Updatable Graph Views over a Graph Database

Luca Pardini
pardini@di.unipi.it

January 15, 2010
Abstract

The problem of propagating updates over views on the corresponding database is a long standing data management problem. Many researchers tackled this problem in the context of relational databases and relational views; more recently, the problem has been further studied in the XML context. Till now, this problem has received little attention in the case of graph views over a generic graph database.

In this document we propose an approach to this problem using bidirectional transformations between the graph views and the original graph database.
Chapter 1

Introduction

The ability to represent and query data with little or no apparent structure arises in several areas: biological databases, database integration, and query systems for the World-Wide Web [34, 39, 12, 36, 28]. The general approach is to represent data as a labeled graph.

Graph databases [1] are born to handle these kind of environments. The model of a graph database is a model where the data structures for the schema and instances are modelled as a labeled directed graph, or generalisation of the graph data structure, where data manipulation is expressed by graph transformations.

Like relational databases, an user can generate personalised views over graph databases by adding (or removing) edges (or nodes) or more complicated functions. These views can be queried or updated using specific languages; unfortunately the problem to put back the update from a view into the original database is not well studied over generic graphs.

In this document we discuss and propose a possible way to resolve this problem, called the view/update problem, over generic graph databases using graph views.

1.1 Organization

This thesis proposal is organised as follow:

- in Chapter 2 we describe briefly the state of the art about the view/update problem;
- in Chapter 3 we describe the bidirectional approach to make transformations over views with a tree structure;
- in Chapter 4 we describe our proposal and the work to be done during the thesis.
Chapter 2

State of the Art

In this chapter we briefly recall the main concepts about views, introduce the view/update problem, and discuss the existing works about relational databases and relational views; then, we illustrate the evolution of the problem in the context of XML databases and views.

2.1 Views

One of the most important features of database management systems is the ability to define views. A view is a “virtual” database defined by a function ranging on the existing database and/or previously defined views. For instance, in the relational setting a view is a table or a set of tables consisting of a fragment of the original database and/or some aggregated values.

The reason behind the introduction of views are the following:

- views are a mechanism for enforcing security and protection policies: indeed, views can be used to restrict the access to a selected fragment of the data;
- views are massively used during query optimization: indeed, the optimizer can materialize the result of frequently used queries (or subqueries) and exploit this materialized information to speed up query processing;
- views grant logic independence.

Views are defined by means of algebraic operators; in the relational context, for instance, view definitions usually contain attribute renaming operations, the creation of computed attributes, and other standard relational operators.

A view can be virtual or materialized; a virtual view is computed every time the view is referenced, while a materialized view is stored to allow for quick data accesses.

A view can be queried or updated. In both cases these operations must be translated into the equivalent operations on the original database. Queries can be composed with view definitions or answered by accessing their materialization, hence queries can be managed in a relatively easy way.
On the contrary, the propagation of updates from a view to the underlying database is a well-known and challenging problem. We will discuss the various aspects of this problem in the next Section.

2.2 The View/Update Problem

The view/update problem was first introduced by Codd in his seminal work about the relational data model \cite{14}; the problem was further studied by many others included, for example, Paolini and Pelagatti \cite{33}, and formalized by Dayal and Bernstein \cite{16}.

The view/update problem can be defined as follows.

**Definition 2.1** (The View/Update Problem). Given a semantically consistent database $R$ and a function $f$ that creates a view $V = f(R)$ and given an update $u$ (insertion, deletion or update) over the view, the problem is to compute a minimal set of updates $w$, that preserve semantic consistency, on the database such that $u(V) = f(w(R))$ (Figure 2.1).

![Figure 2.1: The View/Update Problem](image)

In \cite{16} Dayal and Bernstein showed that propagating an update is not as easy as answering a query over a view: indeed, it may happen that no translation exists for a given update or that this translation is not unique (i.e, the update might be propagated in several different ways).

To deal with these issues, Dayal and Bernstein introduced several criteria to evaluate the correctness of a set of updates $u$ over the database; these criteria are the following:

- **no extraneous updates** in $u$: an update $g \in u$ is extraneous in $u$ if $f(g(R)) = f(R)$ or, in other words, if the update has no effect over the view;

- **no side effects** on $V$: a set of updates $u$ has no side effect if only the desired update is performed on the view;

- **$u$ preserves semantic consistency**: a set of updates $u$ preserves semantic consistency if $u(R)$ is also semantic consistent;
• **$u$ is unique**: there not exists a set of updates $w$, different from $u$, such that $f(u(R)) = f(w(R))$, in other words $u$ must be the only set of updates that produce the desired update over the view.

These criteria form the basis for all the works about the view/update problem.

As a final remark, the view/update must not be confused with the view maintenance problem, where an update on the underlying database must be propagated to a materialized view ([22] [4]).

## 2.3 Solutions on relational databases

In this section we present the works where the authors consider a relational view over a relational database.

Furtado et.al [20] show that semantic information should be supplied in order to resolve the anomalies which may arise in updating using views, and deduce a starting set of view update translation rules. The need of semantic information also emerge in [16] [13].

### Functional dependencies

Dayal and Bernstein [16, 17] use functional dependencies to propagate updates on the view to the database; Carlson and Arora [13] show that functional dependencies do not give enough information in all cases and propose two translation procedures to preserve the interpretation of insert and delete updates.

### Complementary and Consistent view

A different approach based on semantic information, not related with the works by Bernstein and Arora, is proposed by Bancilhon and Spyritos [37, 2]. In these works they introduce the concept of **complementary view**. A complementary view is a view with the property of containing all the information needed to the view to rebuild the whole database. They show that, for each view, in general, more than a complementary view exists; however, once chosen a specific complementary view, an update on the view can be translate uniquely (or cannot be translated at all). Unfortunately, Cosmadakis and Papadimitriou [15] show that finding a minimum complement of a given view is NP-complete.

Paolini et al. [21] extend the work of Bancilhon and Spyritos introducing the concept of **consistent view**. To define a consistent view they define both the view and the database as a **data abstraction**, i.e., a set of states and a set of operations. A view $V$ is a consistent view if:

- $V$ **preserves connections** on $R$: by a given state of the database $R$ it is not possible to reach two different states of the view $V$;

- $V$ **preserves loops** on $R$: if an update leaves unchanged the state of the view $V$, then the update leaves unchanged the state of the database $R$. 
Paolini et al. prove that complementary views are a subclass of consistent views by showing some view of highly applicative importance which is consistent but cannot be modeled by the approach of Bancilhon and Spyros. Finally they show that the update of consistent views can be determined by imposing a partial order on the values of the view complement.

Other semantic approaches

The semantic approach was also used by Masunaga in [31], where the semantics of views is defined using a first order predicate calculus and the view updatability is examined from the semantic point of view. Masumaga presented five local translation rules:

- the direct product view;
- the union view;
- the difference view;
- the projection view;
- the restriction view.

All other views can be built using these five operators and the translations are derived by the composition of the local translations.

As shown by Masumaga, this approach has several ambiguities: solving these ambiguities is still an open problem.

Recent work

In [29] Kotidis et al. propose a solution to handle all possible updates through relational views without side-effects. The main idea of their solution is to divide the problem into two different layers:

- logical layer;
- physical layer.

In the logical layer all tables are normal tables with their attribute.

In the physical layer, they add an identifier to each attribute of the join of each table referred by the view to both physical tables; for each row of the view, they create a duplicate with different identifiers inside the physical tables.

With this approach they can delete (or insert or update) a single row of the view by removing the duplicated rows inside the physical layer with the same identifier, hence achieving the absence of side-effects.

Example 2.2. Consider the table in Figure 2.2. Personnel, Teaching, and Schedule are our tables, while \( V_t \) is a view that represents the join of our three tables at the logical layer. We want to remove by the view the gray row \( (t_d) \), but if we remove the single row \( t_p, t_t \) or \( t_s \) by the three original table we have, as a side effect, the disappearance of other rows.
Consider now the physical layer shown in Figure 2.3; if the three table are stored with an identifier for each attribute of the join, we can remove, for each table, the gray row and the row $t_d$ disappears from the view.

Kotidis et al. implement a system with these characteristics and show an experimental analysis of their work.

## 2.4 Non-Relational views

In this section we present the most important works about non-relational views over a relational database.

### Object views

In \cite{Keller3} Keller et al. study the view/update problem for object views defined over relational databases. They propose an approach that, given a view update, enumerates all valid translations into sequences of database update operations. These translation are ordered according to five validity criteria that must be satisfied to resolve ambiguity. This enumeration approach is inherited from a previous work on relational views (see \cite{Keller27}).
Figure 2.3: Physical Layer

XML views

In Tatarinov et al. consider, for the first time, the problem of updating an XML view over relational data. In this work they present a primitive algorithm to handle that problem in simple cases; these cases do not cover complicated scenarios.

Moving from this early work, Wang et al. formalize the view/update problem for XML views and try to understand when an update can be translated; their approach leverages on a graph-based algorithm that decides whether an update can be translated or not. However, they do not deal with the problem of the actual update translation.

Independently, Braganholo, Davidson, and Heuser try to apply the same approach proposed by Bernstein to understand how an update over an XML view can be pushed into the underlying relational database. Their approach supports the insertion of a subtree at a given node, the deletion of the subtree rooted at a given node, or the modification of a node. They introduce the concept of nest-last XML views using a nested relational algebra expression and reduce the problem of updating a view of this class to that of updating a relational view by translating, if the update is valid, an operation into a sequence of SQL updates. They also build a system that implements their work.

2.5 Bidirectional transformations

In this section we describe a new way of solving the view/update problem. This approach is based on bidirectional transformations, i.e., transformations that specify how to compute the view from the database and how to propagate view updates to the database. By using this approach, hence, a view is defined by a bidirectional transformation.
Lenses

The bidirectional transformation approach has been studied by Pierce et al. in [5, 19, 18, 35, 6]. They build a set of combinators equipped with two different functions: the get function translates data from database to the view, while the put function translates back data from the view to the database. View are expressed using these combinators and one can build the view by using the get function and update the database using the put one. We present these combinators in Chapter 3.

In their work Pierce et al. build their combinators for many instance of the view/update problem: the case where both the database and the view are a tree; the canonical case where both the database and the view are relational; and, finally, the case where the view and the database are strings.

Other works

In [32, 25, 26] Takeichi et al. present a bidirectional transformation language, based on get and put functions, witch is able to deal with duplication and structural changes at the cost of introducing editing tags. They also build an editor to develop transformations based on their language.

Later, in [30] Takeichi et al. propose an approach to the view/update problem for XML database and XQuery views. They define a language with a dual semantic: the forward semantic and the backward semantic. Basically, for each construct of the language, the forward semantic is the get function and the backward semantic is the put one. Finally they translate every construct of the core of the XQuery language into their language and implement this approach.

In [40] Voigtländer describes a function bbf that automatically computes the put function starting from a given a get function. The author describes and implements bbf in Haskell and shows some variant of the function.

The last works we describe are related to graphs: in [45] again Takeichi et al. approach the problem over graphs by representing graphs using trees, than use the previous results over trees to make transformations over graphs. In [23, 24] Hu et al. propose a set of rules, with a bidirectional semantic, to transform rooted edge-labeled directed graphs but they left as a challenge to provide a general bidirectional framework for graphs.
Chapter 3

Combinators for Bidirectional Transformations

In this chapter we will briefly illustrate a novel approach to view update based on bidirectional transformations [19, 18]. We will start by introducing the concept of lenses; later we will present the combinators defined for transformations over trees.

3.1 Lenses

Let’s start with the formal definition of lenses, where we assume $V$ as the set of all views.

**Definition 3.1** (Lenses). A lens $l$ comprises a partial function $l\uparrow$ from $V$ to $V$, called the *get* function of $l$, and a partial function $l\downarrow$ from $V \times V$ to $V$, called the *putback* function of $l$.

Informally the *get* function of a lens lifts an abstract view out of a concrete one, while the *putback* function pushes down a new abstract view into an existing concrete view.

The following property is the materialization of the bidirectional approach of lenses.

**Definition 3.2** (Well-behaved lenses). Let $l$ be a lens and let $C$ and $A$ be subset of $V$. We say that $l$ is a well-behaved lens from $C$ to $A$, written $l \in C \rightleftharpoons A$, iff it maps argument in $C$ to results in $A$ and vice versa:

$$
l\uparrow(C) \subseteq A \quad \text{(Get)}
$$

$$
l\downarrow(A \times C) \subseteq C \quad \text{(Put)}
$$

and its *get* and *putback* functions obey the following laws:

$$
l\downarrow(l\uparrow(c), c) \subseteq c \quad \text{for all } c \in C \quad \text{(GetPut)}
$$

$$
l\uparrow(l\downarrow(a, c)) \subseteq a \quad \text{for all } (a, c) \in A \times C \quad \text{(PutGet)}
$$

where $f(x) \subseteq y$ for $f(x) = \bot$ or $f(x) = y$. 

9
The GetPut law states that, if we get an abstract view \( a \) from a concrete view \( c \) and we putback \( a \) into \( c \) without modify \( a \), we must get back exactly \( c \). PutGet mean that the putback function must capture all the information contained in the abstract view.

**Definition 3.3** (Very well-behaved lenses). Let \( l \in C \Rightarrow A \). We say that \( l \) is a very well-behaved lens from \( C \) to \( A \) if its get and putback functions obey the following law:

\[
l \upcirc (a', l \downcirc (a, c)) \subseteq l \downcirc (a', c) \quad \text{for all } a, a' \in A \text{ and } c \in C \quad \text{(PutPut)}
\]

The PutPut law states that the effect of a sequence of two putbacks is just the effect of the second: the first gets completely overwritten.

**Definition 3.4** (Totality). Let \( l \in C \Rightarrow A \). We say that \( l \) is total, written \( l \in C \iff A \), if \( C \subseteq \text{dom}(l \upcirc) \) and \( A \times C \subseteq \text{dom}(l \downcirc) \) where \( \text{dom}(f) = \{ x \mid f(x) \downarrow \} \).

**Other properties**

We now explore some simple properties of lenses.

**Definition 3.5.** Let \( f \) be a partial function from \( A \times C \) to \( C \) and \( P \subseteq A \times C \). We say that \( f \) is injective on \( P \) if, for all views \( a, a', c \) and \( c' \) with \( (a, c) \in P \) and \( (a', c') \in P \), if \( f(a, c) \downarrow \) and \( f(a', c') \downarrow \), then \( a \neq a' \) implies \( f(a, c) \neq f(a', c') \).

**Lemma 3.6.** If \( l \in C \Rightarrow A \), then \( l \upcirc \) is injective on \( \{(a, c) \mid (a, c) \in A \times C \land l \downcirc (l \upcirc (a, c)) \downarrow \} \).

**Lemma 3.7.** If \( l \in C \iff A \), then \( l \upcirc \) is injective on \( A \times C \).

The injective property is useful to show when a lens is not well-behaved and enhances the design of correct combinators.

An important special case arises when the putback function of a lens is completely insensitive to its concrete argument.

**Definition 3.8** (Oblivious lenses). A lens \( l \) is said to be oblivious if \( l \upcirc (a, c) = l \upcirc (a, c') \) for all \( a, c, c' \in V \).

Oblivious lenses have these properties:

**Lemma 3.9.** If \( l \) is oblivious and \( l \in C_1 \Rightarrow A_1 \) and \( l \in C_2 \Rightarrow A_2 \), then \( l \in C_1 \cup C_2 \Rightarrow A_1 \cup A_2 \).

**Lemma 3.10.** If \( l \) is oblivious and \( l \in C \iff A \), then \( l \downcirc \) is a bijection from \( C \) to \( A \).

Now we introduce an information ordering on lenses and show that the set of lenses equipped with this ordering is a complete partial order with bottom, where the bottom is the lens whose get and putback functions are everywhere undefined. The definitions and theorem are the following.

**Definition 3.11.** We say that a lens \( s \) is more informative than a lens \( l \), written \( l \prec s \), iff \( \text{dom}(l \upcirc) \subseteq \text{dom}(s \upcirc) \), \( \text{dom}(l \downcirc) \subseteq \text{dom}(s \downcirc) \), \( l \downcirc (c) = s \downcirc (c) \) for all \( c \in \text{dom}(l \downcirc) \), and \( l \upcirc (a, c) = s \upcirc (a, c) \) for all \( (a, c) \in \text{dom}(l \upcirc) \).
Lemma 3.12. ≺ is a partial order on lenses.

Theorem 3.13. Let $L$ be the set of well-behaved lenses from $C$ to $A$, then $(L, ≺)$ is a complete partial order with bottom.

In order to deal with cases where a putback function is called where no old concrete view is available, we enrich the universe $V$ of views with a special element $Ω$. We write $C ⊩ A$ for the set of well behaved lenses from $C ∪ Ω$ to $A ∪ Ω$ and $C ⇐≡ A$ for the set of total lenses that obey to these conventions:

- $l(Ω) ⊆ Ω$;
- $l(Ω, c) ⊆$ for any $c$;
- $l(c) ≠ Ω$ for any $c ≠ Ω$;
- $l(a, c) ≠ Ω$ for any $a ≠ Ω$ and any $c$.

Lemma 3.14. For any lens $l$ and set of view $C$ and $A$ we have that:

- $l ∈ C ⊩ A$ implies $l ∈ C ⇐≡ A$;
- $l ∈ C ⇐≡ A$ implies $l ∈ C ⊩ A$.

3.2 Generic combinators

In this section we show several generic lens combinators, whose definitions are independent of the particular choice of the universe $V$. Each definition is accompanied by a type declaration asserting its well behavior under certain conditions.

Identity

The simplest lens is the identity. It copies the concrete view in the get direction and the abstract view in the putback direction.

$$
\begin{align*}
\text{id}^\to(c) &= c \\
\text{id} \downarrow(a, c) &= a
\end{align*}
\forall C ⊆ V. \text{id} ∈ C ⊩ C
$$

On the identity lens the following propriety holds.

Lemma 3.15 (Well-behavior). For each $C ⊆ V$ we have that $\text{id} ∈ C ⊩ C$.

Lemma 3.16 (Totality). For each $C ⊆ V$ we have that $\text{id} ∈ C ⇐≡ C$. 
CHAPTER 3. COMBINATORS FOR BIDIRECTIONAL TRANSFORMATIONS

Composition

The lens composition combinator \( l; k \) places \( l \) and \( k \) in sequence.

\[
\begin{align*}
(l; k)(c) &= \quad k'(l'(c)) \\
(l; k)(a, c) &= \quad l'(k'(a, l'(c)), c)
\end{align*}
\]

\( \forall A, B, C \subseteq V. \forall l \in C \overset{\Omega}{\leftrightarrow} B. \forall k \in B \overset{\Omega}{\leftrightarrow} A. \; l; k \in C \overset{\Omega}{\leftrightarrow} A \)

On the composition lens the following propriety holds.

**Lemma 3.17** (Well-behavior). For each \( A, B, C \subseteq V \), for each \( l \in C \overset{\Omega}{\leftrightarrow} B \) and for each \( k \in B \overset{\Omega}{\leftrightarrow} A \) we have that \( l; k \in C \overset{\Omega}{\leftrightarrow} A \).

**Lemma 3.18** (Totality). For each \( A, B, C \subseteq V \), for each \( l \in C \overset{\Omega}{\leftrightarrow} B \) and for each \( k \in B \overset{\Omega}{\leftrightarrow} A \) we have that \( l; k \in C \overset{\Omega}{\leftrightarrow} A \).

Constant

The constant lens transforms any view into the constant view \( v \) in the get direction. In the putback direction, this lens simply restores the old concrete view, if is available, or returns a default view \( d \).

\[
\begin{align*}
(c \text{const } v d)(c) &= \quad v \\
(c \text{const } v d)(a, c) &= \quad c \text{ if } c \neq \Omega \\
&\quad d \text{ if } c = \Omega
\end{align*}
\]

\( \forall C \subseteq V. \forall v \in V. \forall d \in C. \; (c \text{const } v d) \in C \overset{\Omega}{\leftrightarrow} \{ h \} \)

On the composition lens the following propriety holds.

**Lemma 3.19** (Well-behavior). For each \( C \subseteq V \), for each \( v \in V \) and for each \( d \in C \) we have that \( (c \text{const } v d) \in C \overset{\Omega}{\leftrightarrow} \{ h \} \).

**Lemma 3.20** (Totality). For each \( C \subseteq V \), for each \( v \in V \) and for each \( d \in C \) we have that \( (c \text{const } v d) \in C \overset{\Omega}{\leftrightarrow} \{ h \} \).

Conditional

We have two different kinds of conditional combinators. The first combinator is parameterized on a predicate \( C_1 \) on views and two lenses \( l_1 \) and \( l_2 \). In the get direction it tests the concrete view \( c \) and applies the get of \( l_1 \) if \( c \) satisfies the predicate or applies the get of \( l_2 \) if \( c \) does not satisfy it. In the putback direction \( c \text{cond} \) examines the concrete view and applies the putback of \( l_1 \) or the putback of \( l_2 \).

\[
\begin{align*}
(c \text{cond } C_1 \; l_1 \; l_2)(c) &= \quad l'_1(c) \text{ if } c \in C_1 \\
&\quad l'_2(c) \text{ if } c \notin C_1 \\
(c \text{cond } C_1 \; l_1 \; l_2)(a, c) &= \quad l''_1(a, c) \text{ if } c \in C_1 \\
&\quad l''_2(a, c) \text{ if } c \notin C_1
\end{align*}
\]

\( \forall C, C_1, A \in V. \forall l_1 \in C \cap C_1 \overset{\Omega}{\leftrightarrow} A. \forall l_2 \in C \cap C_1 \overset{\Omega}{\leftrightarrow} A \)

\( (c \text{cond } C_1 \; l_1 \; l_2) \in C \overset{\Omega}{\leftrightarrow} A \)

CHAPTER 3. COMBINATORS FOR BIDIRECTIONAL TRANSFORMATIONS

In the second kind of conditional combinator, the get direction is equal to ccond but in the putback direction acond examines the abstract view a and applies the putback function of l₁ only if a satisfies the predicate A₁, otherwise it applies l₂.

\[
(acond \ C \ A \ l_1 \ l_2)(c) = \begin{cases} 
  l'_1(c) & \text{if } c \in C_1 \\
  l'_2(c) & \text{if } c \notin C_1 
\end{cases}
\]

\[
(acond \ C \ A \ l_1 \ l_2)(a, c) = \begin{cases} 
  l'_1(a, c) & \text{if } a \in A_1 \land c \in C_1 \\
  l'_1(a, \Omega) & \text{if } a \in A_1 \land c \notin C_1 \\
  l'_2(a, c) & \text{if } a \notin A_1 \land c \in C_1 \\
  l'_2(a, \Omega) & \text{if } a \notin A_1 \land c \notin C_1 
\end{cases}
\]

∀C, C₁, A, A₁ ∈ V. ∀l₁ ∈ C ∩ C₁ ⇐⇒ A ∩ A₁. ∀l₂ ∈ C \ C₁ ⇐⇒ A \ A₁

(acond \ C \ A \ l_1 \ l_2) ∈ C ⇐⇒ A

3.3 Combinators for trees

In this section we present a large list of combinators written to work on a set T of finite, unordered edge-labeled trees with labels drawn from a finite set N of names, and with the children of a given node all labeled with distinct names. For each combinator we present the rule and we describe briefly what the combinator does.

**Hoisting**

This combinator is used to shorten a tree by removing an edge at the top. In the get direction, it expects a tree that has exactly one child, named n, and returns this child, removing the edge n. In the putback direction, the value of the old concrete tree is ignored and a new one is created, with a single edge n pointing to the given abstract tree.

\[
(hoist \ n)(c) = \begin{cases} 
  t & \text{if } c = \{n \rightarrow t\} 
\end{cases}
\]

\[
(hoist \ n)(a, c) = \begin{cases} 
  \{n \rightarrow a\} & \text{if } a \in N 
\end{cases}
\]

∀C ⊆ T. ∀n ∈ N. (hoist \ n) ∈ \{n → C\} ⇐⇒ C

**Plunging**

This combinator is used to deepen a tree by adding an edge at the top. In the get direction, a new tree is created with a single edge n pointing to the given concrete tree. In the putback direction the value of the old concrete tree is ignored and the abstract tree is required to have exactly one subtree, labeled n which becomes the result.

\[
(plunge \ n)(c) = \{n \rightarrow c\}
\]

\[
(plunge \ n)(a, c) = \begin{cases} 
  t & \text{if } a = \{n \rightarrow t\} 
\end{cases}
\]

∀C ⊆ T. ∀n ∈ N. (plunge \ n) ∈ C ⇐⇒ \{n → C\}
Forking

This combinator applies different lenses to different parts of a tree. The get direction builds two subtrees:

- \( c_{| pc} \): the tree formed by dropping the immediate children of \( c \) whose names are not in \( pc \);
- \( c_{\setminus pc} \): the tree formed by dropping the immediate children of \( c \) whose names are in \( pc \);

Then, the get applies the first lens to \( c_{| pc} \), the result being a tree whose top-level labels are in \( pa \), and applies the second lens to \( c_{\setminus pc} \), the result being a tree whose top-level labels must not be in \( pa \). The putback direction does the same things, but it maps from \( pa \) to \( pc \) (Figure 3.1).

\[
\begin{align*}
(xfork pc pa l_1 l_2)(c) & = (l_1'(c_{| pc})) \cdot (l_2'(c_{\setminus pc})) \\
(xfork pc pa l_1 l_2)\gamma(a, c) & = (l_1'(a_{| pa}, c_{| pc})) \cdot (l_2'(a_{\setminus pa}, c_{\setminus pc})) \\
\forall pa, pc \subseteq N. \forall C_1 \subseteq T_{| pc}. \forall A_1 \subseteq T_{| pa}. \forall C_2 \subseteq T_{\setminus pc}. \forall A_2 \subseteq T_{\setminus pa}. \forall l_1 \in C_1 \Longleftrightarrow A_1. \forall l_2 \in C_2 \Longleftrightarrow A_2. \ (xfork pc pa l_1 l_2) \in C_1 \cdot C_2 \Longleftrightarrow A_1 \cdot A_2
\end{align*}
\]

Now we can define a list of derived lenses.
The combinator \textbf{fork} handles a split where the set of names are identical.

\[
\begin{align*}
(fork p l_1 l_2) & = (xfork p p l_1 l_2) \\
\forall p \subseteq N. \forall C_1, A_1 \subseteq T_{| p}. \forall C_2, A_2 \subseteq T_{\setminus p}. \forall l_1 \in C_1 \Longleftrightarrow A_1. \forall l_2 \in C_2 \Longleftrightarrow A_2. \ (xfork pc pa l_1 l_2) \in (C_1 \cdot C_2) \Longleftrightarrow (A_1 \cdot A_2)
\end{align*}
\]

The combinator \textbf{filter} discards all of the children of a three whose names are not in the set \( p \).
CHAPTER 3. COMBINATORS FOR BIDIRECTIONAL TRANSFORMATIONS

\[ (\text{filter } p \ d) = (\text{fork } p \ \text{id} \ (\text{const } \{\} \ d)) \]
\[ \forall p \subseteq N. \forall C \subseteq T. \forall d \in C \setminus p. \ (\text{filter } p \ d) \in (C|_{p} \cdot C|_{p}) \overset{\Omega}{\leftrightarrow} C|_{p} \]

The combinator \textit{prune} removes the subtree rooted in \( n \).

\[ (\text{prune } n \ d) = (\text{fork } \{n\} \ (\text{const } \{\} \ n \rightarrow d) \ \text{id}) \]
\[ \forall n \in N. \forall C \subseteq T. \forall d \in C|_{n}. \ (\text{prune } n \ d) \in (C|_{n} \cdot C|_{n}) \overset{\Omega}{\leftrightarrow} C|_{n} \]

The combinator \textit{add} adds the tree \( \{ n \rightarrow \{t\} \} \).

\[ (\text{add } n \ t) = (\text{xfork } \{n\} \ (\text{const } t \ \{\} ; \text{plunge } n) \ \text{id}) \]
\[ \forall n \in N. \forall C \subseteq T|_{n}. \forall t \in T. \ (\text{add } n \ t) \in C \overset{\Omega}{\leftrightarrow} (\{ n \rightarrow \{t\} \} \cdot C) \]

The combinator \textit{focus} focuses the attention on a single child \( n \).

\[ (\text{focus } n \ d) = (\text{filter } \{n\} \ d ; \text{hoist } n) \]
\[ \forall n \in N. \forall C \subseteq T|_{n}. \forall d \in C. \forall D \subseteq T. \ (\text{focus } n \ d) \in (\{ n \rightarrow D \} \cdot C) \overset{\Omega}{\leftrightarrow} D \]

The combinator \textit{hoist_nonunique} is the generalised version of \textit{hoist}.

\[ (\text{hoist_nonunique } n \ p) = (\text{xfork } \{n\} \ p \ (\text{hoist } n) \ \text{id}) \]
\[ \forall n \in N. \forall p \subseteq N. \forall D \subseteq T|_{n \setminus p}. \forall C \subseteq T|_{p}. \ (\text{hoist_nonunique } n \ p) \in (\{ n \rightarrow C \} \cdot D) \overset{\Omega}{\leftrightarrow} (C \cdot D) \]

The combinator \textit{rename} removes a single child.

\[ (\text{rename } m \ n) = (\text{xfork } \{m\} \ \{n\} \ (\text{hoist } m ; \text{plunge } n) \ \text{id}) \]
\[ \forall m, n \in N. \forall C \subseteq T. \forall D \subseteq T|_{\{m,n\}}. \ (\text{rename } m \ n) \in (\{ m \rightarrow C \} \cdot D) \overset{\Omega}{\leftrightarrow} (\{ n \rightarrow C \} \cdot D) \]

**Mapping**

The \textit{map} combinator is parameterized on a single lens \( l \). In the get direction this combinator applies \( l' \) to each subtree of the root and combines the results into a new tree. The putback direction needs two trees with the same domain and applies, for each element of the domain, the \( l' \setminus \) function to the two subtrees.

\[ (\text{map } l)'(c) = \{ n \rightarrow l'(c(n)) \mid n \in \text{dom}(c) \} \]
\[ (\text{map } l)'(a, c) = \{ n \rightarrow l'(a(n), c(n)) \mid n \in \text{dom}(a) \} \]
\[ \forall C, A \subseteq T. \text{dom}(C) = \text{dom}(A). \forall l \in (\cap_{n \in N} C(n) \overset{\Omega}{\leftrightarrow} A(n)). \ (\text{map } l) \in C \overset{\Omega}{\leftrightarrow} A \]

We can generalise this combinator by considering the argument a function mapping names to lenses.

\[ (\text{wmap } m)'(c) = \{ n \rightarrow m(n)'(c(n)) \mid n \in \text{dom}(c) \} \]
\[ (\text{wmap } m)'(a, c) = \{ n \rightarrow m(n)'(a(n), c(n)) \mid n \in \text{dom}(a) \} \]
\[ \forall C, A \subseteq T. \text{dom}(C) = \text{dom}(A). \forall m \in (\cap_{n \in N} C(n) \overset{\Omega}{\leftrightarrow} A(n)). \ (\text{wmap } m) \in C \overset{\Omega}{\leftrightarrow} A \]
CHAPTER 3. COMBINATORS FOR BIDIRECTIONAL TRANSFORMATIONS

Copying

This combinator copies the subtree of \( m \) by using a label \( n \) assuming that \( n \) is never used in the whole tree. In the get direction it creates the copy, while, in the putback direction, the combinator removes the label.

\[
\begin{align*}
\text{(copy } m \ n) \rightarrow (c) & = c \cdot \{ n \rightarrow c(m) \} \\
\text{(copy } m \ n) \leftarrow (a, c) & = a \backslash_n
\end{align*}
\]

\[
\forall m, n \in N. \forall C \subseteq T \setminus \{m, n\}. \forall D \subseteq T \\
(\text{copy } m \ n) \in (C \cdot \{ m \rightarrow D \cup \Omega \}) \Longleftrightarrow (C \cdot \{ \{ m \rightarrow d, n \rightarrow d \} \mid d \in D \cup \Omega \})
\]

Merging

This combinator merges the subtrees \( m \) and \( n \) assuming they are equal. In the get direction it removes the label \( n \) and, in the putback direction, the combinator duplicates the subtree from the abstract view (or concrete view if they are different).

\[
\begin{align*}
\text{(merge } m \ n) \rightarrow (c) & = c \backslash_n \\
\text{(merge } m \ n) \leftarrow (a, c) & = a \cdot \{ n \rightarrow a(m) \} \quad \text{if } c(m) = c(n) \\
& = a \cdot \{ n \rightarrow c(n) \} \quad \text{if } c(m) \neq c(n)
\end{align*}
\]

\[
\forall m, n \in N. \forall C \subseteq T \setminus \{m, n\}. \forall D \subseteq T \\
(\text{merge } m \ n) \in (C \cdot \{ m \rightarrow D \cup \Omega, n \rightarrow D \cup \Omega \}) \Longleftrightarrow (C \cdot \{ m \rightarrow D \cup \Omega \})
\]
Chapter 4

Proposal

In this chapter we present the thesis goals and analyze the potential benefits and issues.

4.1 Goals

As we showed in the previous chapters the subject of this thesis proposal is to handle the view/update problem with bidirectional transformations. In Chapter 2 we discuss several approaches to this problem with relational databases or tree-based formalisms (like XML), as well as an existing approach to solve the problem over graphs.

Our goal is to find a set of combinators for generic graph views over generic graph databases using the bidirectional transformation approach described in the previous chapter. With these combinators we can create graph views where updates are easily reflected over the graph database. These combinators can also be used over trees: rooted acyclic graphs with only one incoming edge for each node except for the root.

The first step toward the definition of a suitable set of combinators is the identification of a proper data model for graphs. In the following section we will describe a potential data model for graph databases: of course, this model could be improved/changed.

4.2 Model

In this section we will define a potential data model for graph databases.

A good starting point is the Resource Description Framework (RDF) proposed by W3C to represent data on the web using a directed, edge-labeled graph. In RDF a graph is a set of triples composed by:

- subject: the node where the edge starts;
- predicate: the propriety defined by a label;
- object: the node where the edge ends (or a basic value).
Example 4.1. Let $G$ be the graph in Figure 4.1, in RDF $P$ is defined by the following set:

$$P = \{ (A, a, B), (B, b, C), (C, c, A) \}$$

Figure 4.1: The graph of example 4.1

Given this graph representation, we can use set operations to build our combinators: for instance, we may use set difference and union to define the get and putback directions of a combinatory that deletes one edge from a graph.

In next section we will illustrate some ideas about these combinators.

Combinators

Finding combinators for graph transformations is our goal. We base our approach on the combinators introduced by Pierce et al. and described in Chapter 3: some of these basic combinators such as identity, composition, constant, and conditional, are valid, by construction, not only for trees but for every set of views; in particular, these are also valid for the set of graph views.

There are also many combinators that cannot be used for graph databases. For instance, combinators defined for tree structure cannot be reused in graph context, as they require a tree structure such as a root and acyclic edges. Other combinators can be nonsense at all over a graph. As an example, consider the hoist lens; this lens requires a parameter like $\{ n \rightarrow t \}$ and cannot be applied to a graph like the one in Example 4.1; moreover this operation is nonsense because this operation change the root of the tree but the graph haven’t a root.

The second issues that prevent the use of old combinators is the notation; can obviously represent a graph with a tree structure, as in the work of Takeichi et al. but the lenses must be redefined in order to obtain the same behavior: removing an edge from a tree is different to removing and edge from a graph represented by a tree.

The first step is to build basic lenses like that manage basic graph transformations such as: adding or removing an edge, adding or removing a node and mapping other lenses recursively starting from a certain node and following output or input edges.

The problem is however not trivial. Consider, for example, the operation that removes an edge labeled $l$ from $n$ to $m$. A naive lens definition would remove a tuple
\((n, l, m)\) in the get direction, and add the tuple \((n, l, m)\) in the putback direction. This lens can be defined as follow:

\[
(r_{\text{edge}} n l m)\nearrow(c) = c - (n, l, m)
\]
\[
(r_{\text{edge}} n l m)\searrow(a, c) = a + (n, l, m)
\]

Unfortunately this definition of the lens \(r_{\text{edge}}\) does not satisfy the GetPut propriety showed in Chapter 3. Consider:

\[
(r_{\text{edge}} A b B)\nearrow\left(\begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}\right) = \begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}
\]
\[
(e_{\text{edge}} A b B)\searrow\begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}
\]

\[
(e_{\text{edge}} A b B)\searrow\begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}
\]

The problem is that in the get direction we try to remove the edge using the difference between sets but the edge wasn’t present in the graph. Hence, in the putback direction we add an edge that was not present in the original graph, so we add an extraneous edge.

We can try to modify the lens this way:

\[
(r_{\text{edge}} n l m)\nearrow(c) = c - (n, l, m)
\]
\[
(r_{\text{edge}} n l m)\searrow(a, c) = a + (n, l, m) \quad \text{if} \quad (n, l, m) \in c
\]
\[
a \quad \text{if} \quad (n, l, m) \notin c
\]

This new version has still problems, as it does not satisfy the PutGet property. Consider:

\[
(r_{\text{edge}} A a B)\nearrow\left(\begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}\right) = \begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}
\]
\[
(r_{\text{edge}} A a B)\nearrow\left(\begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}\right) \neq \begin{array}{c}
(A, a, B) \\
(B, b, C) \\
(C, c, A)
\end{array}
\]

This time the problem is the following one: in the get direction we remove an edge, then we update the graph view by adding the same edge removed by the transformation; finally, when we putback the view into the concrete graph, the update to the view disappears because we added an edge that was already in the graph before the update.

These first issues means that the definition of these basic lenses over graphs are not trivial and they need some work.

The thesis will proceed as follow: we build a set of basic lenses following the approach described before. Then, we will build a set of derived lenses which implement more complicated transformations such as transitive closure or edge concentration. In
particular we are interested in building some lenses that cannot be simulated by standard relational operation such as filtering or join.

After formalised all these lenses and proved their well-behaveness our work can proceed by creating a system that implements these transformations.
Bibliography


