e_1 = (v, a, u)$ and $e_2 = (u, b, w)$ sharing the node u, and the **rec**ursion uses two markers $\&z_1$ and $\&z_2$. To connect G_{e_1} and G_{e_2} correctly, nodes with output marker $(x_1, \&z_1)$ in G_{e_1} must be identified with nodes with input marker $(\&z_1, x_2)$ in G_{e_2} (and similarly for $\&z_2$). To achieve this, we prepare a node $(\mathsf{Hub}_p\ v\ \&z_1)$ in V_{hub}, and connect all the output marker nodes of G_{e_1} and the input marker nodes of G_{e_2} using ε-edges, which are the E_{spoke} edges. Finally, to keep the ε-edges in the input graph unchanged, we connect by an ε-edge the hubs whose origin nodes are connected by an ε-edge; this is the set of edges E_{eps}.

Next, our backward semantics is defined as follows:

$$
\begin{aligned}
&\langle\!\langle \mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}}) \rangle\!\rangle^{\rho}_{G'} = \\
&\quad \mathsf{merge}(\rho, t_a, G_a, \\
&\qquad \mathsf{bwd\_eachedge}(\rho, t_b, G_a, \\
&\qquad\quad \mathsf{decomp}_{\mathbf{rec}}(\rho, t_b, G', G_a, M))) \\
&\text{where}\ \ M = \mathsf{inMarker}(t_{\mathrm{b}}) \cup \mathsf{outMarker}(t_{\mathrm{b}}) \\
&\qquad\ G_a = [\![t_{\mathrm{a}}]\!]^{\rho}
\end{aligned}
$$

Note the duality between the forward semantics. Backward semantics first decomposes by $\mathsf{decomp}_{\mathbf{rec}}$ the modified result graph G' into pieces of graphs, which is intuitively an inverse operation of $\mathsf{comp}_{\mathbf{rec}}$. Each element of the decomposition is the (possibly modified) subpart G'_e of G', which corresponds to the forward computation G_e. Then, in bwd_eachedge, we carry out backward computation on each edge and compute the updated environment ρ'_e. Finally, these environments are merged into the updated environment ρ' of the whole expression. The merge function does two works. First, by combing the information $\rho'_e(\$l)$ and $\rho'_e(\$g)$ from the updated environments, it computes the modified argument graph G'_a[4]. Then we inductively carry out backward evaluation on the argument expression to obtain another updated environment ρ'_a. This ρ'_a and all ρ'_es 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 extract from result graph G' nodes V'_e and edges E'_e that belonging to each edge e by matching structured ID $\mathsf{FrE}_p\ e\ \_$. Note that we require nodes that are newly inserted to the view also 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 $\mathsf{Hub}\ \_\ \_\ \&m$. Top-level constructors of structured ID are erased so that we can inductively compute the backward image from the body expression.

Note that semantics uses node identities in computation, while graph data model assumes value equivalence based on bisimulation. Update detection or conflict resolution in $\uplus_{\rho}$ basically uses node identities, however, bisimulation equivalence may be used to resolve conflicts at the graph level instead of per node and edge basis, to cope with complex updating on the view.

[4]Merging operation in $\bigcup$ does not consist of simple set unification operation. Detail is discussed in Section B in the appendix.

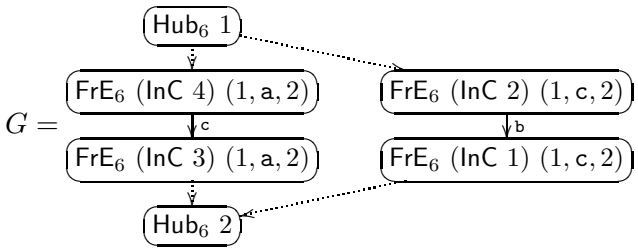
Example 8 (Edge Renaming)**.** The following transformation *a2b* replaces edge label `a` with `b` and leaves other labels unchanged.

$$
\begin{array}{l}
a2b = \\
\quad \mathbf{rec}(\lambda(\$l, \$G).\ \&z := \\
\qquad \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{b} : (\&z)_1\}_2 \\
\qquad \mathbf{else}\qquad\quad \{\$l : (\&z)_3\}_4)(\$db)_6
\end{array}
$$

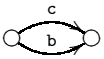
Above, expression t at code position p is written by t_p. Let $\$db$, which represents a source database, be the following graph.

$$s = \textcircled{1} \overset{\overset{\mathtt{c}}{\frown}}{\underset{\underset{\mathtt{a}}{\smile}}{}} \textcircled{2}$$

Then, the transformation result with ε-edges and structured IDs is as follows.



The graph is bisimilar to the following graph.

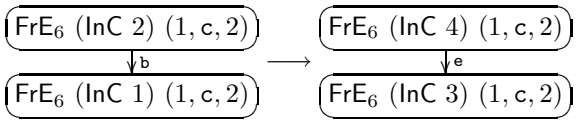


According to the backward semantics of the transformation *a2b*, if we modify graph G to G' by changing the edge label `c` to `d`, then $\langle\!\langle a2b \rangle\!\rangle^{\{\$db \mapsto s\}}_{G'}$ returns binding $\{\$db \mapsto s'\}$ where

$$s' = \textcircled{1} \overset{\overset{\mathtt{d}}{\frown}}{\underset{\underset{\mathtt{a}}{\smile}}{}} \textcircled{2}.$$

The edge `c` in the source is changed to `d` successfully.

If the subgraph in the view is changed as



then, according to the bidirectional semantics of **if**, the source is changed to the following graph.

$$\textcircled{1} \overset{\overset{\mathtt{c}}{\frown}}{\underset{\underset{\mathtt{e}}{\smile}}{}} \textcircled{2}$$

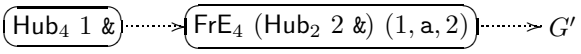
□

Example 9 (Extracting Subgraph)**.** The following transformation *at_ab* extracts all the subgraphs that can be reachable from the root by path `a.b`.

$$
\begin{array}{l}
at\_ab = \\
\quad \mathbf{rec}(\lambda(\$l, \$g). \\
\qquad \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then} \\
\qquad\quad \mathbf{rec}(\lambda(\$l', \$g'). \\
\qquad\qquad \mathbf{if}\ \$l' = \mathtt{b}\ \mathbf{then}\ \$g'\ \mathbf{else}\ \{\}_1)(\$g)_2 \\
\qquad \mathbf{else}\ \{\}_3)(\$db)_4
\end{array}
$$

Then, we can reflect any changes on the extracted graphs to the corresponding source as long as the changed graph has appropriate structured IDs.

Let $\$db$ be graph $\textcircled{1} \xrightarrow{\mathtt{a}} \textcircled{2} \xrightarrow{\mathtt{b}} G$. Then the reachable part of the transformation results with structured ID becomes



where subgraph G' is obtained by replacing node v in G with $\mathsf{FrE}_4\ (\mathsf{FrE}_2\ v\ (2, \mathtt{b}, r))\ (1, \mathtt{a}, 2)$ where r is the root of G. Any modification on G' can be reflected to G in the source. □

Example 10 (Customer2Order)**.** The transformation mentioned in Section 2.3 is translated into UnCAL expression that has nesting of **rec** similar to that in Example 9. Concrete UnCAL expression can be seen in Section C in the appendix. Since extracted subgraph can accept arbitrary updates, if we modify the label `16/10/2008` of the edge from node 7 to node 6 in graph g_{ord} in Figure 6, backward transformation systematically finds the corresponding edge and modify the edge from node 7 to node 6 in g_{cus} in Figure 5, because the modified part is an extracted subgraph reachable from the root by path `customer.order.date`. Subgraphs that are reached by the paths `customer.order.no`, `customer.order.order_of.name` and `customer.order.order_of.add` can be updated similarly. □

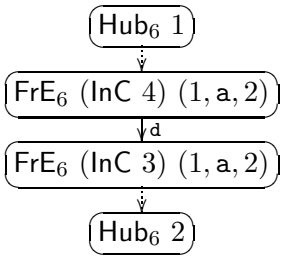
4.5 Insertion Reflection based on ε-edge/ID Structure

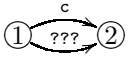
So far the bidirectional semantics can only cope with in-place updating and deletion – modifications on edge labels or updating and deletion on extracted subpart of the original graph. However, except for the extracted subpart, it has the great limitation that no edge can be added to the original graph because it tightly uses the structures of ε-edges/IDs, which also enables us the backward semantics. In this section, we show how to extend it moderately to coped with insertion.

In fact, supporting insertion in bidirectional transformation is challenging because the inserted part has no correspondence in the original source. Foster et al. (2005) and Bohannon et al. (2008) solve this problem by explicitly defining a *create* function to create the "original" data only from

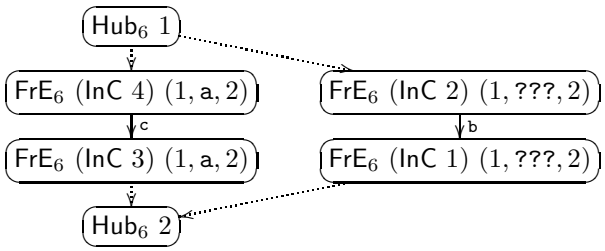
a inserted subpart. Matsuda et al. (2007) allow insertion only if the "original" data is uniquely determined by the inserted subpart. Liu et al. (2007) treat insertion only on the results of "map" but there is no guarantee of bidirectional properties in the framework.

Our insertion handling is inspired by those in Liu et al. (2007) and Matsuda et al. (2007). Like Liu et al. (2007), we treat insertion only on the result of **rec**. And, we follow Matsuda et al. (2007), we use ε-edge/ID structure as "complementary" information to uniquely create source data. Let us explain our basic idea with Example 8. If $\$db$ is a graph $(1) \xrightarrow{\mathtt{d}} (2)$, then the transformation results in the graph on the right. In our insertion handling, we consider what happens if there exists an extra edge between two nodes, i.e., the transformation result of the following graph.

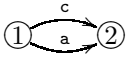




Instead of inserting single edge, we consider the transformation result of the graph with inserted edge $(1, \mathtt{???}, 2)$ that have undetermined label. Concretely, the input of the backward transformation with insertion handling is the following graph.



According to the semantics of **rec**, insertion of the FrE-subpart implies insertion of the edges. The structured ID in the inserted graph contains `???` because at the time we do not know what is the edge that generates the inserted subgraph. Hence, by the backward transformation discussed here, we estimate a concrete label of the inserted edge. Because of the structured ID of the inserted subgraph, we can know that inserted subgraph comes from true-branch of the following **if**-expression **if** $\$l = \mathtt{a}$ **then** $\{\mathtt{b} : \&_1\}_2$ **else** $\{\$l : \&_3\}_4$ Since condition $\$l = \mathtt{a}$ must be true to obtain the inserted subgraph, the backward transformation replaces the label `???` of $\$l$ by `a`. Hence, we obtain the following graph by the insertion.



In the rest of section, we first discuss the modification of the backward semantics for insertion reflection. The backward semantics of **rec** and **if** will be changed: the new semantics of **rec** handles inserted subpart, and the

new semantics of **if** estimates a concrete label to replace `???`. After that, we discuss expressive power of our insertion reflection.

Note that `???` may cause the failure of the forward transformation; if it appears in the condition of **if**-expression, we cannot determine which branch to use. Recall that some backward semantics of an expression uses the forward semantics of its subexpression. Note that the failure of forward transformation do not mean the failure of the backward transformation immediately. When they are used for graph decomposition in a backward transformation, the backward transformation fails.

4.5.1 Backward Semantics of rec for Insertion Reflection

The modification on the **rec** for insertion reflection is very small. Since the IDs of an inserted subgraph must have the form $\mathsf{FrE}\_\ \_\ (\_, \texttt{???}, \_)$, we can easily split the inserted subpart from a modified result of **rec**. The definition of $\mathsf{decomp}_{\mathbf{rec}}$ in Figure 11 changes a bit as follows.

$$
\begin{array}{l}
\mathsf{decomp}_{\mathbf{rec}}(\ldots (V', E', I', O') \ldots) = \\
\quad \{\ldots \mid e \in E_a \ldots \rho_e = \rho\{\ldots \$g \mapsto \mathit{subgraph}(G_a, e)\} \\
\qquad \text{where } \ldots
\end{array}
$$

$$\downarrow$$

$$
\begin{array}{l}
\mathsf{decomp}_{\mathbf{rec}}(\ldots (V', E', I', O') \ldots) = \\
\quad \{\ldots \mid e \in E'_a \ldots \rho_e = \rho\{\ldots \$g \mapsto \mathit{subgraph}(G'_a, e)\} \\
\qquad \text{where } E'_a = E_a \\
\qquad\qquad\qquad \cup \{(u, \texttt{???}, v) \mid \mathsf{FrE}_p\ \_\ (u, \texttt{???}, v) \in E'\} \\
\qquad\qquad G'_a = (V_a, E'_a, I_a, O_a) \\
\qquad\qquad \ldots
\end{array}
$$

The *subgraph* is performed on G'_a, which contains edge $(u, \texttt{???}, v)$, instead of G_a because to guarantee PutGet we must ensure that the inserted edge does not affect other part of the output graph (V', E', I', O').

4.5.2 Estimating Label of Edge to be Inserted

There are only two places that backward transformation can replace `???` with a concrete value.

- Use of label variable $\$l$ caused by **rec**.
- Condition such as $\$l = \texttt{a}$ in **if**.

Since the semantics of the former is already described in the previous section, we discuss the latter here.

Here, we only write the formal definition of the backward semantics of **if** where the evaluation result of condition is undetermined because of variable $\$l$ with `???` in the expression.

$$
\begin{array}{l}
\langle\!\langle \textbf{if } b \textbf{ then } t_1 \textbf{ else } t_2 \rangle\!\rangle^{\rho}_{G} = \\
\quad \begin{cases}
\textsc{Fail} & \text{if } [\![b]\!]^{\rho''_i} = \texttt{???} \text{ for } i = 1, 2 \\
\rho''_1 & \text{if } [\![b]\!]^{\rho''_1} = \mathsf{true} \\
\rho''_2 & \text{if } [\![b]\!]^{\rho''_2} = \mathsf{false}
\end{cases} \\
\quad \text{where } \rho'_1 = \langle\!\langle t_1 \rangle\!\rangle^{\rho}_{G} \\
\qquad\quad \rho'_2 = \langle\!\langle t_2 \rangle\!\rangle^{\rho}_{G} \\
\qquad\quad \rho''_1 = \mathsf{refine}(\rho'_1, b) \\
\qquad\quad \rho''_2 = \mathsf{refine}(\rho'_2, \mathsf{not}\, b)
\end{array}
$$

Above, $\mathsf{refine}(\rho, b)$ is a function to be used to refine the environment ρ by replacing `???` with a concrete value so as to make the evaluation result b to be `true`.

There are many choices of $\mathsf{refine}(\rho, b)$. A strong approach would be to use some constraint solver. Instead, we take a more lightweight choice that cares a condition of the form $\$l = v$ or $\$l \neq v$ and fixes the value of $\$l$ even though there are multiple possibilities. For example, $\mathsf{refine}(\{\$l \mapsto \texttt{???}\}, \$l = \texttt{a})$ returns $\{\$l = \texttt{a}\}$ while $\mathsf{refine}(\{\$l \mapsto \texttt{???}\}, \$l \neq \texttt{a})$ returns $\{\$l \mapsto \texttt{bogus}\}$ where `bogus` is magically chosen value to suffice $\$l \neq \texttt{a}$. Note that $\mathsf{refine}(\{\$l \mapsto \texttt{???}, \$r = \texttt{a}\}, \$l = \$r)$ returns $\{\$l \mapsto \texttt{a}, \$r = \texttt{a}\}$ because we can know the concrete value of $\$r$. To guarantee PutGet, refine fails if it encounters expression such as $\$l = \$r$ where both $\$l$ and $\$r$ are bound to `???`.

4.5.3 Discussion of Insertion Reflection

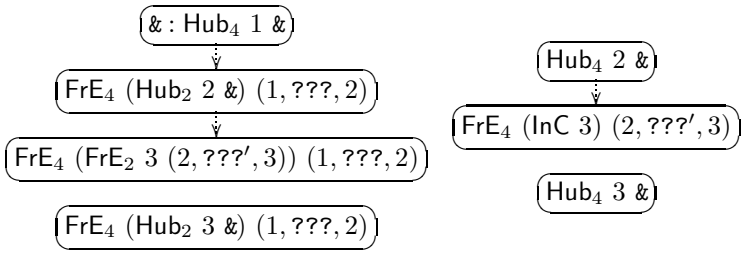
As we have shown with Example 8, the insertion reflection method here work well for single **rec**. However, the insertion reflection method does not work for the nested **rec** such as that in Example 9.

The main reason of the limitation is that the one graph can be traversed more than once by nested **rec**; since the backward transformation reflect the concrete graph to the source by each traverse, the reflection would conflict for each traverse.

Let us consider the transformation of Example 9. Let $\$db$ be a graph with isolated three nodes ①, ② and ③. Then, the transformation result consists of three hub nodes of the form Hub_4 `_ &` but does not have any edge. If we want to insert something to the view, we must insert `a.b` path to the source.

To insert `a.b`, we consider the transformation result of $\$db = ①\xrightarrow{\texttt{???}}②\xrightarrow{\texttt{???}}③$ and try to replace the first `???` with `a` and the second with `b` by the backward transformation. Hence we must consider the backward transformation of the

following graph.



Here, for presentation, we denote `???` to be replaced by `a` and `b` by `???` and `???`′ respectively.

However, the insertion reflection fails due to the following two problems.

- The backward transformation of the outer **rec** calls the backward transformation of **rec**-body for $\mathsf{FrE}_4\ \_\ (1, \texttt{???}, 2)$ nodes. However, the backward transformation of **rec**-body fails because of mismatch of hub nodes of the inner **rec**. The V_{hub} of the inner **rec** in the original data is $\mathsf{Hub}_2\ 2\ \&$ (hub nodes of reachable part of node 2), while the hub nodes of the output graph are $\mathsf{Hub}_2\ 2\ \&$ and $\mathsf{Hub}_2\ 3\ \&$.

- The backward transformation of the outer **rec** tries to estimate the label of inserted edge. If the first problem is solved well, we can estimate the label of the edge $(1, \texttt{???}, 2)$ from the condition $\$l = \texttt{a}$. However, the estimation of the second edge $(2, \texttt{???}, 3)$ by the outer **rec** can replace `???` with label different from `b`, which causes the failure by inconsistent environement in the later step.

There would be many approaches towards the problems. Preprocessing UnCAL expression using fusion method to flatten nested **rec** to one flat **rec** avoids the problems. However, if the transformation has "join", in which inner **rec** uses bound variable of outer **rec**, it is hard to flatten nested **rec**s.

Even when the fusion method is not applicable, the first problem would be relatively easy to solve. If the view contains an extra hub node $\mathsf{Hub}_p\ v\ \_$, we can safely assume that the node comes from the new or existing node v. Since the original graph is available in backward transformation, we can know whether a node is new or existing. On the contrary, more discussion is needed to solve the second problem. The second problem seems to be solved by changing backward semantics to calculate bindings that maps variable to a *set* of values and with constraints on the variables. However, in addition to the problem of choosing appropriate description of such *sets*, it makes the guarantee of PutGet property much and much harder.

- We must give a representation of "set" of variables and "constraints". Since we have variable-to-variable comparison as $\$l = \$r$, the inter-variable constraint must be cared.

- Environment merging operator $\uplus$ must consider the "sets" and "constraints".

Note that approximation by wider (less constrained) set may not work. Since the sets and the constraints represents the set of sources of an output graph, the approximated set might be too wide to guarantee PUTGET.

One may think that it is also a problem that the insertion here does not consider insertion of nodes. However, in contrast to the above two problems, the insertion of node is easily solved because an isolated node in the source is always transformed to an isolated hub node in the view by **rec**. However, because the first problem above, we can use the inserted node only in outermost **rec**.

Since the essence of difficulties of insertion reflection in Customer2Order that appeared in Section 2.3 is contained in insertion reflection in Example 9, we hardly insert subgraphs in Customer2Order because of the same reasons. To give an appropriate and reasonable solution to the problem is one of our future work.

4.6 Main Theorem

Now that semantics of all UnCAL expressions are defined, we summarize our main result as the following theorem. See section A in the appendix for the proof of the theorem.

Theorem 2. *Every bidirectionalized UnCAL expression satisfies* (PUTGET) *and* (GETPUT) *properties, provided that its forward and backward evaluation succeeds.*

5 Implementation and Experiments

The prototype system has been implemented in OCaml and is available online[5]. One can run UnQL queries both forwardly and backwardly. In addition to the examples in Buneman et al. (2000), we have tested the following three non-trivial examples, showing 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 platform specific object model.
- *Class2RDB*; a non-trivial benchmark application for testing power of model transformation languages (Bezivin et al. 2005).

[5] `http://www.biglab.org`

All of them demonstrate the effectiveness of our approach in practical applications.

In our implementation, we carefully treat ε-edges, unreachable parts introduced during operations related to markers, and retrieval of edges or nodes of interest, which greatly affect the performance. Bad treatment would hinder large scale UnQL queries to evaluate in bidirectional mode[6] in a reasonable amount of time. Several orders of magnitude speed-up has been achieved since our initial implementation by the following optimizations.

Reduction of the number of ε-edges As mentioned in the UnQL paper (Buneman et al. 2000), ε-edges are generously generated during evaluation, especially in **rec**. It makes evaluation process slow due to growth of input size. Removing ε-edges during evaluation does 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. Towards prudent removal of ε-edges suitable for backward evaluation, our ε-removal algorithm glue 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 mismatch of markers. This mismatch happens typically in the case of projection of graph components $\&z_i$ by idiom $\&z_i \mathbin{@} g$ and in the case of @ in the definition of **rec** in $\mathbf{rec}(\lambda(\$l, \$g).t_{\mathrm{b}})(l : g) = t_{\mathrm{b}}(l, g) \mathbin{@} \mathbf{rec}(t_{\mathrm{b}})(g)$. Typical usage of **rec** includes t_{b} with no output marker, where actual recursion below the first level from the input nodes — subgraphs that is 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 optimization opportunities; not to evaluate t_{b} for the input graph that is lower than the first level. Performance improvement was significant for the forward evaluation.

Indexing As seen in the formal notion of bulk semantics $\mathsf{compose}_{\mathbf{rec}}$ and $\mathsf{decomp}_{\mathbf{rec}}$, pattern matching on edges for the entire graph is often needed. Constructing maps from node identifiers to the sets of associated incoming and outgoing edges has been effective to improve performance on both forward and backward directions.

6 Related Work

In addition to the related work in the introduction, we discuss other most related work.

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

Bidirectional transformation has been discussed as view updating problem in the database community. Bancilhon and Spyratos (1981) proposed a general approach to the problem. They introduced an elegant solution based on the concept of constant complement view which enables recovery of lost information of the sources. It has been applied to bidirectional transformation on relational database (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001; Lechtenbörger and Vossen 2003) and tree structures (Matsuda et al. 2007). However, this approach assumes the presence of automatic inverse transformation, which makes it difficult to use.

Foster et al. (2005) and Bohannon et al. (2008) proposed the first linguistic approach to bidirectional transformation. They developed some domain specific languages for supporting development of bidirectional transformation on strings and trees. However, their approach is limited to strings and trees, and difficult to be applied to graph transformation due to graph-specific features such as circularity and sharing.

In the context of software engineering, there are several work on bidirectional model (graph) transformation (Ehrig et al. 2005; OMG 2005; Jouault and Kurtev 2006), which can deal with a kind of graph structures. As pointed by Stevens (2007), there lacks clear formal semantics of bidirectional model transformation and there is no powerful bidirectionalization method yet that can be used to automatically derive backward model transformations from forward model transformations so that both transformations form a consistent bidirectional model transformation. This calls for a more serious study on bidirectional graph transformation, which actually inspired the work in this paper.

The concept of structural recursion is not new and has been studied in both the database community (Breazu-Tannen et al. 1991) and the functional programming community (Sheard and Fegaras 1993). However, most of them 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). It is UnCAL (Buneman et al. 2000) that shows the idea of structural recursion can be extended to graph, but the focus there was on query fusion optimization rather than bidirectionalization.

7 Conclusion

In this paper, we report our first attempt towards 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 be recognized so 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. Current prototype system and its application to bidirectional (UML) model transformation and model synchronization show that our framework is promising for practical use.

Future work includes extension of the framework for more flexible insertion, introduction of graph schemas to provide structure information for more efficient bidirectional computation, and more practical application of the system for bidirectional model transformation in software engineering.

Acknowledgements

The research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from National Institute of Informatics, the National Natural Science Foundation of China under Grant No. 60528006, 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

François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

J. Bezivin, B. Rumpe, and Tratt L Schürr A. Model transformation in practice workshop announcement. In *MTiP 2005, International Workshop on Model Transformations in Practice*. Springer-Verlag, 2005.

Richard Bird and Oege de Moor. *Algebras of Programming*. Prentice Hall, 1996.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In George C. Necula and Philip Wadler, editors, *POPL '08: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 407–419. ACM, 2008.

Val Breazu-Tannen, Peter Buneman, and Shamim Naqvi. Structural recursion as a query language. In *Proc. of the Third International Workshop on Database Programming Languages(DBPL 91)*, pages 9–19, 1991.

Peter Buneman, Mary F. Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, 2000.

Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Model Transformation (ICMT 2009)*, pages 260–283. LNCS 5563, Springer, 2009.

Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice*. Springer-Verlag, 2005.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 2005.

Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

Stephen J. Hegner. Foundations of canonical update support for closed database views. In *ICDT '90: Proceedings of the Third International Conference on Database Theory*, pages 422–436, London, UK, 1990. Springer-Verlag.

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

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

Frederic Jouault and Ivan Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, pages 128–138. LNCS 3814, Springer, 2006.

Ralf Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations*, November 2004.

Dominique Laurent, Jens Lechtenbörger, Nicolas Spyratos, and Gottfried Vossen. Monotonic complements for independent data warehouses. *The VLDB Journal*, 10(4):295–315, 2001.

Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Transactions on Database Systems*, 28(2):175–208, 2003.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. Bidirectional interpretation of xquery. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pages 21–30, New York, NY, USA, 2007. ACM Press.

Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 47–58. ACM Press, October 2007.

Lambert 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.

Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE*, pages 251–260. IEEE Computer Society, 1995.

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

T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proc. 10th MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

A Proof of Main Theorem

Before proving the main theorem, we show the following lemma.

Lemma 3. *Composition and decomposition* (decomp) *for every UnCAL constructor* $\odot$ ($\odot \in \{(), \{\}, \&y, \{\_:\_\}, \cup, \oplus, @, :=\}$) *satisfy the following* (DPutGet) *and* (DGetPut) *properties.*

$$\frac{(\odot)\ v_1 \ldots v_n = G}{\mathsf{decomp}_{(\odot)\ v_1 \ldots v_n}(G) = (v_1, \ldots, v_n)} \qquad \text{(DGetPut)}$$

$$\frac{\mathsf{decomp}_{(\odot)\ v_1 \ldots v_n}(G') = (v'_1 \ldots, v'_n)}{(\odot)\ v_1 \ldots v_n = G'} \qquad \text{(DPUTGET)}$$

This lemma trivially holds by the definition of bidirectional semantics for each constructor.

Now return to the proof of the main theorem. It is sufficient to show that every bidirectional UnCAL operator satisfies (PUTGET) and (GETPUT) properties, provided that both $[\![\cdot]\!]$ (get) and $\langle\!\langle \cdot \rangle\!\rangle$ (put) do not fail.

For forward direction, every operator behaves identically to the original UnCAL semantics, so they are defined according to the original semantics. For example, $[\![e_1 \cup e_2]\!]$ is defined only if input markers of result graphs of e_1 and e_2 agree, as in the definition in Section 4.4. So we focus on the backward direction. For the base case, we disallow any modifications on nullary constructors, so $\langle\!\langle \{\} \rangle\!\rangle$, $\langle\!\langle () \rangle\!\rangle$ and $\langle\!\langle \&y \rangle\!\rangle$ is defined only if the target remains unchanged. In this case, both (GETPUT) and (PUTGET) trivially hold. Next we examine the inductive case. For $\langle\!\langle \{t_1{:}t_2\} \rangle\!\rangle^{\rho}$, we disallow modifications of the number of edges from the root. Under this constraint, it succeeds if put on both components t_1 and t_2 succeed. Let $[\![t_1]\!]^{\rho} = l_1$ and $[\![t_2]\!]^{\rho} = G_1$, respectively so $G = \{l_1{:}G_1\}$ is produced as a final result. To see if (GETPUT) holds, suppose (GETPUT) on both operands hold as an induction hypothesis. $\mathsf{decomp}_{\{l_1:G_1\}}(G)$ in $\langle\!\langle \{t_1{:}t_2\} \rangle\!\rangle$ decompose G by traversing one edge $(\mathsf{root}(G), l', v)$ from the root of G and obtain l_1 as the label of the edge, and remaining part as $\mathsf{subgraph}(G, v)$. If there are more than one of such edges, the put is undefined (and the violation of the above condition is signaled). But if the target is unmodified, we obtain l_1 and G_1 again by (DGETPUT). Because of the induction hypothesis, we obtain unmodified environment ρ for both operands. The result environment is unchanged because $\rho = \rho \uplus_{\rho} \rho$. Other simple operators except **rec** can be treated similarly.

To see if (PUTGET) holds, suppose (PUTGET) on both operands hold. We have to show that $\langle\!\langle \{t_1{:}t_2\} \rangle\!\rangle^{\rho}_{G'} = \rho'$ implies $[\![\{t_1{:}t_2\}]\!]^{\rho'} = G'$. According to the definition of $\langle\!\langle \{t_1{:}t_2\} \rangle\!\rangle_{G'}$, $\mathsf{decomp}_{\{l_1:G_1\}}\, G' = (l', G'')$, $\langle\!\langle t_1 \rangle\!\rangle^{\rho}_{l'} = \rho'_1$ and $\langle\!\langle t_2 \rangle\!\rangle^{\rho}_{G''} = \rho'_2$ should have been computed. Since we can assume $[\![t_1]\!]^{\rho'_1} = l'$ and $[\![t_2]\!]^{\rho'_2} = G''$ from the above induction hypothesis, and $\rho'_1 \uplus_{\rho} \rho'_2$ has been equal to ρ', and by (DPUTGET), i.e., $\{l'{:}G''\} = G'$, $[\![\{t_1{:}t_2\}]\!]^{\rho'} = G'$ holds. Other simple operators except **rec** can be treated similarly.

For (GETPUT) on $\mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}})$, since $\mathsf{compose}_{\mathbf{rec}}$ and $\mathsf{decomp}_{\mathbf{rec}}$ satisfy symmetry similar to (DGETPUT) and (DPUTGET), no modification on the view obtain identical set of bindings of $\$l$ and $\$g$. So $\uplus_{\rho}$ in merge for $\langle\!\langle \mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}}) \rangle\!\rangle^{\rho}_{G}$ for $G = [\![\mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}})]\!]^{\rho}$ results in no conflict. For (PUTGET) on **rec**, G'_a in the definition of $\langle\!\langle \mathbf{rec}(\lambda(\$l, \$g).\ t_{\mathrm{b}})(t_{\mathrm{a}}) \rangle\!\rangle^{\rho}_{G'}$ is obtained again in another get assuming (PUTGET) property on t_{a}. So assuming (DPUTGET) like property of

$\mathsf{decomp}_{\mathbf{rec}}$ and $\mathsf{compose}_{\mathbf{rec}}$, input of $\mathsf{compose}$ in another get will be identical to decomposition $\{(e, (V'_e, E'_e, I'_e, O'_e))\}$ of the first put in the definition of $\mathsf{decomp}_{\mathbf{rec}}$, thus another get results in G' again. Same argument hold in the presence of insertion because of the given structured IDs for every insertion unit discussed in Section 4.5.

B Merging operation in backward semantics of rec

Backward evaluation of $\mathbf{rec}(t_b)(t_a)$ needs to combine backward evaluation results of the body (t_b). This has been simply represented using $\bigcup$ in function merge in Figure 11. This operation does not consist of simple compnentwise set union, since simple unification would result in apppearance of original edge as well as the edge after modification at the same time, when the label of an edge is modified.

This section describes this operation. Meanings of each symbols are the same as in Figure 11 unless mentioned otherwise.

The following equation shows, in function merge, how to compute G'_a, the updated graph to be fed to backward evaluation of t_a, where $G_1 \setminus G_2$ denotes componetwize (set) difference of two graphs.

$$
\begin{aligned}
&G'_a = G_a \oplus G_a^{\mathrm{add}} \setminus G_a^{\mathrm{del}} \\
&\quad \text{where} \quad (G_a^{\mathrm{add}}, G_a^{\mathrm{del}}) = \\
&\bigoplus \left\{ (G'_e \setminus G_e, G_e \setminus G'_e) \;\middle|\; \begin{array}{l} e \in E_a, (e, \rho'_e) \in \mathcal{R}, \\ (V'_e, E'_e, I'_e, O'_e) = \rho'_e(\$g), \\ G'_e = (V'_e \cup \{u\}, E'_e \cup \{(u, \rho'_e(\$l), \mathsf{root}(I'_e))\}, I'_e, O'_e), \\ (V_e, E_e, I_e, O_e) = \rho_e(\$g), \\ G_e = (V_e \cup \{u\}, E_e \cup \{(u, \rho_e(\$l), \mathsf{root}(I_e))\}, I_e, O_e) \end{array} \right\}
\end{aligned}
$$

where $\bigoplus$ is defined by $\oplus$ across first and second component of pairs:

$$(g_1, g_2) \bigoplus (g_3, g_4) = (g_1 \oplus g_3, g_2 \oplus g_4).$$

G'_a is computed by accumulating "effects" for graphs G_e consists of edge e in E_a and reachable subgraphs pointed by e. "effects" are computed by set union and set difference between original subgraph G_e and updated subgraph G'_e. □

C Customer2Order in UnCAL

Figure 12 shows UnCAL code that is translated from UnQL code in Section 2.3.

```
rec(\ ($L,$fv3).
  if $L = customer
  then rec(\ ($L,$o).
         if $L = order
         then rec(\ ($L,$c).
                if $L = order_of
                then rec(\ ($L,$date).
                       if $L = date
                       then rec(\ ($L,$no).
                              if $L = no
                              then rec(\ ($L,$a).
                                     if $L = add
                                     then rec(\ ($L,$name).
                                            if $L = name
                                            then rec(\ ($L,$code).
                                                   if $L = code
                                                   then rec(\ ($L,$info).
                                                          if $L = info
                                                          then rec(\ ($L,$fv1).
                                                                 if $L = type
                                                                 then
                                                                 rec(\ ($L,$anyv2).
                                                                   if
                                                                   $L = shipping
                                                                   then
                                                                   {order:
                                                                    {
                                                                    date: $date,
                                                                    no: $no,
                                                                    customer_name: $name,
                                                                    addr: $a}}
                                                                   else
                                                                   {})
                                                                 ($fv1)
                                                                 else
                                                                 {})
                                                          ($a)
                                                   else {})
                                            ($a)
                                     else {})
                              ($a)
                       else {})
                ($c)
                else {})
         ($o)
         else {})
  ($fv3)
  else {})
($db)
```

Figure 12: UnCAL code translated from UnQL code in Section 2.3