ISSN 1884-0760

GRACE TECHNICAL REPORTS

Marker-directed Optimization of UnCAL Graph Transformations (revised version)

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

GRACE-TR 2011–06 November 2011

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Marker-directed Optimization of UnCAL Graph Transformations*

Soichiro Hidaka[1], Zhenjiang Hu[1], Kazuhiro Inaba[1], Hiroyuki Kato[1], Kazutaka Matsuda[2], Keisuke Nakano[3], and Isao Sasano[4]

[1] National Institute of Informatics, Japan,
{hidaka, hu, kinaba, kato}@nii.ac.jp
[2] The University of Electro-Communications, Japan,
ksk@cs.uec.ac.jp
[3] Tohoku University, Japan,
kztk@kb.ecei.tohoku.ac.jp
[4] Shibaura Institute of Technology, Japan,
sasano@sic.shibaura-it.ac.jp

Abstract. Buneman et al. proposed a graph algebra called UnCAL (Unstructured CALculus) for compositional graph transformations based on structural recursion, and we have recently applied to model transformations. The compositional nature of the algebra greatly enhances the modularity of transformations. However, intermediate results generated between composed transformations cause overhead. Buneman et al. proposed fusion rules that eliminate the intermediate results, but auxiliary rewriting rules that enable the actual application of the fusion rules are not apparent so far. UnCAL graph model includes the concept of markers, which correspond to recursive function call in the structural recursion. We have found that there are many optimization opportunities at rewriting level based on static analysis, especially focusing on markers. The analysis can safely eliminate redundant function calls. Performance evaluation shows its practical effectiveness for non-trivial examples in model transformations.
Keywords: program transformations, graph transformations, UnCAL

1 Introduction

Graph transformation has been an active research topic [8] and plays an important role in model-driven engineering [5, 10]; models such as UML diagrams are

---

* This is a revised version of the technical report: *Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, Keisuke Nakano and Isao Sasano, "Marker-directed Optimization of UnCAL Graph Transformations", GRACE-TR-2011-02, GRACE Center, National Institute of Informatics, Jun. 2011*, which was a full version of the paper appeared in informal proceedings of The 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011), pp. 168-182.

represented as graphs, and model transformation is essentially graph transformation. We have recently proposed a bidirectional graph transformation framework [6] based on providing bidirectional semantics to an existing graph transformation language UnCAL [4], and applied it to bidirectional model transformation by translating from existing model transformation language to UnCAL [9]. Our success in providing well-behaved bidirectional transformation framework was due to structural recursion in UnCAL, which is a powerful mechanism of visiting and transforming graphs. Transformation based on structural recursion is inherently compositional, thus facilitates modular model transformation programming.

However, compositional programming may lead to many unnecessary intermediate results, which would make a graph transformation program terribly inefficient. As actively studied in programming language community, optimization like fusion transformation [11] is desired to make it practically useful. Despite a lot of work being devoted to fusion transformation of programs manipulating lists and trees, little work has been done on fusion on programs manipulating graphs. Although the original UnCAL has provided some fusion rules and rewriting rules to optimize graph transformations [4], we believe that further work and enhancement on fusion and rewriting are required.

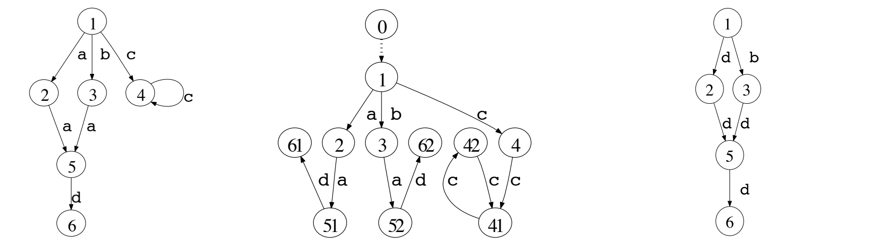
The key idea presented in this paper is to analyze input/output markers, which are sort of labels on specific set of nodes in the UnCAL graph model and are used to compose graphs by connecting nodes with matching input and output markers. By statically analyzing connectivity of UnCAL by our marker analysis, we can simplify existing fusion rule. Consider, for instance, the following existing generic fusion rule of the structural recursion operator in UnCAL:

$$\begin{aligned}&\mathbf{rec}(\lambda(\$l_2,\$t_2).e_2)(\mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)(e_0))\\&\quad=\mathbf{rec}(\lambda(\$l_1,\$t_1).\,\mathbf{rec}(\lambda(\$l_2,\$t_2).e_2)(e_1 \mathbin{@} \mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)(\$t_1)))(e_0)\end{aligned}$$

where $\mathbf{rec}(\lambda(\$l,\$t).e)$ applies transformation e on each edge (whose label is bound to $\$l$ and subgraph pointed by the edge is bound to $\$g$) of the input graph, and combine the results of e to produce the output graph. **rec** encodes a structural recursive function which is an important computation pattern and explained later. The graph constructor @ connects two graphs by matching markers on nodes, and in this case, result of transformation e_1 is combined to another structural recursion $\mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)$. If we know by static analysis that e_1 creates no output markers, or equivalently, $\mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)$ makes no recursive function call, then we can eliminate $@\,\mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)(\$t_1)$ and further simplify the fusion rule. Our preliminary performance analysis reports relatively good evidence of usefulness of this optimization.

The main technical contributions of this paper are two folds:

- A sound static inference of markers that is refined over that in [4] (Section 3). In the prior inference, the set of output markers was inferred using subtyping rule, which could lead to a set that is unnecessarily larger than actually produced at run time. For example, the set of output markers of the body of **rec** was treated as identical to the set of input markers. This



(a) A Simple Graph (b) An Equivalent Graph (c) Result of $a2d\_xc$ on Fig. 1(a)

Fig. 1. Graph Equivalence Based on Bisimulation

over-approximation missed the chance of expression simplification exemplified above. Our inference can avoid this over-approximation by avoiding subtyping rule and computing the sets in a bottom-up manner.
- A set of rewriting rules for optimization using inferred markers (Section 4), that is more powerful than that in [4] in the sense that more expressions are simplified as exemplified above.

All have been implemented and tested with graph transformations widely recognized in software engineering research. The source code of the implementation can be downloaded via our project web site at `www.biglab.org`.

The rest of this paper is organized as follows. Section 2 reviews UnCAL graph model, graph transformation language and existing optimizations. Section 3 proposes enhanced static analysis of markers. In Section 4, we build enhanced rewriting optimization algorithm based on the static analysis. Section 5 reports preliminary performance results. Section 6 reviews related work, and Section 7 concludes this paper.

2 UnCAL Graph Algebra and Prior Optimizations

In this section, we review the UnCAL graph algebra [3, 4], in which our graph transformation is specified.

2.1 Graph Data Model

We deal with rooted, directed, and edge-labeled graphs with no order on outgoing edges. They are edge-labeled in the sense that all information is stored on labels of edges while nodes have no labels. UnCAL graph data model has two prominent features, *markers* and *ε-edges*. Nodes may be marked with *input* and *output markers*, which are used as an interface to connect them to other graphs. An ε-edge represents a shortcut of two nodes, working like the ε-transition in an automaton. We use *Label* to denote the set of labels and $\mathcal{M}$ to denote the set of markers.

Formally, a graph G, sometimes denoted by $G_{(V,E,I,O)}$, is a quadruple (V, E, I, O), where V is a set of nodes, $E \subseteq V \times (\mathit{Label} \cup \{\varepsilon\}) \times V$ is a set of edges, $I \subseteq \mathcal{M} \times V$ is a set of pairs of an input marker and the corresponding node, and $O \subseteq V \times \mathcal{M}$ is a set of pairs of nodes and associated output markers. For each marker $\&x \in \mathcal{M}$, there is at most one node v such that $(\&x, v) \in I$. The node v is called an *input node* with marker $\&x$ and is denoted by $I(\&x)$. Unlike input markers, more than one node can be marked with an identical output marker. They are called *output nodes*. Intuitively, input nodes are root nodes of the graph (we allow a graph to have multiple root nodes, and for singly rooted graphs, we often use default marker & to indicate the root), while an output node can be seen as a "context-hole" of graphs where an input node with the same marker will be plugged later. We write $\mathsf{inMarker}(G)$ to denote the set of input markers and $\mathsf{outMarker}(G)$ to denote the set of output markers in a graph G.

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

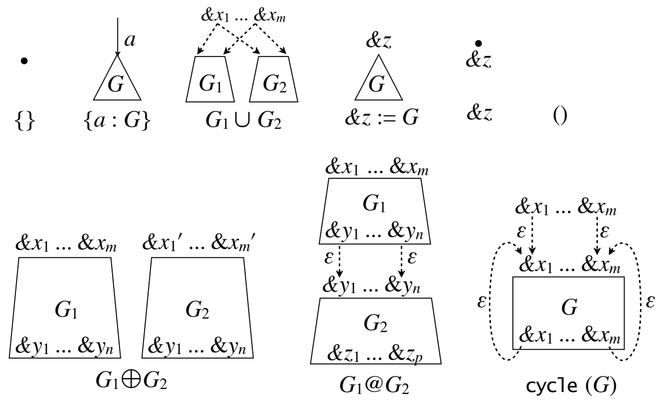
2.2 Notion of Graph Equivalence

Two graphs are value equivalent if they are bisimilar. Please refer to [4] for the complete definition. Informally, graph G_1 is bisimilar to graph G_2 if every node x_1 in G_1 has at least a bisimilar counterpart x_2 in G_2 and vice versa, and if there is an edge from x_1 to y_1 in G_1, then there is a corresponding edge from x_2 to y_2 in G_2 that is a bisimilar counterpart of y_1, and vice versa. Therefore, unfolding a cycle or duplicating shared nodes does not really change a graph. This notion of bisimulation is extended to cope with ε-edges. For instance, the graph in Fig. 1(b) is value equivalent to the graph in Fig. 1(a); the new graph has an additional ε-edge (denoted by the dotted line), duplicates the graph rooted at node 5, and unfolds and splits the cycle at node 4. Unreachable parts are also disregarded, i.e., two bisimilar graphs are still bisimilar if one adds subgraphs unreachable from input nodes.

This value equivalence provides optimization opportunities because we can rewrite transformation so that transformation before and after rewriting produce results that are bisimilar to each other [4]. For example, optimizer can freely cut off expressions that is statically determined to produce unreachable parts.

2.3 Graph Constructors

Figure 2 summarizes the nine graph constructors that are powerful enough to describe arbitrary (directed, edge-labeled, and rooted) graphs [4]. Here, {} constructs a root-only graph, $\{a : G\}$ constructs a graph by adding an edge with



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

Fig. 2. Graph Constructors

label $a \in \mathit{Label} \cup \{\varepsilon\}$ pointing to the root of graph G, and $G_1 \cup G_2$ adds two ε-edges from the new root to the roots of G_1 and G_2. Also, $\&x := G$ associates an input marker, $\&x$, to the root node of G, $\&y$ constructs a graph with a single node marked with one output marker $\&y$, and $()$ constructs an empty graph that has neither a node nor an edge. Further, $G_1 \oplus G_2$ constructs a graph by using a componentwise (V, E, I and O) union. $\cup$ differs from $\oplus$ in that $\cup$ unifies input nodes while $\oplus$ does not. $\oplus$ requires input markers of operands to be disjoint, while $\cup$ requires them to be identical. $G_1 \mathbin{@} G_2$ composes two graphs vertically by connecting the output nodes of G_1 with the corresponding input nodes of G_2 with ε-edges, and $\mathbf{cycle}(G)$ connects the output nodes with the input nodes of G to form cycles. Formal definitions can be found in the full version of [6]. The definition here is based on graph isomorphism (identical graph construction expressions results in identical graphs up to isomorphism), and they are, together with other operators, also bisimulation generic [4], i.e., bisimilar result is obtained for bisimilar operands.

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

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

□

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

Fig. 3. Core UnCAL Language

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

2.4 UnCAL Syntax

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

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

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

and f can be encoded by $\mathbf{rec}(\lambda(\$l, \$g).e)$. Despite its simplicity, the core UnCAL is powerful enough to describe interesting graph transformation including all graph queries (in UnQL) [4], and nontrivial model transformations [7].

Example 2. The following structural recursion *a2d_xc* replaces all labels `a` with `d` and removes edges labeled `c`.

$$
\begin{array}{lll}
\mathit{a2d\_xc}(\$db) = \mathbf{rec}(\lambda(\$l, \$g).\,\mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then} & \{\mathtt{d} : \&\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then} & \{\varepsilon : \&\} & \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else} & \{\$l : \&\}) & (\$db)
\end{array}
$$

The outer **if** of the nested **if**s corresponds to e in the above equations. Applying the function *a2d_xc* to the graph in Fig. 1(a) yields the graph in Fig. 1(c). □

2.5 Revisiting Original Marker Analysis

There were actually previous work on marker analysis by original authors of UnCAL. Figure 6 of Section A.1 in the appendix shows typing rules from the technical report version of [2]. Note that we call type to denote sets of input and output markers. Compared to our analysis, these rules were provided declaratively. For example, the rule for **if** says that if sets of output markers in both branches are equal, then the result have that set of output markers. It is not apparent how we obtain the output marker of **if** if the branches have different sets of output markers.

Buneman et al. [4] did mention optimization based on marker analysis, to avoid evaluating unnecessary subexpressions. But it was mainly based on *run-time* analysis. As we propose in the following sections, we can *statically* compute the set of markers and further simplify the transformation itself.

2.6 Fusion Rules and Output Marker Analysis

Buneman et al. [3, 4] proposed the following fusion rules that aim to remove intermediate results in successive applications of structural recursion **rec**.

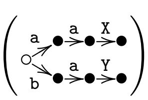
$$
\begin{aligned}
&\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(\mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(e_0)) \\
&= \begin{cases}
\mathbf{rec}(\lambda(\$l_1, \$t_1).\,\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(e_1))(e_0) & \text{if } \$t_2 \text{ does not appear free in } e_2 \\
\begin{array}{l}\mathbf{rec}(\lambda(\$l_1, \$t_1).\,\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2) \\ \quad (e_1 \mathbin{@} \mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(\$t_1)))(e_0)\end{array} & \text{for arbitrary } e_2
\end{cases}
\end{aligned}
\tag{1}
$$

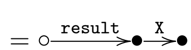
If you can statically guarantee that e_1 does not produce any output marker, which means the **rec** is "non-recursive", then the second rule is promoted to the first rule, opening another optimization opportunities.

Non-recursive Query. Now questions that might be asked would be how often do such kind of "non-recursive" queries appear. Actually it frequently appears as *extraction* or *join*. Extraction transformation is a transformation in which some subgraphs are simply extracted. It is achieved by direct reference of the bound graph variable in the body of **rec**. Join is achieved by nesting of these extraction transformations. Finite steps of edge traversals are expressed by this nesting.

Example 3. The following structural recursion *consecutive* extracts subgraphs that can be accessible by traversing two connected edges of the same label.

$$
\begin{aligned}
\mathit{consecutive}(\$db) = \mathbf{rec}(\lambda(\$l, \$g).\,&\mathbf{rec}(\lambda(\$l', \$g'). \\
&\mathbf{if}\ \$l = \$l'\ \mathbf{then}\ \{\mathtt{result} : \$g'\} \\
&\mathbf{else}\ \qquad\qquad \{\} \qquad\qquad)(\$g))(\$db)
\end{aligned}
$$

For example, we have *consecutive* (

) = 

.

If this transformation is followed by $\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)$ where e_2 refers to $\$t_2$, the second condition of fusion rule applies, but it will be promoted to the first, since the body of **rec** in *consecutive*, which corresponds to e_1 in the fusion rule, does not have output markers. We revisit this case in Example 4 in Section 4.

$$\&x := (\&z := e) \longrightarrow \&x.\&z := e \qquad \&x := (e_1 \oplus e_2) \longrightarrow (\&x := e_1) \oplus (\&x := e_2)$$
$$e \cup \{\} \longrightarrow e \qquad \{\} \cup e \longrightarrow e \qquad e \oplus () \longrightarrow e \qquad () \oplus e \longrightarrow e$$
$$() \mathbin{@} e \longrightarrow () \qquad \frac{e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \mathcal{X} \cap \mathcal{Y} = \phi}{\mathbf{cycle}(e) \longrightarrow e}$$

Fig. 4. Auxiliary Rewriting Rules

2.7 Other Prior Rewriting Rules

Apart from fusion rules, the following rewriting rules for **rec** are proposed in [4] for optimizations. Type of e is assumed to be $DB^{\mathcal{Z}}_{\mathcal{Z}}$. They simplify the argument of **rec** and increase chances of fusions.

$$\begin{array}{ll}
\mathbf{rec}(\lambda(\$l,\$t).e)(\{\}) & = {}^1 \bigoplus_{\&z \in \mathcal{Z}} \&z := \{\} \\
\mathbf{rec}(\lambda(\$l,\$t).e)(\{l : d\}) & = e[{}^{l}/_{\$l}][{}^{d}/_{\$t}] \mathbin{@} \mathbf{rec}(\lambda(\$l,\$t).e)(d) \\
\mathbf{rec}(\lambda(\$l,\$t).e)(d_1 \cup d_2) & = \mathbf{rec}(\lambda(\$l,\$t).e)(d_1) \cup \mathbf{rec}(\lambda(\$l,\$t).e)(d_2) \\
\mathbf{rec}(\lambda(\$l,\$t).e)(\&x := d) & = \&x := {}^2(\mathbf{rec}(\lambda(\$l,\$t).e)(d)) \\
\mathbf{rec}(\lambda(\$l,\$t).e)(\&x) & = \bigoplus_{\&z \in \mathcal{Z}} \&z := \&y.\&z \\
\mathbf{rec}(\lambda(\$l,\$t).e)() & = () \\
\mathbf{rec}(\lambda(\$l,\$t).e)(d_1 \oplus d_2) & = \mathbf{rec}(\lambda(\$l,\$t).e)(d_1) \oplus \mathbf{rec}(\lambda(\$l,\$t).e)(d_2)
\end{array}$$

The first rule eliminates **rec**, while the second rule eliminates an edge from the argument.

$$\frac{\$t \text{ does not occur free in } e}{\mathbf{rec}(\lambda(\$l,\$t).e)(d_1 \mathbin{@} d_2) = \mathbf{rec}(\lambda(\$l,\$t).e)(d_1) \mathbin{@} \mathbf{rec}(\lambda(\$l,\$t).e)(d_2)}$$

$$\frac{\$t \text{ does not occur free in } e}{\mathbf{rec}(\lambda(\$l,\$t).e)(\mathbf{cycle}(d)) = \mathbf{cycle}(\mathbf{rec}(\lambda(\$l,\$t).e)(d))}$$

Additional rules proposed by (full version of) Hidaka et al. [7] to further simplify the body of **rec** are given in Fig. 4. The rules in the last line in Fig. 4 can be generalized by static analysis of the marker in the following section. And given the static analysis, we can optimize further as described in Section 4.

3 Enhanced Static Analysis

This section proposes our enhanced marker analysis. Figure 5 shows the proposed marker *inference* rules for UnCAL. Dot notation $(\cdot)$ between markers and sets of markers represents "concatenation" of markers that satisfies the properties at the top of the figure. Static environment Γ denotes mapping from variables to their types. We assume that the types of free variables are given. Since we focus on graph values, we omit rules for labels. Roughly speaking, $DB^{\mathcal{X}}_{\mathcal{Y}}$ is a type for graphs that have $\mathcal{X}$ input markers exactly and have at most $\mathcal{Y}$ output markers, which will be shown formally by Lemma 1.

[1] Original right hand side was {} in [4], but we corrected here.

[2] We overload := in $\&x := g$ to denote renaming of each input marker $\&z$ in g to $\&x.\&z$.

$$(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z) \qquad \& \cdot \&x = \&x \cdot \& = \&x \qquad \mathcal{X} \cdot \mathcal{Y} \stackrel{\text{def}}{=} \{\&x \cdot \&y \mid \&x \in \mathcal{X}, \&y \in \mathcal{Y}\}$$

$$\frac{}{\Gamma \vdash \{\} :: DB_{\emptyset}} \qquad \frac{\Gamma \vdash l :: \mathit{Label} \quad \Gamma \vdash e :: DB_{\mathcal{Y}}}{\Gamma \vdash \{l : e\} :: DB_{\mathcal{Y}}} \qquad \frac{\Gamma \vdash e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1} \quad \Gamma \vdash e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_2}}{\Gamma \vdash e_1 \cup e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}}$$

$$\frac{}{\Gamma \vdash () :: DB^{\emptyset}_{\emptyset}} \qquad \frac{\Gamma \vdash e :: DB^{\mathcal{Z}}_{\mathcal{Y}}}{\Gamma \vdash \&x := e :: DB^{\{\&x\} \cdot \mathcal{Z}}_{\mathcal{Y}}} \qquad \frac{}{\Gamma \vdash \&y :: DB_{\{\&y\}}}$$

$$\frac{\Gamma \vdash e_1 :: DB^{\mathcal{X}_1}_{\mathcal{Y}_1} \quad \Gamma \vdash e_2 :: DB^{\mathcal{X}_2}_{\mathcal{Y}_2} \quad \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset}{\Gamma \vdash e_1 \oplus e_2 :: DB^{\mathcal{X}_1 \cup \mathcal{X}_2}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}} \qquad \frac{\Gamma \vdash e_1 :: DB^{\mathcal{X}_1}_{\mathcal{Y}_1} \quad \Gamma \vdash e_2 :: DB^{\mathcal{X}_2}_{\mathcal{Y}_2}}{\Gamma \vdash e_1 \mathbin{@} e_2 :: DB^{\mathcal{X}_1}_{\mathcal{Y}_2}}\,\text{[5]} \qquad \frac{\Gamma \vdash e :: DB^{\mathcal{X}}_{\mathcal{Y}}}{\Gamma \vdash \mathbf{cycle}(e) :: DB^{\mathcal{X}}_{\mathcal{Y} \setminus \mathcal{X}}}$$

$$\frac{\Gamma(\$g) = DB^{\mathcal{X}}_{\mathcal{Y}}}{\Gamma \vdash \$g :: DB^{\mathcal{X}}_{\mathcal{Y}}} \qquad \frac{\Gamma \vdash e_{\mathrm{a}} :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \Gamma\{\$l \mapsto \mathit{Label}, \$g \mapsto DB_{\mathcal{Y}}\} \vdash e_b :: DB^{\mathcal{Z}_i}_{\mathcal{Z}_o} \quad \mathcal{Z} = \mathcal{Z}_i \cup \mathcal{Z}_o}{\Gamma \vdash \mathbf{rec}(\lambda(\$l, \$g).e_{\mathrm{b}})(e_{\mathrm{a}}) :: DB^{\mathcal{X} \cdot \mathcal{Z}}_{\mathcal{Y} \cdot \mathcal{Z}}}$$

$$\frac{\Gamma \vdash l_1 :: \mathit{Label} \quad \Gamma \vdash l_2 :: \mathit{Label} \quad \Gamma \vdash e_{\mathrm{t}} :: DB^{\mathcal{X}}_{\mathcal{Y}_{\mathrm{t}}} \quad \Gamma \vdash e_{\mathrm{f}} :: DB^{\mathcal{X}}_{\mathcal{Y}_{\mathrm{f}}}}{\Gamma \vdash \mathbf{if}\ l_1 = l_2\ \mathbf{then}\ e_{\mathrm{t}}\ \mathbf{else}\ e_{\mathrm{f}} :: DB^{\mathcal{X}}_{\mathcal{Y}_{\mathrm{t}} \cup \mathcal{Y}_{\mathrm{f}}}} \qquad \frac{\Gamma \vdash e_1 :: DB^{\mathcal{X}_1}_{\mathcal{Y}_1} \quad \Gamma\{\$g \mapsto DB^{\mathcal{X}_1}_{\mathcal{Y}_1}\} \vdash e_2 :: DB^{\mathcal{X}_2}_{\mathcal{Y}_2}}{\Gamma \vdash \mathbf{let}\ \$g = e_1\ \mathbf{in}\ e_2 :: DB^{\mathcal{X}_2}_{\mathcal{Y}_2}}$$

Fig. 5. UnCAL Static Typing (Marker Inference) Rules: Rules for *Label* are Omitted

The original typing rules were provided based on the subtyping rule

$$\frac{\Gamma \vdash e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \mathcal{Y} \subseteq \mathcal{Y}'}{\Gamma \vdash e :: DB^{\mathcal{X}}_{\mathcal{Y}'}}$$

and required the arguments of $\cup$, $\oplus$, **if** to have identical sets of output markers. Unlike the original rules, the proposed type system does not use the subtyping rule directly for inference. Combined with the forward evaluation semantics $\mathcal{F}[\![]\!]$ that is summarized in [6], we have the following type safety property.

Lemma 1 (Type Safety). *Assume that g is the graph obtained by $g = \mathcal{F}[\![e]\!]$ for an expression e. Then, $\vdash e :: DB^{\mathcal{X}}_{\mathcal{Y}}$ implies both $\mathsf{inMarker}(g) = \mathcal{X}$ and $\mathsf{outMarker}(g) \subseteq \mathcal{Y}$.*

Lemma 1 guarantees that the set of input markers estimated by the type inference is exact in the sense that the set of input markers generated by evaluation exactly coincides with that of the inferred type. For the output markers, the type system provides an over-approximation in the sense that the set of output

[5] Original rule (let's say $@_o$) which requires $\mathcal{Y}_1 = \mathcal{X}_2$ is relaxed here. Our @ can be defined by $g_1 \mathbin{@} g_2 = (g_1 \mathbin{@_o} \mathsf{Id}_{\mathcal{X}_2 \setminus \mathcal{Y}_1}) \mathbin{@_o} (\mathsf{Bot}_{\mathcal{Y}_1 \setminus \mathcal{X}_2} \oplus g_2)$, where Bot and Id are defined in Section 4.2.1. This particular definition in which markers $\mathcal{Y}_1 \setminus \mathcal{X}_2$ are peeled off is close to the original semantics because final output markers coincide. Extension in which these excess output markers remain would be possible, allowing the markers to be used later to connect to other graphs.

markers generated by evaluation is a subset of the inferred set of output markers. Since the treatment of the input markers are identical to that in [4], we focus that on the output markers and prove it. The proof, which is based on induction on the structure of UnCAL expressions, is in Sect. A.2 in the appendix.

Between the original typing rules in [4], the following property holds: for all $\mathcal{X}$ and $\mathcal{Y}$, $e :: DB^{\mathcal{X}}_{\mathcal{Y}}$ for some $\mathcal{Y}' \supseteq \mathcal{Y}$ if and only if e has a type $DB^{\mathcal{X}}_{\mathcal{Y}'}$ in the original type system. The proof can be conducted by simple induction on the structure of the UnCAL expressions and appears in Sect. A.3 of the appendix.

4 Enhanced Rewiring Optimization

This section proposes enhanced rewriting optimization rules based on the static analysis shown in the previous section.

4.1 Rule for @ and Revised Fusion Rule

Statically-inferred markers enable us to optimize expressions much more. We can generalize, for example, the rewriting rule $() \mathbin{@} e \longrightarrow ()$ in the last row of Fig. 4 to the following, by not just referring to the pattern of subexpressions but its estimated markers.

$$\frac{e_1 :: DB^{\mathcal{X}}_{\emptyset}}{e_1 \mathbin{@} e_2 \longrightarrow e_1} \tag{2}$$

As we have seen in Sect. 2, we have two fusion rules (1) for **rec**. Although the first rule can be used to gain performance, the second rule is more complex so less performance gain is expected. Using (2), we can relax the first condition of the fusion rule (1) to increase chances to apply the first rule as follows.

$$\begin{aligned}&\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(\mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(e_0))\\ &\quad = \mathbf{rec}(\lambda(\$l_1, \$t_1).\,\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(e_1))(e_0)\\ &\qquad\quad \text{if } \$t_2 \text{ does not appear free in } e_2,\ \underline{\text{or } e_1 :: DB^{\mathcal{X}}_{\emptyset}}\end{aligned}$$

Here, the underlined part is added to relax the entire condition.

4.2 Further Optimization with Static Marker Information

In this section, general rules for $e_1 \mathbin{@} e_2$ is investigated. First to eliminate $@\, e_2$, and then to statically compute @ by plugging e_2 into e_1.

4.2.1 Static Output Marker Removal Algorithm and Soundness

For more general cases of @ where connections by ε do not happen, we have the following rule.

$$\frac{e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1} \quad e_2 :: DB^{\mathcal{Y}_2}_{\mathcal{Z}} \quad \mathcal{Y}_1 \cap \mathcal{Y}_2 = \emptyset \quad \mathsf{Rm}_{\mathcal{Y}_1}\langle\!\langle e_1 \rangle\!\rangle = e}{e_1 \mathbin{@} e_2 \longrightarrow e}$$

$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle$ denotes static removal of the set of output markers, i.e., if $\vdash e :: DB^{\mathcal{X}}_{\mathcal{Y}}$, then $\vdash \mathsf{Rm}_{\mathcal{W}}\langle\!\langle e\rangle\!\rangle :: DB^{\mathcal{X}}_{\mathcal{Y}\setminus\mathcal{W}}$. Without this, rewriting result in spurious output markers from e_1 remained in the final result. The formal definition of $\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle$ is shown below.

$$\mathsf{Rm}_{\emptyset}\langle\!\langle e\rangle\!\rangle = e \quad \mathsf{Rm}_{\mathcal{X}\cup\mathcal{Y}}\langle\!\langle e\rangle\!\rangle = \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle \mathsf{Rm}_{\mathcal{X}}\langle\!\langle e\rangle\!\rangle\rangle\!\rangle \quad \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle \{\}\rangle\!\rangle = \{\}$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle ()\rangle\!\rangle = () \quad \mathsf{Rm}_{\{\&y\}}\langle\!\langle \&y\rangle\!\rangle = \{\} \quad \mathsf{Rm}_{\{\&y\}}\langle\!\langle \&x\rangle\!\rangle = \&x$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_1 \odot e_2\rangle\!\rangle = \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_1\rangle\!\rangle \odot \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_2\rangle\!\rangle \quad (\odot \in \{\cup, \oplus\})$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle \&x := e\rangle\!\rangle = (\&x := \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle)$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle \{l : e\}\rangle\!\rangle = \{l : \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle\}$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_1 \mathbin{@} e_2\rangle\!\rangle = e_1 \mathbin{@} \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_2\rangle\!\rangle$$
$$\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle\!\rangle = \mathbf{if}\ b\ \mathbf{then}\ \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_1\rangle\!\rangle\ \mathbf{else}\ \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e_2\rangle\!\rangle$$

Since the output markers of the result of $e_1 \mathbin{@} e_2$ are not affected by those of e_1, e_1 is not visited in the rule of @. In the following, $\mathsf{Id}_{\mathcal{Y}}$ and $\mathsf{Bot}_{\mathcal{Y}}$ are respectively defined as $\bigoplus_{\&z\in\mathcal{Y}} \&z := \&z$ and $\bigoplus_{\&z\in\mathcal{Y}} \&z := \{\}$.

$$\frac{e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \&y \in (\mathcal{Y}\setminus\mathcal{X}) \quad \mathsf{Rm}_{\{\&y\}}\langle\!\langle e\rangle\!\rangle = e'}{\mathsf{Rm}_{\{\&y\}}\langle\!\langle \mathbf{cycle}(e)\rangle\!\rangle = \mathbf{cycle}(e')} \quad \frac{e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \&y \notin (\mathcal{Y}\setminus\mathcal{X})}{\mathsf{Rm}_{\{\&y\}}\langle\!\langle \mathbf{cycle}(e)\rangle\!\rangle = \mathbf{cycle}(e)}$$

$$\frac{\$v :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \&y \notin \mathcal{Y}}{\mathsf{Rm}_{\{\&y\}}\langle\!\langle \$v\rangle\!\rangle = \$v} \quad \frac{\$v :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \&y \in \mathcal{Y}}{\mathsf{Rm}_{\{\&y\}}\langle\!\langle \$v\rangle\!\rangle = \$v \mathbin{@} (\mathsf{Bot}_{\{\&y\}} \oplus \mathsf{Id}_{\mathcal{Y}\setminus\{\&y\}})}$$

The first rule of $\$v$ says that according to the safety of type inference, $\&y$ is guaranteed not to result at run-time, so the expression $\$v$ remains unchanged. The second rule actually removes the output marker $\&y_j$, but static removal is impossible. So the removal is deferred till run-time. The output node marked $\&y_j$ is connected to node produced by $\&y := \{\}$. Since the latter node has no output marker, the original output marker disappears from the graph produced by the evaluation. The rest of the $\&y_k := \&y_k$ does no operation on the marker. Since estimation $\mathcal{Y}$ is the upper bound, the output maker may not be produced at run-time. If it is the case, connection with ε-edge by @ does not occur, and the nodes produced by the := expressions are left unreachable, so the transformation is still valid. As another side effect, @ may connect identically marked output nodes to single node. However, the graph before and after this “funneling” connection are bisimilar, since every leaf node with identical output markers are bisimilar by definition. Should the output nodes are to be further connected to other input nodes, the target node is always single, because more than one node with identical input marker is disallowed by the data model. So this connection does no harm. Note that the second rule increases the size of the expression, so it may increase the cost of evaluation.

$$\frac{\mathbf{rec}(\lambda(\$l,\$t).e_{\mathrm{b}})(e_{\mathrm{a}}) :: DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}} \quad \&y \in \mathcal{Y} \quad \mathsf{Rm}_{\{\&y\}}\langle\!\langle e_{\mathrm{a}}\rangle\!\rangle = e'_{\mathrm{a}}}{\mathsf{Rm}_{\{\&y.\&z \mid \&z\in\mathcal{Z}\}}\langle\!\langle \mathbf{rec}(\lambda(\$l,\$t).e_{\mathrm{b}})(e_{\mathrm{a}})\rangle\!\rangle = \mathbf{rec}(\lambda(\$l,\$t).e_{\mathrm{b}})(e'_{\mathrm{a}})}$$

For **rec**, one output marker $\&y$ in e_{a} corresponds to $\{\&y\} \cdot \mathcal{Z} = \{\&y.\&z \mid \&z \in \mathcal{Z}\}$ in the result. So removal of $\&y$ from e_{a} results in removal of all of the $\{\&y\} \cdot \mathcal{Z}$. So only removal of all of $\{\&y.\&z \mid \&z \in \mathcal{Z}\}$ at a time is allowed.

Lemma 2 (Soundness of Static Output-Marker Removal Algorithm). *Assume that $G = (V, E, I, O)$ is a graph obtained by $G = \mathcal{F}[\![e]\!]$ for an expression e, and e' is the expression obtained by $\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle$. Then, we have $\mathcal{F}[\![e']\!] = (V, E, I, \{(v, \&y) \in O \mid \&y \notin \mathcal{Y}\})$.*

Lemma 2 guarantees that no output marker in $\mathcal{Y}$ appears at run-time if $\mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle$ is evaluated.

4.2.2 Plugging Expression to Output Marker Expression

The following rewriting rule is to plug an expression into another through correspondingly marked node.

$$\{l : \&y\} \mathbin{@} (\&y := e) \longrightarrow \{l : e\}$$

This kind of rewriting was actually implicitly used in the exemplification of optimization in [4], but was not generalized. We can generalize this rewriting as

$$e \mathbin{@} (\&y := e') \longrightarrow \begin{cases} \mathsf{Rm}_{\mathcal{Y}\setminus\{\&y\}}\langle\!\langle e\rangle\!\rangle[{}^{e'}\!/\!{}_{\&y}] & \text{if } \&y \in \mathcal{Y} \text{ where } e :: DB^{\mathcal{X}}_{\mathcal{Y}} \\ \mathsf{Rm}_{\mathcal{Y}}\langle\!\langle e\rangle\!\rangle & \text{otherwise.} \end{cases}$$

where $e[{}^{e'}\!/\!{}_{\&y}]$ denotes substitution of $\&y$ by e' in e. Since nullrary constructors $\{\}$, $()$, and $\&x \neq \&y$ do not produce output marker $\&y$, the substitution takes no effect and the rule in the latter case apply. So we focus on the former case in the sequel. For most of the constructors the substitution rules are rather straightforward:

$$\begin{aligned} \&y[{}^{e}\!/\!{}_{\&y}] &= e \\ (e_1 \odot e_2)[{}^{e}\!/\!{}_{\&y}] &= (e_1[{}^{e}\!/\!{}_{\&y}]) \odot (e_2[{}^{e}\!/\!{}_{\&y}]) \quad (\odot \in \{\cup, \oplus\}) \\ (\&x := e)[{}^{e'}\!/\!{}_{\&y}] &= (\&x := (e[{}^{e'}\!/\!{}_{\&y}])) \\ \{l : e\}[{}^{e'}\!/\!{}_{\&y}] &= \{l : (e[{}^{e'}\!/\!{}_{\&y}])\} \\ (e_1 \mathbin{@} e_2)[{}^{e}\!/\!{}_{\&y}] &= e_1 \mathbin{@} (e_2[{}^{e}\!/\!{}_{\&y}]) \\ (\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2)[{}^{e}\!/\!{}_{\&y}] &= \textbf{if } b \textbf{ then } (e_1[{}^{e}\!/\!{}_{\&y}]) \textbf{ else } (e_2[{}^{e}\!/\!{}_{\&y}]) \end{aligned}$$

Since the final output marker for @ is not affected by that of e_1, e_1 is not visited in the rule of @. For **cycle**, we should be careful to avoid capturing of marker.

$$\textbf{cycle}(e)[{}^{e'}\!/\!{}_{\&y}] = \begin{cases} \overline{\textbf{cycle}}(e[{}^{e'}\!/\!{}_{\&y}]) & \text{if } (\mathcal{Y}' \cap \mathcal{X}) = \emptyset \text{ where } e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad e' :: DB_{\mathcal{Y}'} \\ \overline{\textbf{cycle}}(e)[{}^{e'}\!/\!{}_{\&y}] & \text{otherwise.} \end{cases}$$

The above rule says that if $\mathcal{Y}'$ will be "free" markers in e, that is, the output markers in e', namely $\mathcal{Y}'$ will not be captured by **cycle**, then we can plug e' into output marker expression in e. If some of the output markers in $\mathcal{Y}'$ are included in $\mathcal{X}$, then the renaming is necessary. As suggested in the full version of [3], markers in $\mathcal{X}$ instead of those in $\mathcal{Y}'$ should be renamed. And that renaming can be compensated outside of **cycle** as follows:

$$\overline{\mathbf{cycle}}(e) \stackrel{\text{def}}{=} (\bigoplus_{\&x \in \mathcal{X}} \&x := \&tmp_x) \mathbin{@} \mathbf{cycle}(e[{}^{\&tmp_{x_1}}/_{\&x_1}] \ldots [{}^{\&tmp_{x_M}}/_{\&x_M}])$$

where $\&x_1, \ldots, \&x_M = \mathcal{X}$ are the markers to be renamed, and $\mathcal{X}$ of $e :: DB^{\mathcal{X}}_{\mathcal{Y}}$ is used. Note that in the renaming, not only output markers, but also input markers are renamed. $\&tmp_{x_1}, \ldots, \&tmp_{x_M}$ are corresponding fresh (temporary) markers. The left hand side of @ recovers the original name of the markers. After renaming by $\overline{\mathbf{cycle}}$, no marker is captured anymore, so substitution is guaranteed to succeed. For variable reference and **rec**, static substitution is impossible. So we resort to the following generic "fall back" rule.

$$\frac{e \in \{\$v, \mathbf{rec}(\_)(\_)\} \quad e :: DB^{\mathcal{X}}_{\mathcal{Y}} \quad \mathcal{Y} = \{\&y_1, \ldots, \&y_j, \ldots, \&y_n\}}{e[{}^{e'}/_{\&y_j}] = e \mathbin{@} \begin{pmatrix} \&y_1 := \&y_1, \ldots, \&y_{j-1} := \&y_{j-1}, \&y_j := e', \\ \&y_{j-1} := \&y_{j-1}, \ldots, \&y_n := \&y_n \end{pmatrix}}$$

The "fall back" rule is used for **rec** because unlike output marker removal algorithm, we can not just plug e into e_{a} since that will not plug e but $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(e)$ in the result. We could have used the inverse $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})^{-1}$ to plug $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})^{-1}(e')$ instead, but the inverse does not always exist in general.

The overall rewriting is conducted by two mutually recursive functions as follows: a driver function first applies itself to subexpressions recursively, and then applies a function that implements $\longrightarrow$ and other rewriting rules recursively such as fusions described in this paper, on the result of the driver function. The implemented rewriting system is deterministic by imposing consistent order of rule applications by these functions.

With respect to proposed rewriting rules in this section, the following theorem holds.

Theorem 1 (Soundness of Rewriting). *If $e \longrightarrow e'$, then $\mathcal{F}[\![e]\!]$ is bisimilar to $\mathcal{F}[\![e']\!]$.*

It can be proved by simple induction on the structure of UnCAL expressions, and omitted here.

Example 4. The following transformation that apply selection after *consecutive* in Example 3

$\mathbf{rec}(\lambda(\$l_1, \$g_1).\, \mathbf{if}\ \$l_1 = \mathtt{a}\ \mathbf{then}\ \{\$l_1 : \$g_1\}\ \mathbf{else}\ \{\})(\mathit{consecutive}(\$db))$

is rewritten as follows:

$$\begin{array}{l} = \quad \{ \text{ expand definition of } \textit{consecutive} \text{ and apply 2nd fusion rule } \} \\ \quad \mathbf{rec}(\lambda(\$l, \$g).\, \mathbf{rec}(\lambda(\$l_1, \$g_1).\, \mathbf{if}\ \$l_1 = \mathtt{a}\ \mathbf{then}\ \{\$l_1 : \$g_1\}\ \mathbf{else}\ \{\}) \\ \quad\quad (\mathbf{rec}(\lambda(\$l', \$g').\, \mathbf{if}\ \$l = \$l'\ \mathbf{then}\ \{\mathtt{result} : \$g'\}\ \mathbf{else}\ \{\})(\$g) \\ \quad\quad\quad @\, \mathbf{rec}(\lambda(\$l, \$g).\, \mathbf{rec}(\lambda(\$l', \$g'). \\ \quad\quad\quad\quad \mathbf{if}\ \$l = \$l'\ \mathbf{then}\ \{\mathtt{result} : \$g'\}\ \mathbf{else}\ \{\})(\$g))(\$g)))(\$db) \end{array}$$

$$
\begin{aligned}
= &\quad \{\ (2)\ \} \\
&\mathbf{rec}(\lambda(\$l,\$g).\,\mathbf{rec}(\lambda(\$l_1,\$g_1).\,\mathbf{if}\ \$l_1 = \mathtt{a}\ \mathbf{then}\ \{\$l_1 : \$g_1\}\ \mathbf{else}\ \{\}) \\
&\qquad (\mathbf{rec}(\lambda(\$l',\$g').\,\mathbf{if}\ \$l = \$l'\ \mathbf{then}\ \{\mathtt{result} : \$g'\}\ \mathbf{else}\ \{\})(\$g)))(\$db) \\
= &\quad \{\ \text{2nd fusion rule, (2), } \mathbf{rec} \text{ rule for } \mathbf{if} \text{ and } \{l : d\}, \text{ static label comparison }\} \\
&\mathbf{rec}(\lambda(\$l,\$g).\,\mathbf{rec}(\lambda(\$l',\$g').\{\})(\$g))(\$db)
\end{aligned}
$$

This example demonstrates the second fusion rule promotes to the first. The top level edges of the result of *consecutive* are always labeled `result` while the selection selects subgraphs under edges labeled `a`. So the result will always be empty, and correspondingly the body of **rec** in the final result is {}. □

More examples can be found in Sect. A.4 in the appendix.

The following remark summarizes how far can we remove intermediate graphs. Proof can be found in Section A.5 in the appendix.

Remark 1 (Removal of Intermediage Graph). Suppose we have a composition of the form

$$\mathbf{rec}(\lambda(\$l_2,\$t_2).e_2)(C[\mathbf{rec}(\lambda(\$l_1,\$t_1).e_1)(e_0)])$$

where $C[]$ denotes context using constructors and if expressions. Then, (i) if $\$t_2$ does not appear free in e_2, then the composition of the above form, including the ones that are generated during fusion, are removed. (ii) if $\$t_2$ appears free in e_2 but $e_1 :: DB_{\emptyset}^{\bar{}}$, and e_1 consists of nested **rec** with context not using @ or **cycle** with body of type $DB_{\emptyset}^{\bar{}}$, then the composition, including the ones that are generated during the fusion rule application, are removed.

5 Implementation and Performance Evaluation

This section reports preliminary performance evaluations. All of the transformations in the paper are implemented in GRoundTram, or Graph Roundtrip Transformation for Models, which is a system to build a bidirectional transformation between two models (graphs). All the source codes are available online at `www.biglab.org`. The following experimental results are obtained by the system.

Performance evaluation was conducted on GRoundTram, running on MacOSX over MacBookPro 17 inch, with 3.06 GHz Intel Core 2 Duo CPU. An UnCAL transformation runs in time exponential to the size (number of compositions or nesting of **rec**s) of the transformation (and polynomial to the size of input graph [4]). Thus, the proposed rewriting, which can reduce the size of transformation, may change the elapsed time drastically even for the small graphs (up to a hundred of nodes) used in the experiments.

Table 1 shows the experimental results. Each running time includes time for forward and backward[6] transformations [6], and for backward transformations, algorithm for edge-renaming is used, and no modification on the target is

[6] Since we are conducting research on bidirectional transformations, we are not only interested in the performance of forward transformations, but also that of backward transformations.

Table 1. Summary of Experiments (running time is in CPU seconds)

	direction	no rewriting	previous [4, 7]	ours
Class2RDB	forward	1.18	0.68	0.68
	backward	14.5	7.99	7.89
PIM2PSM	forward	0.08	0.77 (2*3)	0.07 (2*13)
	backward	1.62	3.64	0.75
C2Osel	forward	0.04	0.04 (2*1)	0.05 (2*11)
	backward	2.26	0.26	0.27
C2Osel'	forward	0.05	0.06 (2*1)	0.04 (2*11)
	backward	2.53	2.58	1.26
Ex1 [4]	forward	0.022	0.016 (1*1)	0.010 (1*1)
	backward	0.85	0.30	0.15

actually given. However, we suppose presence of modification would not make much difference in the running time. Running time of forward transformation in which rewriting is applied (last two columns) includes time for rewriting. Rewriting took 0.006 CPU seconds at the worst case (*PIM2PSM*, ours). *Class2RDB* stands for class diagram to table diagram transformation, *PIM2PSM* for platform independent model to platform specific model transformation, *C2Osel* is for transformation of customer oriented database into order oriented database, followed by a simple selection, and *Ex1* is the example that is extracted from our previous paper [7], which was borrowed from [4]. It is a composition of two **rec**s. Concrete plugging optimizations in this example can be traced in Sect. A.4 in the appendix.

The numbers in parentheses show how often the fusion transformation happened. For example, *PIM2PSM* led to 3 fusions based on the second rule, and further enhanced rewriting led to 10 more fusion rule applications, all of which promoted to the first rule via proposed rewriting rule (2). Same promotions happened to *C2Osel*. Except for *C2Osel'*, a run-time optimization in which unreachable parts are removed after every application of **rec** is applied. Enhanced rewriting led to performance improvements in both forward and backward evaluations, except *C2Osel*. Comparing “previous” with “no rewriting”, *PIM2PSM* and *C2Osel'* led to slowdown. This slowdown is explained as follows. The fusion turns composition of **rec**s to their nesting. In the presence of the run-time optimization, composition is more advantageous than nesting when only small part of the result is passed to the subsequent **rec**s, which will run faster than when passed entire results (including unreachable parts). Once nested, intermediate result is not produced, but the run-time optimization is suppressed because every execution of the inner **rec** traverses the input graph. *C2Osel'* in which runtime optimization is turned off, shows that the enhanced rewriting itself lead to performance improvements.

6 Related Work

Although some of our optimization rules were mentioned in [7], the relationship with static marker analysis was not covered in depth. Our optimization, based on the enhanced marker analysis in Sect. 3 , generalizes all the rules in [7] uniformly.

In our previous paper [6], an implementation of rewriting optimizations was reported, but concrete strategies were not included in the paper.

Plugging constructor-only expressions into output marker expressions was discussed in the full (technical report) version of [3]. Their motivation was to express semantics of @ at the constructor expression level and not graph data level as in [4]. It also mentioned renaming of markers to avoid capture of the output markers in **cycle** expressions[7]. We do attempt the same thing at the expression level but we argue here more formally.

Our rewriting rules are inspired by the technical report but the idea there is not yet exploited fully. They discussed the semantics of **rec** on the **cycle** expressions, even when the body refered to graph variables, although marker environment that maps markers to connected subgraphs introduced there makes the semantics complex. But we could use the semantics to enhance rewriting rules for **rec** with **cycle** arguments.

The journal version [4] mentioned run-time optimization in which, assuming top-down evaluation, only necessary components of structural recursion are executed. For example, only $\&z_1$ component of **rec** in $\&z_1 \mathbin{@} \mathbf{rec}(\_)(\_)$ is evaluated. It is not applicable to our bidirectional settings which rely on bulk semantics [6].

A static analysis of UnCAL was described in [1], but the main motivation was to analyze the structure of graphs using graph schema, whereas our analysis focus on the connectivity of graphs.

7 Conclusion

In this paper, under the context of graph transformation using UnCAL graph algebra, enhanced static marker inference is first formalized. Fusion rule becomes more powerful thanks to the static marker analysis. Further rewriting rules based on this inference are also explored. Marker renaming for capture avoidance is formalized to support the rewriting rules. Under the context of bidirectional graph transformations [6], one of the advantage of static analysis is that we can keep implementation of bidirectional interpreter intact. The marker analysis and rewriting proposed can be considered as dead-code detection and elimination. We believe this technique can be used for other graph languages that based on graph model that have named connecting points like input/output nodes. Preliminary performance evaluation shows the usefulness of the optimization for various non-trivial transformations in the field of software engineering research.

Future work under this context includes reasoning about effects on the backward updatability. Although rewriting is sound with respect to well-behavedness

[7] In the technical report, cycle was represented by parallel equations, without **cycle** operator in current UnCAL form.

of bidirectional transformations, backward transformation before and after rewriting may accept different update operations. Our conjecture is that simplified transformation accepts more updates, but this argument requires further discussions.

Acknowledgments We thank reviewers and Kazuyuki Asada for their thorough comments on the earlier versions of the paper. The research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from the National Institute of Informatics, Encouragement of Young Scientists (B) of the Grant-in-Aid for Scientific Research No. 20700035 and Grant-in-Aid for Research Activity Start-up No. 22800003.

References

1. A. A. Benczúr and B. Kósa. Static analysis of structural recursion in semistructured databases and its consequences. In *ADBIS*, pages 189–203, 2004.
2. P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, volume 1186 of *LNCS*, pages 336–350, 1997.
3. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516, 1996. long version appears as U.Penn TR MS-CIS-96-09.
4. P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB J.*, 9(1):76–110, 2000.
5. K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. Presented at *MTiP 2005*. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf, 2005.
6. S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.
7. S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *SAC 2009*, pages 468–475, 2009.
8. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
9. I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano. Toward bidirectionalization of ATL with GRoundTram. In *ICMT*, pages 138–151, June 2011.
10. P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MoDELS 2007*, pages 1–15, 2007.
11. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

A Appendix

A.1 UnCAL Original Static Typing Rules

$$\frac{a \in \mathcal{U}}{a : \mathit{Label}} \quad \frac{y \text{ a label variable}}{y : \mathit{Label}} \quad \frac{t \text{ a tree variable of type } \mathit{Tree}_{\mathcal{X}}}{t : \mathit{Tree}_{\mathcal{X}}}$$

$$\frac{}{\{\} : \mathit{Tree}_{\mathcal{X}}} \quad \frac{X \in \mathcal{X}}{X : \mathit{Tree}_{\mathcal{X}}} \quad \frac{l : \mathit{Label} \quad Q : \mathit{Tree}_{\mathcal{X}}}{\{l \Rightarrow Q\} : \mathit{Tree}_{\mathcal{X}}}$$

$$\frac{l_1 : \mathit{Label} \quad l_2 : \mathit{Label}}{l_1 = l_2 : \mathit{Bool}} \quad \frac{l_1 : \mathit{Label} \quad \ldots \quad l_n : \mathit{Label} \quad p \text{ a variable}}{p(l1, \ldots, l_n) : \mathit{Bool}}$$

$$\frac{b : \mathit{Bool} \quad Q_1 : \mathit{Tree}_{\mathcal{X}} \quad Q_2 : \mathit{Tree}_{\mathcal{X}}}{\textbf{if } b \textbf{ then } Q_1 \textbf{ else } Q_2 : \mathit{Tree}_{\mathcal{X}}}$$

$$\frac{Q_1 : \mathit{Tree}_{\mathcal{Y}} \quad \ldots Q_m : \mathit{Tree}_{\mathcal{Y}}}{(X_1 := Q_1, \ldots, X_m := Q_m) : \mathit{Tree}_{\mathcal{Y}}^{\{\mathcal{X}_1, \ldots, \mathcal{X}_m\}}}$$

$$\frac{Q_1 : \mathit{Tree}_{\mathcal{X}} \quad Q_2 : \mathit{Tree}_{\mathcal{X}}}{Q_1 \cup Q_2 : \mathit{Tree}_{\mathcal{X}}} \quad \frac{Q_1 : \mathit{Tree}_{\mathcal{X}} \quad Q_2 : \mathit{Tree}_{\mathcal{Y}}^{\mathcal{X}}}{Q_1 \mathbin{@_{\mathcal{X}}} Q_2 : \mathit{Tree}_{\mathcal{Y}}}$$

$$\frac{y \text{ label variable} \quad t \text{ tree variable of type } \mathit{Tree}_{\mathcal{Y}} \quad Q_1 : \mathit{Tree}_{\mathcal{X}}^{\mathcal{X}} \quad Q_2 : \mathit{Tree}_{\mathcal{Y}}}{\mathit{gext}_{\mathcal{X}}(\lambda(y,t).Q_1)(Q_2) : \mathit{Tree}_{\mathcal{X}\cdot\mathcal{Y}}^{\mathcal{X}}}$$

Fig. 6. UnCAL Original Static Typing Rules (TR ver. of [2])

Note that *gext* is an old notation of structural recursion **rec**.

A.2 Proof of Lemma 1 (Refined Type Safety)

The proof of Lemma 1 is based on induction on the structure of UnCAL expression.

Proof. Base case:
Free variables: We assume that the type of free variables such as $\$db$ (input of the entire transformation) is available.
$\boldsymbol{\{\}}$: By the definition of $\mathcal{F}[\![\{\}]\!]$, $\mathsf{outMarker}(g) = \emptyset$. By the type inference rule, $\{\} :: DB_{\emptyset}$. Therefore, $\emptyset = \mathsf{outMarker}(g) \subseteq \mathcal{Y} = \emptyset$.
$\boldsymbol{\&y}$ **:** $\mathsf{outMarker}(\mathcal{F}[\![\&y]\!]) = \{\&y\}$ and $\&y :: DB_{\{\{\&y\}\}}$. $\&y :: DB_{\{\&y\}}$. Therefore, $\{\&y\} = \mathsf{outMarker}(g) \subseteq \mathcal{Y} = \{\&y\}$. Another nullrary constructor $\boldsymbol{()}$: is treated similarly.
Inductive case:
$\boldsymbol{\{l : e\}}$: Suppose $e :: DB_{\mathcal{Y}}$, $\mathcal{F}[\![e]\!] = g$, and $\mathcal{F}[\![\{l : e\}]\!] = g'$. Then $\mathsf{outMarker}(g') = \mathsf{outMarker}(g)$ by the definition of $\mathcal{F}[\![]\!]$ and $\{l : e\} :: DB_{\mathcal{Y}}$ by the type inference rule. Now suppose $\mathsf{outMarker}(g) \subseteq \mathcal{Y}$ as an induction hypothesis. Then we have $\mathsf{outMarker}(g) = \mathsf{outMarker}(g') \subseteq \mathcal{Y}$. $\boldsymbol{\&m := e}$ is treated similarly.
$\boldsymbol{e_1 \cup e_2}$: Suppose $e_1 :: DB_{\mathcal{Y}_1}^{\mathcal{X}}$, $e_2 :: DB_{\mathcal{Y}_2}^{\mathcal{X}}$, $\mathcal{F}[\![e_1]\!] = g_1$, $\mathcal{F}[\![e_2]\!] = g_2$, and

$\mathcal{F}[\![e_1 \cup e_2]\!] = g'$. Then $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_1) \cup \mathsf{outMarker}(g_2)$ by the definition of $\mathcal{F}[\![]\!]$ and $e_1 \cup e_2 :: DB_{\mathcal{Y}_1 \cup \mathcal{Y}_2}$ by the type inference rule. Now suppose $\mathsf{outMarker}(g_1) \subseteq \mathcal{Y}_1$ and $\mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_2$ as induction hypotheses. Then, by the property of the set union, we have $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_1) \cup \mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_1 \cup \mathcal{Y}_2$. $\oplus$ is treated similarly because type inference and evaluation rules for the output markers are identical to those of $\cup$.

$\boldsymbol{e_1}$ @ $\boldsymbol{e_2}$: Suppose $e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1}$, $e_2 :: DB^{\mathcal{Z}}_{\mathcal{Y}_2}$, $\mathcal{F}[\![e_1]\!] = g_1$, $\mathcal{F}[\![e_2]\!] = g_2$, and $\mathcal{F}[\![e_1 \mathbin{@} e_2]\!] = g'$. Then $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_2)$ by the definition of $\mathcal{F}[\![]\!]$ and $e_1 \mathbin{@} e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_2}$ by the type inference rule. Observe that (after connecting with matching input markers in g_2) the output markers in g_1 are ignored. Now suppose $\mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_2$ as an induction hypothesis. Then we have $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_2$.

cycle$\boldsymbol{(e)}$: Suppose $e :: DB^{\mathcal{X}}_{\mathcal{Y}}$, $\mathcal{F}[\![e]\!] = g$, and $\mathcal{F}[\![\mathbf{cycle}(e)]\!] = g'$. Then $\mathsf{outMarker}(g') = \mathsf{outMarker}(g) \setminus \mathsf{inMarker}(g)$ by the definition of $\mathcal{F}[\![]\!]$ and $\mathbf{cycle}(e) :: DB_{\mathcal{Y} \setminus \mathcal{X}}$ by the type inference rule. Now suppose $\mathsf{outMarker}(g) \subseteq \mathcal{Y}$ as an induction hypothesis. Then, since $\mathcal{X} = \mathsf{inMarker}(g)$ by the exactness of input marker inference, we have $\mathsf{outMarker}(g') = \mathsf{outMarker}(g) \setminus \mathsf{inMarker}(g) \subseteq \mathcal{Y} \setminus \mathcal{X}$.

if $\boldsymbol{b}$ **then** $\boldsymbol{e_1}$ **else** $\boldsymbol{e_2}$: Suppose $e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1}$, $e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_2}$, $\mathcal{F}[\![e_1]\!] = g_1$, $\mathcal{F}[\![e_2]\!] = g_2$, and $\mathcal{F}[\![\mathbf{if}\, b \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2]\!] = g'$. Then, depending on the value of b, $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_1)$ or $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_2)$ by the definition of $\mathcal{F}[\![]\!]$ and $\mathbf{if}\, b \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_1 \cup \mathcal{Y}_2}$ by the type inference rule. Now suppose $\mathsf{outMarker}(g_1) \subseteq \mathcal{Y}_1$ and $\mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_2$ as induction hypotheses and do case analysis for b. If $b = \mathsf{true}$, then $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_1)$, so $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_1) \subseteq \mathcal{Y}_1$. For the other case, $\mathsf{outMarker}(g') = \mathsf{outMarker}(g_2) \subseteq \mathcal{Y}_2$. For either case, by the property of the set union, we have $\mathsf{outMarker}(g') \subseteq \mathcal{Y}_1 \cup \mathcal{Y}_2$.

rec$\boldsymbol{(\lambda(\$l, \$t).e_{\mathrm{b}})(e_{\mathrm{a}})}$: Suppose $e_{\mathrm{a}} :: DB^{\mathcal{X}}_{\mathcal{Y}}$, $\mathcal{F}[\![e_{\mathrm{a}}]\!] = g$, $e_{\mathrm{b}} :: DB^{\mathcal{Z}_i}_{\mathcal{Z}_o}$, and $\mathcal{F}[\![\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(e_{\mathrm{a}})]\!] = g'$. Then, $\mathsf{outMarker}(g') = \{\&y.\&z \mid \&y \in \mathsf{outMarker}(g), \&z \in \mathcal{Z}\}$ by the definition of $\mathcal{F}[\![]\!]$ where $\mathcal{Z} = \mathcal{Z}_i \cup \mathcal{Z}_o$, and $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(e_{\mathrm{a}}) :: DB^{\mathcal{X}.\mathcal{Z}}_{\mathcal{Y}.\mathcal{Z}}$ by the type inference rule. Now suppose $\mathsf{outMarker}(g) \subseteq \mathcal{Y}$ as induction hypotheses. Then we have $\mathsf{outMarker}(g') = \{\&y.\&z \mid \&y \in \mathsf{outMarker}(g), \&z \in \mathcal{Z}\} \subseteq \{\&y.\&z \mid \&y \in \mathcal{Y}, \&z \in \mathcal{Z}\}$. Observe that $\mathcal{F}[\![]\!]$ does not use set of markers produced by e_{b} at run-time. Readers may wonder how the output markers are accessed via graph variable t, i.e., $\mathcal{Y}$ bound by **rec** affect the final result. Buneman et al. [4] does not explicitly mention, but it is natural to interpret as follows: Usually $\mathcal{Y}$ is disjoint from $\mathcal{Z}_i$ and therefore the output nodes marked by $\mathcal{Y}$ are not connected to $S1$ node [8]. Therefore we can safely ignore such $\mathcal{Y}$ in e_{b}.

Bound Variables : Variable $\$t$ is introduced by $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(e_{\mathrm{a}})$ and $\$t$ is bound to each of the subgraphs reachable from each edge. Similarly to [4], the type inference rule estimates the output markers as identical to that for e_{a}. So assuming type safety for e_{a}, type safety for $\$t$ immediately follows.

The above analysis covers all the expressions and thus conclude the proof. $\square$

[8] S1 node is a sort of Hub nodes, each of which corresponds to node produced by e_{a}

A.3 Proof of Refinement of Marker Analysis

This section gives a proof of the property:

$$^{\forall}\mathcal{X}^{\forall}\mathcal{Y}, ^{\exists}\mathcal{Y}' \supseteq \mathcal{Y}(e{::}DB^{\mathcal{X}}_{\mathcal{Y}} \Leftrightarrow e{:}DB^{\mathcal{X}}_{\mathcal{Y}'})$$

that appeared in Sect. 3. Note that we write $e{:}DB^{\mathcal{X}}_{\mathcal{Y}}$ to denote e has a type $DB^{\mathcal{X}}_{\mathcal{Y}}$ in the original type system.
(⇒)
Base case:
{} : According to the typing rules, $\{\} :: DB_{\emptyset}$ and $\{\} : DB_{\mathcal{Y}}$. Since $\emptyset \subseteq \mathcal{Y}$ for any $\mathcal{Y}$, the property holds.
() : Treated similarly to {}.
&y : According to the typing rules, $\&y :: DB_{\{\&y\}}$ and $\&y : DB_{\mathcal{Y}}$ for any $\mathcal{Y} \ni \&y$. Since $\{\&y\} \subseteq \mathcal{Y}$, the property holds.
Inductive case:
{l : e}: Suppose $e{::}DB_{\mathcal{Y}}$, $e{:}DB_{\mathcal{Y}'}$, and $\{\&y\} \subseteq \mathcal{Y}$ as the induction hypothesis. Then, according to the typing rule, $\{l{:}e\}{::}DB_{\mathcal{Y}}$, and $\{l{:}e\}{:}DB_{\mathcal{Y}'}$. So, the property holds.
&m := e is treated similarly.
cycle(e): Suppose $e{::}DB^{\mathcal{X}}_{\mathcal{X}\cup\mathcal{Y}}$, $e{:}DB^{\mathcal{X}}_{\mathcal{X}\cup\mathcal{Y}'}$, and $\{\&y\} \subseteq \mathcal{Y}$ as the induction hypothesis. We also assume $\mathcal{X}$ to be disjoint with $\mathcal{Y}$ and $\mathcal{Y}'$. Then, according to the typing rule, $\mathbf{cycle}(e){::}DB^{\mathcal{X}}_{\mathcal{Y}}$ and $\mathbf{cycle}(e){:}DB^{\mathcal{X}}_{\mathcal{Y}'}$. So, the property holds.
$\boldsymbol{e_1 \cup e_2}$: Suppose $e_1 :: DB^{\mathcal{X}}_{\mathcal{Y}_1}$ and $e_2 :: DB^{\mathcal{X}}_{\mathcal{Y}_2}$, and suppose $e_1 : DB^{\mathcal{X}}_{\mathcal{Y}'_1}$ and $e_2 : DB^{\mathcal{X}}_{\mathcal{Y}'_2}$ where $\mathcal{Y}_1 \subseteq \mathcal{Y}'_1$ and $\mathcal{Y}_2 \subseteq \mathcal{Y}'_2$ as induction hypotheses. Then, according to the typing rule, $(e_1 \cup e_2) :: DB^{\mathcal{X}}_{\mathcal{Y}_1\cup\mathcal{Y}_2}$ and $(e_1 \cup e_2) : DB^{\mathcal{X}}_{\mathcal{Y}'}$ where $\mathcal{Y}'_1 \subseteq \mathcal{Y}'$ and $\mathcal{Y}'_2 \subseteq \mathcal{Y}'$. So $\mathcal{Y}_1 \cup \mathcal{Y}_2 \subseteq \mathcal{Y}'$. Therefore, the property holds.
$\boldsymbol{e_1 \oplus e_2}$ and **if *b* then $\boldsymbol{e_1}$ else $\boldsymbol{e_2}$** are treated similarly.
$\boldsymbol{e_1}$ @ $\boldsymbol{e_2}$: For $e_1{::}DB^{\mathcal{X}}_{\mathcal{Y}}$ and $e_2{::}DB^{\mathcal{Y}'}_{\mathcal{Z}}$, suppose $e_1{:}DB^{\mathcal{X}}_{\mathcal{Y}_2}$ and $e_2{:}DB^{\mathcal{Y}'}_{\mathcal{Z}'}$ s.t. $\mathcal{Y} \subseteq \mathcal{Y}_2$ and $\mathcal{Z} \subseteq \mathcal{Z}'$ as induction hypotheses. Note that we assume $\mathcal{Y} \subseteq \mathcal{Y}'$ for compatibility with the original type system, which require set of output markers of first operand and the input markers of the second operand to coincide, which means the former should be a subset of the latter,[9] i.e., $\mathcal{Y}_2 \subseteq \mathcal{Y}'$. This means we cannot assume $\mathcal{Y}_2$ to be arbitrary larger than $\mathcal{Y}$, but only a set that is smaller or equal to $\mathcal{Y}'$. Then, according to the typing rule, e_1 @ $e_2{::}DB^{\mathcal{X}}_{\mathcal{Z}}$ and e_2 @ $e_2{:}DB^{\mathcal{X}}_{\mathcal{Z}'}$. Since $\mathcal{Z} \subseteq \mathcal{Z}'$, the property holds.
$\mathbf{rec}(\boldsymbol{\lambda(\$l, \$t).e_{\mathrm{b}}})(\boldsymbol{e_{\mathrm{a}}})$: Assume $e_{\mathrm{a}} :: DB^{\mathcal{X}}_{\mathcal{Y}}$ and $e_{\mathrm{a}'} : DB^{\mathcal{X}}_{\mathcal{Y}'}$ where $\mathcal{Y} \subseteq \mathcal{Y}'$. Special care is needed for e_{b}: There is no use to have an output marker that is not included in the set of input marker in e_{b}, since such excess output node is not connected to any other node. It is best explained by the rule of **rec**: $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(\{l : d\}) = e_{\mathrm{b}}[{}^{l}\!/\!_{\$l}][{}^{d}\!/\!_{\$t}]$ @ $\mathbf{rec}(\lambda(\$l, \$t).e_{\mathrm{b}})(d)$. The set of cor-

[9] Suppose the set of markers common to the two positions (output of the first operand and input of the second operand) which the original type system assigns is $\mathcal{Y}''$. Then $\mathcal{Y}'_2 \subseteq \mathcal{Y}''$ and $\mathcal{Y}' = \mathcal{Y}''$ should be satisfied, because the set of input marker does not change by subtyping rule. Therefore, $\mathcal{Y}_2 \subseteq \mathcal{Y}'$ follows.

responding input markers is the set of input markers of e_{b} itself. So we assume equal set of input and output markers for e_{b} thus $e_{\mathrm{b}} :: DB^{\mathcal{Z}}_{\mathcal{Z}}$. Then, we have $\mathbf{rec}(\lambda(\$l,\$g).e_{\mathrm{b}})(e_{\mathrm{a}}) :: DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}}$ and $\mathbf{rec}(\lambda(\$l,\$g).e_{\mathrm{b}})(e_{\mathrm{a'}}) : DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y'}\cdot\mathcal{Z}}$. Since $\mathcal{Y}\cdot\mathcal{Z} \subseteq \mathcal{Y}'\cdot\mathcal{Z}$, the property holds.

The above case analysis covers all the cases by the induction and thus concludes the proof. The opposite direction ($\Leftarrow$) can be proved similarly. $\square$

A.4 Concrete Rewriting Examples

This section shows input and output of optimizations used in *Ex1* transformation appeared in Sect. 5. For input transformation **Q1**, our system produces **Q2** by applying first fusion rule. Previously the translation from **Q2** to **Q3** was not automatic, but algorithm in Sect. 4 enables deriving **Q3** automatically.

Q3 can be obtained by the plugging based rewriting rules. For example,

```
(&z1 := (&z1 := {"name": &z2}, &z2 := {"name": &z2}))
  @ (&z2 := &z1&z2, &z1 := &z1&z1)
```

becomes

```
&z1 := (&z1 := {"name": &z1&z2}, &z2 := {"name": &z1&z2}).
```

This pattern frequently appears after **rec** fusion because **rec** often appears in the pattern $\&z$ @ $\mathbf{rec}(\_)(\_)$ because from the UnQL surface syntax, only one component of structural recursion is necessary and the idiom $\&z$ @ _ implements this projection.

Q 1.

```
&z1@rec(\ ($L,$T).
  if $L = "name"
  then (&z1 := {"name": &z2},
        &z2 := {"name": &z2})
  else (&z1 := &z1, &z2 := {$L: &z2}))
  (&z1@rec(\ ($L,$T).
    if $L = "name"
    then (&z1 := {"name": &z1},
          &z2 := {"typeName": &z2})
    else if $L = "primitiveDataType"
      then (&z1 := {"primitiveDataType": &z2},
            &z2 := {"primitiveDataType": &z2})
      else (&z1 := {$L: &z1}, &z2 := {$L: &z2}))
      ($db))
```

Q 2.

```
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ ($Sa1,$T).
  if $Sa1="name"
```

```
  then (&z1 := (&z1 := {"name": &z2},
                &z2 := {"name": &z2})
     @ (&z2 := &z1&z2, &z1 := &z1&z1),
        &z2 := (&z1 := &z1,
                &z2 := {"typeName": &z2})
     @ (&z2 := &z2&z2, &z1 := &z2&z1))
  else if $Sa1 = "primitiveDataType"
    then (&z1 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1),
          &z2 := (&z1 := &z1,
                  &z2 := {"primitiveDataType": &z2})
       @ (&z2 := &z2&z2, &z1 := &z2&z1))
    else (&z1 := if $Sa1 = "name"
     then (&z1 := {"name": &z2},
           &z2 := {"name": &z2})
     else (&z1 := &z1, &z2 := {$Sa1: &z2})
           @ (&z2 := &z1&z2, &z1 := &z1&z1),
              &z2 := if $Sa1 = "name"
       then (&z1 := {"name": &z2},
             &z2 := {"name": &z2})
       else (&z1 := &z1, &z2 := {$Sa1: &z2})
          @ (&z2 := &z2&z2, &z1 := &z2&z1)))($db)
```

Q 3.

```
&z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
 rec(\ ($Sa1,$T).
  if $Sa1="name"
  then (&z1&z1 := {"name": &z1&z2},
        &z1&z2 := {"name": &z1&z2},
        &z2&z1 := &z2&z1,
        &z2&z2 := {"typeName": &z2&z2})
  else if $Sa1 = "primitiveDataType"
    then (&z1&z1 := &z2&z1,
          &z1&z2 := {"primitiveDataType": &z2&z2},
          &z2&z1 :=  &z2&z1,
          &z2&z2 := {"primitiveDataType": &z2&z2}
    else (&z1 := if $Sa1 = "name"
             then (&z1 := {"name": &z1&z2},
                   &z2 := {"name": &z1&z2})
             else (&z1 := &z1&z1,
                   &z2 := {$Sa1: &z1&z2}),
          &z2 := if $Sa1 = "name"
            then (&z1 := {"name": &z2&z2},
                  &z2 := {"name": &z2&z2})
            else (&z1 := &z2&z1,
                  &z2 := {$Sa1: &z2&z2})
           ))($db)
```

A.5 Proof of Remark 1 (Removal of Intermediate Graph)

The remark gives necessary condition of the removal of intermediate graphs via fusion rule applications.

In the following, the context C with a hole $\Box$ can be defined as

$$\begin{aligned} C ::= & \Box \mid \{a : C\} \mid C \cup e \mid e \cup C \mid C \oplus e \mid e \oplus C \mid \&x := C \\ \mid & \ C \mathbin{@} e \mid e \mathbin{@} C \mid \mathbf{cycle}(C) \end{aligned}$$

where e denotes UnCAL expressions which consist only of constructors, and $C[e']$ denotes an expression that is made by replacing the hole in C by UnCAL expression e'.

case (i): $\$t_2$ does not appear free in e_2.

The context $C[]$ in the expression

$$\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(C[\mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(e_0)])$$

can be removed by the prior auxiliary rewriting rules summarized in Section 2.7 to form the following direct composition.

$$\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(\mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(e_0))$$

The first fusion rule will turn it into

$$\mathbf{rec}(\lambda(\$l_1, \$t_1).\, \mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(e_1))(e_0).$$

If e_1 contains another **rec**, i.e.,

$$\mathbf{rec}(\lambda(\$l_1, \$t_1).\, \mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(C'[\mathbf{rec}(\lambda(\$l_0, \$t_0).e')(\ldots)]))(e_0),$$

then, because $\$t_2$ still does not appear free in e_2, this composition will also be turned into nesting

$$\mathbf{rec}(\lambda(\$l_1, \$t_1).\, \mathbf{rec}(\lambda(\$l_0, \$t_0).\, \mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(e')(\ldots)))(e_0)$$

using the prior rewriting rule of **rec** (to remove the context C') and the first fusion rule. In this way, the body of the downstream **rec** of the composition newly introduced by the fusion comes from the downstream **rec** before fusion. So the condition to apply the first fusion rule is maintained. These process of fusion are repeated until all the generated compositions become nested. Therefore, all the intermediate results (compositions) are removed.

case (ii): $\$t_2$ appears free in e_2.

Suppose the context $C[]$ in $\mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(C[\mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(e_0)])$ does not contain @ or **cycle**. Then, auxiliary rules of **rec** can remove the context and turn the expression into direct composition. Suppose our proposed marker analysis detects that e_1 does not produce output markers. Then, the first fusion rule becomes applicable. At this point, possible composition that emerge as a result of fusion will be similar to the prior case, thus

$$\mathbf{rec}(\lambda(\$l_1, \$t_1).\, \mathbf{rec}(\lambda(\$l_2, \$t_2).e_2)(C'[\mathbf{rec}(\lambda(\$l_0, \$t_0).e')(\ldots)]))(e_0).$$

If we can further determine that e' does not produce output markers, then the first fusion rule will be applied again. Therefore, the composition of **rec** whose body of the upstream only contains nesting of **rec** via contexts that do not include @ or **cycle**, can be completely removed. $\square$

Let us examine, within case (ii) the subcase where e_1 has output markers. If $C[]$ does not contain @ or **cycle**, then $C[]$ is removed to leave direct composition of **rec**. The second fusion rule will produce, at the argument position of the inner rec, the expression of the form $e_1 \mathbin{@} \mathbf{rec}(\lambda(\$l_1, \$t_1).e_1)(\$t_1)$, which will produce new intermediate results. @ in the expression cannot be removed just by using property of @. Furthermore, if we try to remove @ using auxiliary rules of **rec**, it is impossible because $\$t_2$ appears free in e_2. Therefore, we cannot reduce the indirect composition to direct composition, and therefore, the composition cannot be removed completely.