ISSN 1884-0760

GRACE TECHNICAL REPORTS

Guaranteeing Free-edits to Bidirectional Graph Transformations

Ezgi Çiçek Soichiro Hidaka

GRACE-TR 2016–04 September 2016

Guaranteeing Free-edits to Bidirectional Graph Transformations [*]

Ezgi Çiçek
MPI-SWS, Germany
ecicek@mpi-sws.org

Soichiro Hidaka[†]
Hosei University, Japan
hidaka@hosei.ac.jp

Abstract

With recent advances in bidirectional graph transformations, changes to input and output graphs can be synchronized and both input and output graphs can be maintained consistently. However, in the previous work, the nature of supported input and output changes has been limited to in-place updates, whereas insertions and deletions have been handled by different mechanisms (such as universal resolving algorithm) that are costly and impractical. Motivated by this gap, we present a free-edit analysis to establish formal bidirectional properties to support free-edits (eg: deletions, insertions, in-place updates etc.) to a graph. With our approach, whenever free-edits are possible to a graph, the change can be directly reflected to the source without doing any backward evaluation. The key to our analysis is the utilization of correspondence traces that specify how a view edge is obtained. We prove our analysis sound with respect to the in-place update semantics and show that valid free-edits constitute a well-behaved bidirectional transformation. We implement the free-edit analysis and integrate it to the existing GRoundTram tool.

1 Introduction

Many applications operate on data that changes bidirectionally: view updates in a database are reflected to the corresponding source of the query [1, 6], changes to UML diagrams are incorporated to the source code or vice versa [18], meta data is kept consistently in two different formats with possible changes to both formats [9]. In all these applications, whenever the output is modified, the input should be updated consistently; and whenever the input is modified, the output should be updated consistently.

One way to ensure the consistency of information between input and output is to design programming languages that are bidirectional in nature. A program written in a bidirectional programming language specifies forward and backward transformations simultaneously and makes sure that consistency is preserved by construction. Bidirectional programs (transformations) are being widely used in different domains including the synchronization of replicated data in

[*]The project was supported by the International Internship Program of the National Institute of Informatics.

[†]Formerly of National Institute of Informatics, Japan

different formats [8], view maintenance in relational databases [2], interactive user interface design [17], coupled software transformation [14] etc.

A complex data structure that is at the heart of many non-trivial bidirectional applications such as UML diagrams, biological information etc. is graphs. In their work, Hidaka *et al.* has proposed a framework to bidirectionalize graph transformations in which structural recursion is handled with bulk semantics [13]. Their work has developed a rigorous bidirectional semantics to ensure the correctness of backward and forward propagations for in-place updates.

Although the previous work has considered a wide range of graph edits, e.g. in-place updates, node/subgraph insertions and deletions, bulk of the bidirectional formalization has focused on providing well-behavedness and correctness guarantees for in-place updates. Unlike the in-place updates, graph edits such as insertions and deletions require non-uniform custom algorithms that might end up being inefficient and difficult to use. For instance, insertions to the view graph are handled in [13] by the universal resolving algorithm (URA) which tries to enumerate all possible subgraphs in the source that could have produced the updated view graph. In practice, this process is often costly since it requires enumerating and searching through all possibly inserted source subgraphs, and it gives little flexibility to the user to direct the enumeration process. Similarly, deletions to the view graph are handled in [13] by a correspondence based algorithm that require re-executing the forward evaluation from source to view to show correctness.

Even though these algorithms for handling insertions and deletions are correct, they are often inefficient. In this work, we aim to design language-based techniques for efficient bidirectional propagation of arbitrary changes such as insertions, deletions or in-place updates. Our analysis on this direction relies on the fact that in practice, many queries just move parts of the data (i.e subgraphs) around without inspecting the underlying input value. We observe that for parts of view graph which is obtained from the input data which is merely copied without being inspected, changes to the view graph can be directly reflected to the source graph without doing any backward propagation, i.e. for the case of insertions, without invoking the costly URA algorithm, hence the name 'free' and for the case of deletions, without requiring a forward evaluation to check the correctness. Similarly, for parts of the data that is not inspected, source changes can be directly reflected to the view graph without doing any forward propagation.

Therefore, we are interested in designing static techniques to identify whether free-edits to a graph (such as insertions, deletions, edge modifications, or arbitrary structural changes) are possible. As a first step in this direction, we design a proof system for establishing formal bidirectional properties to support free-edits to a graph for bidirectional programs written in the UnCAL language. In addition, we also ensure that free-edits preserve the well-behavedness properties.

We first provide an overview of our free-edit analysis, highlighting key design principles and challenges. First, in most cases, free-edit analysis depends on the edit location, i.e. where the user is interested in making a change to. Since some parts of the input data might be inspected and some parts might be merely copied, our analysis has to take into account a particular edit location. Secondly, to reflect the changes back to the source (or the view) without doing any extra costly computation, we rely on the correspondence traces that specify how a view edge was obtained, and where it originates from in the source graph [11]. Initially, correspondence traces were developed for identifying the origin of a view edge and checking editability of a particular view edge without violating the well-behavedness properties. We observe that they are also useful for free-edit analysis.

A correspondence trace is defined as a list of code positions that ended in either an edge (if the view edge is originating from source graph) or a particular code position (if the view edge is originating from a constant label in the code). However, for edges that are originating from multiple code positions, e.g. resulting from union, lists do not suffice. In this work, we also lift traces from lists to sets for accommodating a more precise correspondence analysis and extend them further for structural recursion to also obtain which labels and variables contribute to the result. By utilizing an enhanced form of trace, we are able to identify whether free-edits are possible to a graph at a particular edit location in addition to obtaining a more precise origin and editability analysis. Thirdly, we also ensure that the WPutGet property holds for free-edits, i.e. the changes induced by backward evaluation followed by a forward evaluation also satisfy free-edit conditions.

Our proof system has a free-edit judgment of the form $\Gamma \vdash ep : \mathtt{b}$ where ep specifies the execution path for a specific edge ζ in the view graph and Γ specifies all the labels and variables occurring in the trace of the edit edge ζ. If the free-edit analysis succeeds, then the result of the analysis $\mathtt{b}$ is true (denoted as $\top$), else it is false (denoted as $\bot$).

In summary, we make the following contributions.

1. We develop a proof system for establishing bidirectional properties to support free-edits to a graph and provide a cheap(free) backward propagation mechanism.

2. We show that the well-behavedness properties are preserved for any edit that passes our analysis.

3. We develop an enhanced form of correspondence traces to facilitate static proofs of free-edits.

4. We implement our analysis and integrate it into the existing GRoundTram tool and showcase the usefulness of our approach on several applications.

2 Free-edit Analysis by Example

Example 1 Suppose we have a program as shown in Listing 1 that extracts information for European countries from a source graph (shown in Fig. 1) that contains fact book information about different countries and their demographics. The resulting view graph is shown in Figure 2.

Suppose that a user wants to insert another ethnic group, e.g. Turkish, next to the ethnicity edge $\zeta = (12, German, 11)$ that corresponds to the ethnicity of Germany. Using the insertion reflection algorithm of [13] based on the *universal resolving algorithm*, such an insertion would require searching for all possible computations of the query that could have produced the updated view with the insertion. In practice, this mechanism often becomes unpractical and difficult to

```
select {result: {ethnic: $e, language: $lang, located: $cont}}
where {country: {name:$g, people: {ethnicGroup: $e}, language: $lang, continent:
    $cont}} in $db,
      {$l:$Any} in $cont, $l = Europe
```

Listing 1: Transformation in UnQL

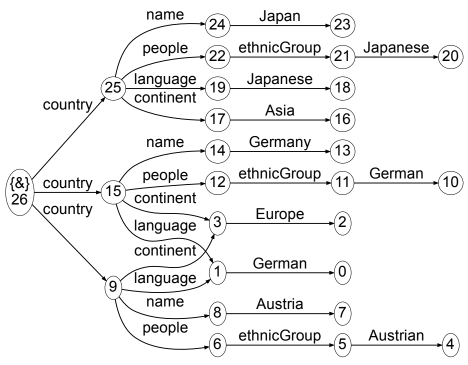

Figure 1: Example Source Graph

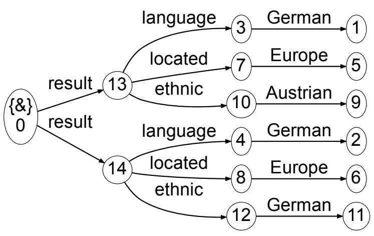

Figure 2: View Graph Generated by Transformation of the Graph in Fig. 1

use because the user doesn't have much control over the search strategy and can not specify the labels of the inserted graph. In addition, if the inserted subgraph is big, searching for a possible evaluation path that created the resulting view could be very inefficient due to an increase in the search space. Instead, using our proposed proof system, the user can determine whether free-edits are possible for a particular edit edge; in this case ζ. Since the graph variable $\$e$ corresponding to the ethnicity information is not inspected, our analysis succeeds and we can directly reflect the insertion change to the source graph. To find where the new ethnicity information must be inserted in the source, we look up the origin edge of the edit edge ζ from its trace: the edge $(12, \mathit{ethnicGroup}, 11)$ is the origin. Hence, we insert the subgraph shown in Figure 3 next to the node $(12, \mathit{ethnicGroup}, 11)$.

3 UnCAL: A core language for Graphs

In this section, we first describe UnCAL (Unstructured CALculus), a powerful graph algebra [3] that is at the core of our development. We briefly explain the syntax and the evaluation semantics for vanilla UnCAL. Later on in Section 4, we extend UnCAL's evaluation semantics with trace information to facilitate backward propagation.

We start with the semi-structured data shown in Figure 2 as a tree with

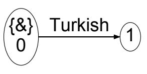

Figure 3: Graph to be Inserted

labelled edges and leaves. It contains the information about the result, e.g. language, location and ethnicity. This graph can be described in UnCAL's syntax as shown below.

```
{result: {language: "Germany", located: "Europe", ethnic: "Austrian"},
 result: {language: "Germany", located: "Europe", ethnic: "German"}}
```

Then, we can define functions (queries) in UnCAL to process the graph. For instance, a query that returns all countries can be defined in UnCAL by

$$\mathbf{rec}(\lambda(\$l, \$g).\,\mathbf{if}\ l = \text{country}\ \mathbf{then}\ \{\text{country} : \$g\}\ \mathbf{else}\ \{\})(\$db)$$

The notation $\{\cdots : \cdots, \cdots\}$ is used to represent graphs in UnCALs term language. The notation $\{t_1, \cdots, t_n\}$ is used as an abbreviation of $\{t_1\} \cup \cdots \cup \{t_n\}$.

UnCAL Graph Model An UnCAL graphs are directed and have (possibly multiple) root(s) and leaves. A graph is a quadruple (V, E, I, O), where V is a set of vertices (nodes), E is a set of edges, I is a one-to-one mapping from a set of input markers to V and O is a many-to-many mapping from V to a set of output markers. A special marker `&` is called the default marker. If $\{\texttt{\&} \mapsto v\} \in I$, v is called an input node and if $v \mapsto \texttt{\&}m \in O$, v is called an output node. Intuitively, input nodes serves entry points whereas the output nodes serve as exit points. The markers are used for references for cycles and sharings. Moreover, they are used as connection points to connect two graphs.

A dotted edge labelled ϵ is referred to as an ϵ-edge, which is a virtual edge connecting two nodes directly. UnCAL's notion of equality for graphs is modulo bisimulation (i.e., not modulo graph isomorphism).

UnCAL Query Language UnCAL's term language consists of constants $\{\}$,$()$, markers $\texttt{\&}y$, definitions $\texttt{\&}x := e$, labelled edges $\{l : g\}$, vertical compositions $g_1 @ g_2$ (append), horizontal compositions $g_1 \oplus g_2$, other horizontal compositions $g_1 \cup g_2$ for merging roots and cycles $\mathbf{cycle}(g)$. The underlying graph theoretic meaning of these constructors are shown in Figure 4.

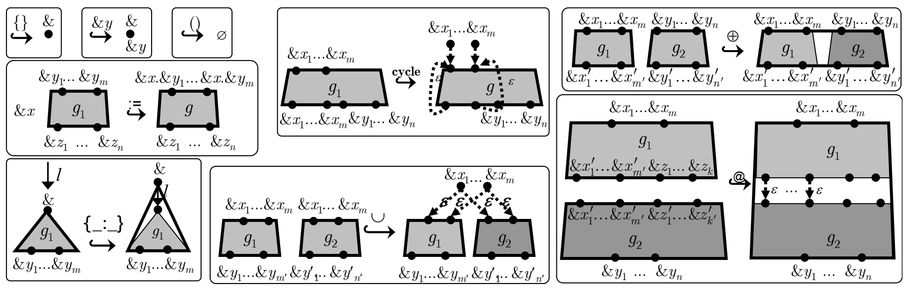

Figure 4: Graph Constructors of UnCAL

There are nine constructors in UnCAL. Three of these constructs are nullary. The constructor () constructs a graph without and edges or nodes. The constructor {} constructs a graph with a single node (marked with the default marker `&`) and no edges. The construct `&`y constructs a graph like $\{\}$, where the node's output marker is `&`y.

The remaining six constructors operate on other graphs. The constructor $\{l : g\}$ constructs a new root with the default input marker `&` with an edge l from the new root to the root of the graph g, i.e. $g.I(\&)$. The union constructor $g_1 \cup g_2$ operates on two graphs with identical sets of input markers and constructs new input nodes for each of the input markers of the graphs, where each new node is connected to the roots of g_1 and g_2 by ϵ edges. The vertical composition construct $g_1 \mathbin{@} g_2$ appends two graphs by connecting the output nodes of g_1 to the input nodes of g_2 by ϵ edges. The rest of the markers of g_1 and g_2 that do not match are ignored. The horizontal composition construct $g_1 \oplus g_2$ unions two graphs disjointly so that the resulting graph inherits all the markers, edges and nodes from the operands. The cycle construction **cycle**(g) connects the output nodes of g to the input nodes of g by ϵ edges to form cycles.

4 Bidirectional Semantics

In this section, we first describe a trace augmented forward and backward semantics for UnCAL (Section 4.1). We then prove our free-edit analysis sound relative to the backward semantics for three different kinds of edits: insertions, deletions, and edge label modifications (Section 6).

4.1 Traced Forward Semantics

We present a new trace-augmented forward semantics that is an extension of the original forward semantics presented in [13] and a variation of [11, 12]. The main extension that is motivated by [12] is that as a result of the forward execution, we also obtain a correspondence trace (similar to [12]) in addition to the resulting view graph. Correspondence traces keep track of code positions and source edges that directly contribute to the creation of each view edge (or respectively each node). In this paper, we make use of the traces for free-edit analysis unlike editability analysis conducted in [12].

The traced forward semantics $\mathcal{F}[\![e]\!]_\rho = (g, t)$ evaluates an expression e under the environment ρ and produces a view graph g and a trace t. The trace maps each view edge $\in$ *Edge* to either a source edge (if it is originating from the source) or code position (if it is originating from the code), preceded by a set or a list of code positions representing the corresponding language constructs that created that particular view edge.

$$\mathit{Trace} \;=\; \mathit{Edge} \rightarrow \mathit{Trace}_{\mathrm{E}}$$
$$\mathit{Trace}_{\mathrm{E}} ::= \mathit{Pos} :: \mathit{Trace}_{\mathrm{E}} \mid [\mathit{Edge}] \mid [\mathit{Pos}] \mid \{\mathit{Trace}_{\mathrm{E}}\} \mid \mathit{Trace}_{\mathrm{E}} \cup \mathit{Trace}_{\mathrm{E}}$$

The environment ρ represents bindings for graph and label variables; it maps each graph variable to the corresponding input graph and each label variable to the corresponding label in the query.

Figure 5 shows the forward evaluation of each expression. We will focus on the trace component since the graph component is the same as in the original forward semantics of [13]. For the constructor $\{\}^p$ that creates a single node, the trace maps the newly created node to the code position p of the constructor. Similarly, the same trace is generated for the output marker constructor `&`y. For the nullary constructor $()^p$ that creates an empty graph, no trace is

$$\overline{\mathcal{F}}\llbracket \{\}^p \rrbracket_\rho = ({}_G\{\}^p, \{G.\mathrm{I}(\&) \mapsto [p]\}) \qquad (\textsc{T-emp})$$

$$\overline{\mathcal{F}}\llbracket \&y^p \rrbracket_\rho = ({}_G\&y^p, \{G.\mathrm{I}(\&) \mapsto [p]\}) \qquad (\textsc{Omrk})$$

$$\overline{\mathcal{F}}\llbracket ()^p \rrbracket_\rho = (()^p, \emptyset) \qquad (\textsc{G-emp})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket e_1 \cup^p e_2 \rrbracket_\rho &= ({}_G(g_1 \cup^p g_2), (t_1 \cup t_2 \cup \{v \mapsto [p] \mid (\&x \mapsto v) \in G.\mathrm{I}\}))\\ &\quad \textbf{where} \quad ((g_1, t_1), (g_2, t_2)) = (\overline{\mathcal{F}}\llbracket e_1 \rrbracket_\rho, \overline{\mathcal{F}}\llbracket e_2 \rrbracket_\rho)\end{aligned} \qquad (\textsc{Uni})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket e_1 \oplus^p e_2 \rrbracket_\rho &= (g_1 \oplus^p g_2, t_1 \cup t_2)\\ &\quad \textbf{where} \quad ((g_1, t_1), (g_2, t_2)) = (\overline{\mathcal{F}}\llbracket e_1 \rrbracket_\rho, \overline{\mathcal{F}}\llbracket e_2 \rrbracket_\rho)\end{aligned} \qquad (\textsc{Duni})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket e_1 \mathbin{@^p} e_2 \rrbracket_\rho &= (g_1 \mathbin{@^p} g_2, t_1 \cup t_2)\\ &\quad \textbf{where} \quad ((g_1, t_1), (g_2, t_2)) = (\overline{\mathcal{F}}\llbracket e_1 \rrbracket_\rho, \overline{\mathcal{F}}\llbracket e_2 \rrbracket_\rho)\end{aligned} \qquad (\textsc{Apnd})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket \{e_{\mathrm{L}} : e\}^p \rrbracket_\rho &= ({}_G\{l : g\}^p, \{(G.\mathrm{I}(\&), l, g.\mathrm{I}(\&)) \mapsto \tau, G.\mathrm{I}(\&) \mapsto [p]\} \cup t)\\ &\quad \textbf{where} \quad ((l, \tau), (g, t)) = (\overline{\mathcal{F}}_{\mathrm{L}}\llbracket e_{\mathrm{L}} \rrbracket_\rho, \overline{\mathcal{F}}\llbracket e \rrbracket_\rho)\end{aligned} \qquad (\textsc{Edg})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket (\&x := e)^p \rrbracket_\rho &= (\&x := g, t)\\ &\quad \textbf{where} \quad (g, t) = \overline{\mathcal{F}}\llbracket e \rrbracket_\rho\end{aligned} \qquad (\textsc{Imrk})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket \mathbf{cycle}^p(e) \rrbracket_\rho &= ({}_G\mathbf{cycle}^p(g), t \cup \{v \mapsto [p] \mid (\&x \mapsto v) \in G.\mathrm{I}\})\\ &\quad \textbf{where} \quad (g, t) = \overline{\mathcal{F}}\llbracket e \rrbracket_\rho\end{aligned} \qquad (\textsc{Cyc})$$

$$\overline{\mathcal{F}}_{\mathrm{L}}\llbracket \mathsf{a}^p \rrbracket_\rho = (\mathsf{a}, [p]) \qquad (\textsc{Lcnst})$$

$$\begin{aligned}\overline{\mathcal{F}}_{\mathrm{L}}\llbracket \$l^p \rrbracket_\rho &= (l, [p])\\ &\quad \textbf{where}\, l = \rho(\$l)\end{aligned} \qquad (\textsc{Lvar})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket \mathbf{llet}^p\ \$l = e_{\mathrm{L}}\ \mathbf{in}\ e \rrbracket_\rho &= (g, t \circ_{\$l} (p : \tau))\\ &\quad \textbf{where}\ (l, \tau) = \overline{\mathcal{F}}_{\mathrm{L}}\llbracket e_{\mathrm{L}} \rrbracket_\rho\\ &\quad \textbf{where}\ (g, t) = \overline{\mathcal{F}}\llbracket e \rrbracket_{\rho \cup \{\$l \mapsto l\}}\end{aligned} \qquad (\textsc{Llet})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket \mathbf{let}^p\ \$g = e_1\ \mathbf{in}\ e_2 \rrbracket_\rho &= (g_2, t_2 \circ_{\$g} (\mathsf{prep}_p\, t_1))\\ &\quad \textbf{where}\ (g_1, t_1) = \overline{\mathcal{F}}\llbracket e_1 \rrbracket_\rho\\ &\quad \textbf{where}\ (g_2, t_2) = \overline{\mathcal{F}}\llbracket e_2 \rrbracket_{\rho \cup \{\$g \mapsto g_1\}}\end{aligned} \qquad (\textsc{Let})$$

$$\begin{aligned}\overline{\mathcal{F}}\llbracket \$g^p \rrbracket_\rho &= (g, \mathsf{prep}_p \ulcorner g \urcorner)\\ &\quad \textbf{where}\ g = \rho(\$g)\end{aligned} \qquad (\textsc{Var})$$

$$\begin{aligned}\overline{\mathcal{F}}\left\llbracket \begin{matrix}\mathbf{if}^p(e_{\mathrm{L}} = e'_{\mathrm{L}})\ \mathbf{then}\ e_{\mathrm{true}}\\ \mathbf{else}\ e_{\mathrm{false}}\end{matrix} \right\rrbracket_\rho &= (g, \mathsf{prep}_p\, t)\\ &\quad \textbf{where}\ ((l, \_), (l', \_)) = (\overline{\mathcal{F}}_{\mathrm{L}}\llbracket e_{\mathrm{L}} \rrbracket, \overline{\mathcal{F}}_{\mathrm{L}}\llbracket e'_{\mathrm{L}} \rrbracket)\\ &\qquad\qquad b = (l = l')\\ &\qquad\qquad (g, t) = \overline{\mathcal{F}}\llbracket e_b \rrbracket_\rho\end{aligned} \qquad (\textsc{If})$$

Figure 5: Forward Semantics

$$\overline{\mathcal{F}}[\![\mathbf{rec}^{p}_{\mathcal{Z}}(\lambda(\$l,\$g).e_{\mathrm{b}})(e_{\mathrm{a}})]\!]_{\rho} = (g', \bigcup_{\zeta\in g.\mathrm{E}} t_{\zeta} \cup t'_{\mathrm{V}}) \tag{Rec}$$

$$\begin{aligned}
\textbf{where}\,(g,t) &= \overline{\mathcal{F}}[\![e_{\mathrm{a}}]\!]_{\rho}\\
M &= \{\zeta \mapsto (g_m, t''_m) \mid \zeta \in g.\mathrm{E}, \zeta \neq \varepsilon, (u,l,v) = \zeta,\\
&\quad (g_m, t_m) = \overline{\mathcal{F}}[\![e_{\mathrm{b}}]\!]_{\rho\cup\{\$l\mapsto l,\$g\mapsto g_v\}},\\
&\quad t'_m = (t_m \circ_{\$l} (p : t(\zeta)) \circ_{\$g} (\mathsf{prep}_p\, t_v)),\\
&\quad \forall \zeta_m \in g_m.\ \ \text{if } \mathtt{origin}(\zeta_m, t_m) \in g_a.\mathrm{Top}\\
&\quad \text{then } t''_m = \mathsf{prep}_{\$l} t'_m\\
&\quad \text{else } t''_m = t'_m\}\\
g' &= (V_{\mathsf{RecN}} \cup \ldots, \_, \_, \_) = \mathsf{compose}^{p}_{\mathbf{rec}}(M, g, \mathcal{Z})\\
t'_{\mathrm{V}} &= \{v \mapsto [p] \mid v \in V_{\mathsf{RecN}}\}\\
t_{\zeta} &= \mathsf{rece}_{p,\zeta}(\mathsf{prep}_p\, \pi_2(M(\zeta)))
\end{aligned}$$

generated since there is no edge or node created. For the binary constructors $\cup$, $\oplus$ and @, the traces of both subexpressions are merged together with the trace created by themselves. The merging of traces $t_1 \cup t_2$ is computed as follows:

$$\begin{aligned}
&\{\zeta \mapsto (\mathtt{if}\ (\zeta \mapsto \tau_2) \in t_2\ \mathtt{then}\ \tau_1 \cup \tau_2\ \mathtt{else}\ \tau_1) &&\mid (\zeta \mapsto \tau_1) \in t_1\} \cup\\
&\{\zeta \mapsto \tau_2 &&\mid (\zeta \mapsto \tau_2) \in t_2 \wedge (\zeta \mapsto \tau_1) \notin t_1\}
\end{aligned}$$

If the view edge ζ is originating from only one of the operands, then we return the corresponding trace edge set. It might also be the case that the view edge is originating from both of the operands (i.e. $\zeta \in dom(t_1) \wedge \zeta \in dom(t_2)$). For instance the query $\mathbf{rec}((\lambda\$l,\$g).\$g^{p_1} \cup \$g^{p_2})$ will create a subgraph which is shared and any edge in this subgraph will be originating from both p_1 and p_2 code locations with an identical origin edge. In such cases, we simply union their corresponding trace edge sets τ_1 and τ_2.

Next, we define what we mean by the origin of a view edge. The edge origin `originE` is defined by

$$\begin{aligned}
\mathtt{originE}(p :: \tau) &= \mathtt{originE}(\tau)\\
\mathtt{originE}([x]) &= x \text{ where } x \in (\mathrm{Edge} \mid \mathrm{Pos})\\
\mathtt{originE}(\{\tau_1, \tau_2, \cdots, \tau_N\}) &= \mathtt{originE}(\tau_1) \text{ where } \forall i.\ \mathtt{originE}(\tau_1) = \mathtt{originE}(\tau_i).
\end{aligned}$$

Note that $\tau_1 \cup \tau_2$ is defined only if the view edge has the same origin edge, that is $\mathtt{originE}(\tau_1) = \mathtt{originE}(\tau_2)$.

For the remaining binary graph constructors $\oplus$ and @, the traces of both subexpressions are combined together with the trace created by themselves.

For variables, we return the identity trace that maps each edge (resp. each node) in the graph to itself.

Label evaluation is denoted as $\overline{\mathcal{F}}_{\mathrm{L}}[\![e_l]\!]_{\rho}$, taking as input a label expressions e_L and an environment ρ, and returning a label and a corresponding trace. If the label expression is a constant, then the trace is simply the code position of the label (rule Lconst). If the label expression is a label variable, then the trace is the code position of the label variable followed by the corresponding trace of the variable from the environment.

For the let bindings of the form $\mathbf{llet}^{p}\ \$l = e_{\mathrm{L}}\ \mathbf{in}\ e$, we first evaluate the label expressions e_L (rule LLET). Then we evaluate the body e in the extended environment mapping l to the result of the label evaluation. The trace for the whole let expression does not only contain the trace of the body, but also the trace of the bind expression as well as the code location of the let expression. This composition is represented by a substitution operation, denoted as $t_1 \circ_{\$l} t_2$, that substitutes the trace of t_1 for all the edges originating from the label variable l in the trace of the body expression. The aim of this substitution is to create a final trace for the whole let expression that encapsulates all the information contained in e_L and e. Let bindings for the graph variables follow a similar pattern.

Our forward semantics is a variation of the trace-augmented forward semantics developed in [11], with two important differences: 1) traces are not merely lists, but are also sets and 2) the environment for the forward evaluation doesn't store the trace information for each binding. Therefore, the initial environment ρ_0 just maps the distinct graph variable $\$db$ to the initial graph g_s. The advantage of having a more complex type for traces is that it allows us to have more precise information for union operations. For instance, consider the following program $\textbf{let}\,\$g = \{a : \$db^{p_1}\}$ **in** $\{b : \$g^{p_2}\} \cup \{c : \$db^{p_3}\}$ and the view edge ζ that originates from the input graph bound to $\$db$. According to the semantics proposed in [11], the traces are lists, and for union operation, union of the traces is not well-defined if the view edge originates from both components; it only stores the code locations of one of the operands, not both. To gain back the precision for such cases, we allow traces to be also sets. Then, for view edge ζ, the trace will store both p_2 and p_3 as well as p_1.

Coincidence Our forward semantics coincides with the forward semantics developed in [11] as long as the environments are identity-preserving, i.e. the initial environment given to the forward semantics of [11] simply maps the graph variable $\$db$ to the input graph without recording any additional trace information.

5 Free-edit analysis

Initially, we assume that after forward-execution of the expression e under an input environment ρ, we obtain a resulting graph G and a corresponding trace t. Given this trace, we assume that user selects an edge ζ in the resulting graph G. Free-edit analysis consists of three steps.

In the first step, given an edit edge ζ in the resulting graph G, we extract the execution path of the expression e that resulted in the selected edit edge. This extraction mechanism, denoted as $\texttt{extract}(t, e, \zeta)$, is defined by a recursive procedure, shown in Figure 7. The aim of the extraction phase is to eliminate parts of the expression that are irrelevant for the free-edit analysis. The syntax of execution paths is a subset of the syntax of expressions: the only difference is that execution paths do not contain conditional expressions. For instance, consider the expression $\textbf{if}^p(e_{\text{L}} = e'_{\text{L}})$ **then** e_{true} **else** e_{false}. If the resulting edge is originating from the **then** branch, the extracted execution path would only contain the execution path corresponding to e_{true} whereas e_{false} is simply omitted.

In the second step, we obtain all the free label variables and constants that occur in the trace of the selected edge, i.e. $t(\zeta)$, and we collect them in Γ,

$\boxed{\Gamma \vdash e : \mathtt{b}}$ execution path e

$$\frac{\$g \in \Gamma}{\Gamma \vdash \$g : \top}\ \text{GVAR} \qquad \frac{}{\Gamma \vdash \$l : \bot}\ \text{LVAR} \qquad \frac{}{\Gamma \vdash \{\}^p, \&y^p, ()^p : \bot}\ \text{CONST}$$

$$\frac{\mathtt{FLV}(l_1, l_2) \cap \Gamma = \emptyset \qquad \Gamma \vdash e : \mathtt{b}}{\Gamma \vdash e\ \mathtt{with}\ (l_1 = l_2) : \mathtt{b}}\ \mathit{IF} \qquad \frac{e_\mathrm{l} \notin \Gamma \qquad \Gamma \vdash e : \mathtt{b}}{\Gamma \vdash \{e_\mathrm{l} : e\}^p : \mathtt{b}}\ \text{EDG}$$

$$\frac{\Gamma \vdash e : \mathtt{b}}{\Gamma \vdash (\&x := e)^p : \mathtt{b}}\ \text{IMRK} \qquad \frac{\Gamma \vdash e_g : \top \qquad \Gamma \vdash e : \mathtt{b} \qquad \$g \in \Gamma}{\Gamma \vdash \mathbf{let}^p\ \$g = e_\mathrm{g}\ \mathbf{in}\ e : \mathtt{b}}\ \text{LET1}$$

$$\frac{\Gamma \vdash e_g : \bot \qquad \Gamma \vdash e : \mathtt{b} \qquad \$g \notin \Gamma}{\Gamma \vdash \mathbf{let}^p\ \$g = e_\mathrm{g}\ \mathbf{in}\ e : \mathtt{b}}\ \text{LET2} \qquad \frac{\Gamma \vdash e : \mathtt{b} \qquad \$l \notin \Gamma}{\Gamma \vdash \mathbf{llet}^p\ \$l = e_\mathrm{L}\ \mathbf{in}\ e : \mathtt{b}}\ \text{LLET}$$

$$\frac{\Gamma \vdash e_\mathrm{a} : \top \qquad \Gamma \vdash e_\mathrm{b} : \top \qquad \$l \notin \Gamma \qquad \$g \in \Gamma}{\Gamma \vdash \underset{\mathcal{Z}}{\overset{p}{\mathbf{rec}}}(\lambda(\$l, \$g).e_\mathrm{b})(e_\mathrm{a}) : \top}\ \text{REC1}$$

$$\frac{\Gamma \vdash e_\mathrm{a} : \bot \qquad \Gamma \vdash e_\mathrm{b} : \mathtt{b} \qquad \{\$g, \$l\} \notin \Gamma}{\Gamma \vdash \underset{\mathcal{Z}}{\overset{p}{\mathbf{rec}}}(\lambda(\$l, \$g).e_\mathrm{b})(e_\mathrm{a}) : \mathtt{b}}\ \text{REC2} \qquad \frac{\Gamma \vdash e_1 : b_1 \qquad \Gamma \vdash e_2 : b_2}{\Gamma \vdash e_1 \cup^p e_2 : b_1 \vee b_2}\ \text{UNI}$$

$$\frac{\Gamma \vdash e : \mathtt{b}}{\Gamma \vdash \mathbf{cycle}^p(e) : \mathtt{b}}\ \text{CYC}$$

Figure 6: Analysis rules

referred as the origin set. Intuitively, Γ contains the origin label variables and constants in the expression e that contributed to the creation of the selected edge ζ.

Finally, in the third step, we perform the actual free-edit analysis given the execution path e and the origin set Γ. Our analysis abstracts over the edge, it only requires the origin set (Γ) and the execution path (e). The analysis judgment $\Gamma \vdash e : \mathtt{b}$ specifies whether free-edits for the execution path e and the origin set Γ are possible. If so, the analysis succeeds, i.e. the result is $\top$. If not, the analysis fails, i.e. the result is $\bot$.

Figure 6 represents the analysis rules for determining whether free-edits are possible. The main principle behind the rules is that analysis fails whenever a label variable or a constant is in the origin set, i.e. it might be possibly inspected. We explain each of the rules in detail. The rule GVAR succeeds whenever graph variable $\$g$ is in the origin set Γ, i.e. the edge is originating from part of the graph that is directly copied. The rule LVAR immediately fails for any label variable. Our analysis conservatively rejects free-edits to label variables due to two main reasons: 1) their modification might require backward propagation due to a possibly different execution path and 2) insertions/deletions of subgraphs at the selected edge correspond to inserting/deleting parts of the original expression. The latter is a modification that is not allowed in bidirectional transformations.

CONST rule forbids modifications to any constant expression because such modifications correspond to changing the original expression. The rule IF forbids modifications to guard label variables and succeeds only if the result

$$\texttt{extract}(t, \{\}^p, \zeta) = \{\} \qquad \text{(E-T-EMP)}$$

$$\texttt{extract}(t, \&y^p, \zeta) = \&y \qquad \text{(E-OMRK)}$$

$$\texttt{extract}(t, ()^p, \zeta) = () \qquad \text{(E-G-EMP)}$$

$$\texttt{extract}(t_1 \cup t_2 \cup t', e_1 \cup^p e_2, \zeta) = \texttt{extract}(t_1, e_1, \zeta) \cup \texttt{extract}(t_2, e_2, \zeta) \quad \textbf{where} \quad \zeta \in \mathrm{dom}(t_1) \wedge \zeta \in \mathrm{dom}(t_2) \qquad \text{(E-UNI-1)}$$

$$\texttt{extract}(t_1 \cup t_2 \cup t', e_1 \cup^p e_2, \zeta) = \texttt{extract}(t_1, e_1, \zeta) \quad \textbf{where} \quad \zeta \in \mathrm{dom}(t_1) \wedge \zeta \notin \mathrm{dom}(t_2) \qquad \text{(E-UNI-2)}$$

$$\texttt{extract}(t_1 \cup t_2 \cup t', e_1 \cup^p e_2, \zeta) = \texttt{extract}(t_2, e_2, \zeta) \quad \textbf{where} \quad \zeta \notin \mathrm{dom}(t_1) \wedge \zeta \notin \mathrm{dom}(t_2) \qquad \text{(E-UNI-3)}$$

$$\texttt{extract}(t_1 \cup t_2, e_1 \oplus^p e_2, \zeta) = \texttt{extract}(t_1, e_1, \zeta) \oplus \texttt{extract}(t_2, e_2, \zeta) \qquad \text{(E-DUNI)}$$

$$\texttt{extract}(t_1 \cup t_2, e_1 \mathbin{@^p} e_2, \zeta) = \texttt{extract}(t_1, e_1, \zeta) \mathbin{@} \texttt{extract}(t_2, e_2, \zeta) \qquad \text{(E-APND)}$$

$$\texttt{extract}(t \cup t', \{e_{\mathrm{L}} : e\}^p, \zeta) = \{e_{\mathrm{L}} : \texttt{extract}(t', e, \zeta)\} \qquad \text{(E-EDG)}$$

$$\texttt{extract}(t, (\&x := e)^p, \zeta) = (\&x := \texttt{extract}(t, e, \zeta)) \qquad \text{(E-IMRK)}$$

$$\texttt{extract}(t \cup t', \mathbf{cycle}^p(e), \zeta) = \mathbf{cycle}(\texttt{extract}(t, e, \zeta)) \qquad \text{(E-CYC)}$$

$$\texttt{extract}(t, \mathtt{a}^p, \zeta) = \mathtt{a} \qquad \text{(E-LCNST)}$$

$$\texttt{extract}(t, \$l^p, \zeta) = \$l \qquad \text{(E-LVAR)}$$

$$\texttt{extract}(t \circ_{\$l} (p : \tau), \mathbf{llet}^p\ \$l = e_{\mathrm{L}}\ \mathbf{in}\ e, \zeta) = \mathbf{llet}^p\ \$l = e_{\mathrm{L}}\ \mathbf{in}\ \texttt{extract}(t, e, \zeta) \qquad \text{(E-LLET)}$$

$$\texttt{extract}(t_2 \circ_{\$g} (\mathsf{prep}_p\, t_1), \mathbf{let}^p\ \$g = e_1\ \mathbf{in}\ e_2, \zeta) = \mathbf{let}^p\ \$g = \texttt{extract}(t_1, e_1, \texttt{origin}(t_2, \zeta))\ \mathbf{in}\ \texttt{extract}(t_2, e_2, \zeta) \quad \textbf{where} \quad \$g \text{ occurs in } t_2 \qquad \text{(E-LET1)}$$

$$\texttt{extract}(t_2 \circ_{\$g} (\mathsf{prep}_p\, t_1), \mathbf{let}^p\ \$g = e_1\ \mathbf{in}\ e_2, \zeta) = \texttt{extract}(t_2, e_2, \zeta) \quad \textbf{where} \quad \$g \text{ not occurs in } t_2 \qquad \text{(E-LET2)}$$

$$\texttt{extract}(t, \$g^p, \zeta) = \$g \qquad \text{(E-GVAR)}$$

$$\texttt{extract}(\underset{p,b}{\mathsf{prep}}\, t, \begin{array}{l}\mathbf{if}^p(e_{\mathrm{L}} = e'_{\mathrm{L}})\ \mathbf{then}\ e_{\mathrm{true}} \\ \qquad\qquad\quad \mathbf{else}\ \ e_{\mathrm{false}}\end{array}, \zeta) = \texttt{extract}(t, e_b, \zeta) \qquad \text{(E-IF)}$$

$$\texttt{extract}(\bigcup_{\zeta \in g.\mathrm{E}} t_\zeta \cup t'_{\mathrm{V}}, \overset{p}{\underset{\mathcal{Z}}{\mathbf{rec}}}(\lambda(\$l, \$g).e_{\mathrm{b}})(e_{\mathrm{a}}), \zeta) = \mathbf{rec}(\lambda(\$l, \$g).e_{\mathrm{b}})(e_{\mathrm{a}}) \quad \textbf{where} \quad \$g \text{ occurs in } t_\zeta \qquad \text{(E-Rec)}$$

Figure 7: Execution path extraction

of the body succeeds. The rule EDG succeeds only if the analyzed edge is not originating from the selected edge. The rule IMRK succeeds only if the analysis of the renamed graph succeeds. Rules LET1 and LET2 represent two cases for the analysis of the let construct for graph variables. If the bound variable $\$g$ is in the origin set, then analysis succeeds if analysis of both of the body and the argument expressions succeed (rule LET1). If the bound variable $\$g$ is not in the origin set, then it suffices for only the analysis of the body expression to succeed (rule LET2). The rule LLET represent the analysis of the let construct for label variables. It succeeds only if the bound label is not in the origin set and the body expression's analysis succeeds.

Similar to let construct for graph variables, there are two cases for the **rec** construct. If the graph variable $\$g$, but not the label variable $\$l$, is in the origin set, then analysis succeeds if analyses of both of the body and the argument expressions succeed (rule LET1). If none of the the graph variable $\$g$ and the label variable $\$l$ are in the origin set, then it suffices for only the analysis of the body expression to succeed. The rule UNI succeeds if any one of the analyses of the operands of the union succeed. Finally, the rule CYC succeeds if the analysis of the body of the cycle construct succeed.

6 Metatheory

In this section, we first show the equivalence of our forward semantics to the one developed in [11] modulo the problem mentioned with respect to unions. Then, we show the soundness of our analysis, i.e. whenever our analysis succeeds, it is indeed possible to directly reflect the changes at the selected edge to the origin without actually performing any backward propagation. In addition, we show that any change that passes our analysis also satisfies the WPutGet property.

We denote the forward semantics developed in [11] as $\mathcal{F}[\![\cdot]\!]_{\cdot}$. An identity environment is the one that takes the distinct graph variable $\$db$ to the corresponding input graph.

Lemma 1 (Equivalence of Forward Semantics)
Let ρ be the identity environment and ρ' be the arbitrary environment, then $\mathcal{F}[\![e]\!]_{\rho\cup\rho'} = (g,t) \iff \overline{\mathcal{F}}[\![e]\!]_{\rho\cup\pi_1\rho'} = (g, t \circ_{\$x_1} \cdots \circ_{\$x_n} \pi_2(\rho'(\$x_n)))$ where $\mathtt{dom}(\rho') = \{\$x_1, \cdots, \$x_n\}$.

The coincidence of the two forward semantics follows as a corollary of this equivalence lemma whenever the initial envrironment is the identity environment.

Corollary 2 (Coincidence)
If ρ_0 is the identity environment, then $\mathcal{F}[\![e]\!]_{\rho_0} = \overline{\mathcal{F}}[\![e]\!]_{\pi_1(\rho_0)}$.

Proof. This follows immediately by Lemma 1. □

Theorem 3 (Soundness Equivalence Simplified)
Let $\overline{\mathcal{F}}[\![e]\!]_{\rho_0} = (G,t)$ and $\zeta \in G.Edges$ be the location user wants to edit.
Assume $e' = \mathtt{extract}(e,t,\zeta)$ and $\mathtt{FLV}(t(\zeta)) \subseteq \Gamma$.
If $\Gamma \vdash e' : \top$, then for any update $\mathtt{upd}(G,\zeta) = G'$, we have
$\mathcal{B}[\![e]\!]_{\rho_0} G' = \mathtt{prop}(\rho_0, \mathtt{upd}, \zeta, t)$

Theorem 4 (Main Soundness Theorem)
Let $\overline{\mathcal{F}}[\![e]\!]_{\rho_0} = (G, t)$ and $\zeta \in G.Edges$ be the location user wants to edit.
Let $\overline{\zeta_i}$ be the set of edges that are created by the same applied edge as ζ.
Assume $e' = \texttt{extract}(t, \zeta, e)$ and $e'_i = \texttt{extract}(e, t, \zeta_i)$ and $\texttt{FLV}(t(\zeta)) \subseteq \Gamma$ and $\texttt{FLV}(\overline{t(\zeta_i)}) \subseteq \Gamma_i$.
If $\Gamma \vdash e' : \top$ and $\Gamma_i \vdash e'_i : \top$, then for any update $\texttt{upd}(G, \zeta) = G'$, we have $\mathcal{B}[\![e]\!]_{\rho_0} G' = \texttt{prop}(\rho_0, \texttt{upd}, \zeta, t)$ and the WPutGet will be satisfied.

Origin Let $\overline{\mathcal{F}}[\![e]\!]_{\rho} = (g, t)$. For an edge $\zeta \in g$, its origin is defined as a pair (p, ζ') where $t(\zeta') = [\cdots p\ \zeta']$.

Update Propagation Assume that $\overline{\mathcal{F}}[\![e]\!]_{\rho} = (g, t)$. Then, for an edge $\zeta \in g$, update propagation is defined as $\texttt{prop}(\rho, \texttt{upd}, \zeta, t) = \{\rho[\$x \leftarrow \texttt{upd}(G, \zeta')] \mid (\$x, \zeta') = \texttt{origin}(\zeta, t)\ \wedge G = \rho(\$x)\ \wedge \zeta' \in G.Edges\}$

7 Related Work

The notion of provenance – information about the origin, derivation, ownership, or history of an object [4] – is extensively studied in the database community and many other application fields [5]. Why (lineage), how and where provenance are computed using linguistic approach in a homomorphic manner where the mathematical structure of the trace is preserved during the computation of the queries. Present work also rely on trace information to accommodate not only label renaming but also deletion and insertion, by identifying the region in the view that accepts free edits.

Though database community exploits provenance information in the context of backward transformation like the work by Fegaras [7], perhaps the most closely related work to ours is semantic bidirectionalization [19] that generates correspondences between input and output elements over polymorphic transformations by feeding to the forward transformation the "artificial" data that includes location identifiers for each element instead of the actual elements. Resulting location information appears in the view corresponds to our traces when they are originated from source. Since we deal with graphs, we combine two traces using directed acyclic graph, so the trace is more enriched in our setting. Voigtlaender's approach benefits from semantic approach in which the forward transformation can be described in any general purpose language, whereas we use syntactic information of a domain (graph transformation) specific language UnCAL.

Our transformations are monomorphic as opposed to the Voigtlaender's setting, because we have transformations including conditionals that compares label variables with label constants. Matsuda and Wang [15, 16] tackled this problems by run-time recording of control flow to be able to check and reject the control flow change during the backward transformation.

Recent work on UnCAL includes that of Hamana [10]. The work provides complete equational reasoning which was not achieved by the original authors of UnCAL [3] when the body of structural recursion depends on the graph variable. This innovation may further bridge our domain-specific approach with more general approach in the programming language field, but we do not have clear road map for this yet.

Acknowledgments

The project was supported by the International Internship Program of the National Institute of Informatics. The authors would like to thank the IPL members for their valuable comments during the internship period.

References

[1] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[2] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS 2006*, pages 338–347, 2006.

[3] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[4] James Cheney, Umut A. Acar, and Amal Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[5] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

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

[7] Leonidas Fegaras. Propagating updates through xml views using lineage tracing. In *ICDE 2010*, pages 309–320, 2010.

[8] 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 2005*, pages 233–246, 2005.

[9] John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, 1998.

[10] Makoto Hamana. Iteration algebras for unql graphs and completeness for bisimulation. In *Proceedings Tenth International Workshop on Fixed Points in Computer Science, FICS 2015, Berlin, Germany, September 11-12, 2015.*, pages 75–89, 2015.

[11] Soichiro Hidaka, Martin Billes, and Quang Minh Tran. A trace-based approach to increased comprehensibility and predictability of bidirectional graph transformations. Technical Report GRACE-TR-2015-03, GRACE Center, National Institute of Informatics, September 2014.

[12] Soichiro Hidaka, Martin Billes, and Quang Minh Tran. Trace-based approach to editability and correspondence analysis for bidirectional graph transformations. In *Fourth International Workshop on Bidirectional Transformations (Bx 2015)*, 2015.

[13] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP'10*, pages 205–216. ACM, 2010.

[14] Ralf Lämmel. Coupled Software Transformations (Extended Abstract). In *First International Workshop on Software Evolution Transformations (SET 2004)*, pages 31–35, November 2004.

[15] Kazutaka Matsuda and Meng Wang. Bidirectionalization for free with runtime recording: Or, a light-weight approach to the view-update problem. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 297–308, New York, NY, USA, 2013. ACM.

[16] Kazutaka Matsuda and Meng Wang. "bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 111:79–109, 2015. DOI:10.1016/j.scico.2014.07.008.

[17] Lambert Meertens. Designing constraint maintainers for user interaction. `http://www.kestrel.edu/home/people/meertens/`, June 1998.

[18] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

[19] Janis Voigtländer. Bidirectionalization for free! (pearl). In *POPL '09*, pages 165–176. ACM, 2009.