Toward Recursive View Update Strategies on Relations

Van-Dang Tran[3,1], Hiroyuki Kato[1,3] and Zhenjiang Hu[2,1]

[1]National Institute of Informatics, Tokyo, Japan
[2]Peking University, Beijing, China
[3]The Graduate University for Advanced Studies, SOKENDAI, Kanagawa, Japan

Abstract

Recent work has shown how to use non-recursive Datalog as a programming language for view update strategies in relational databases. In this paper, we extend the idea by considering recursions in the Datalog language for more interesting view definitions (forward transformation) and update strategies (backward transformation), especially over relations that store complex data structures such as trees and graphs. Firstly, we present how recursive bidirectional transformations over such relations can be formulated in Datalog and encapsulated in high-order logic predicates. Secondly, we discuss the validation problem for well-behavedness of the recursive Datalog programs.

Keywords

Relational, XML, Datalog, Structural Recursion, Logic Programming

1. Introduction

Recent work [1] has proposed an approach to use non-recursive Datalog - a well-known unidirectional language - to program view update strategies, i.e., backward transformations, more freely. The well-behavedness of a program is guaranteed by a validation algorithm. This is based on the interesting fact that while there may be many backward transformations for a forward transformation, there is at most one forward transformation for a backward transformation [2]. Therefore, the well-behavedness of a backward transformation can be validated by checking the existence of the corresponding forward transformation.

Datalog without recursion is a suitable and friendly language for implementing data transformations in relational database management systems where views are commonly defined in relational algebra without recursion. However, in deductive databases where relations are used to represent more complex data such as graphs or tree-like data structures, recursion in Datalog plays a central role.

In this paper, we extend the idea proposed in [1] to use recursive Datalog with extensions such as negation in programming more interesting bidirectional transformations over relations that represent complex data structures such as trees and graphs.

Manually writing a recursive view update strategy is an expensive task for programmers and it is even more challenging to automatically validate such a program. The fixed-point semantics

Bx 2021: 9th International Workshop on Bidirectional Transformations, part of STAF, June 21, 2021
dangtv@nii.ac.jp (V. Tran); kato@nii.ac.jp (H. Kato); huzj@pku.edu.cn (Z. Hu)

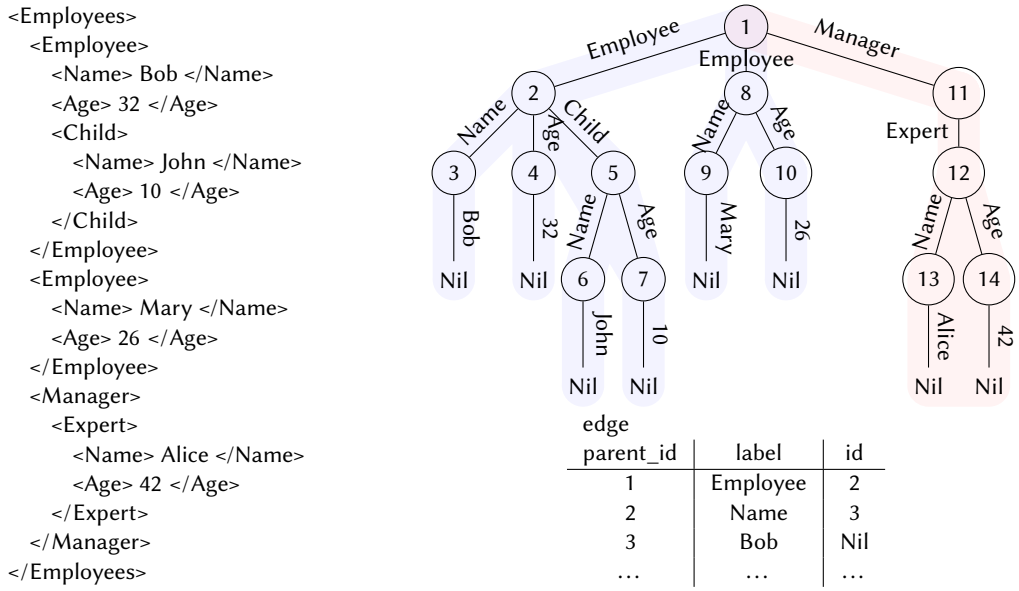

Figure 1: Storing an XML document in a single relation `edge`.

of Datalog makes the well-behavedness property not expressible in first-order logic.

Our key idea to solve the aforementioned challenge is to formulate a Datalog-written view update strategy into two parts: one for recursive patterns and another for inner update strategies. On the one hand, the recursive Datalog rules of the first part are predefined and pre-validated so that their well-behavedness is guaranteed. On the other hand, we allow programmers to manually write the inner update strategies in Datalog without recursion, which can be validated automatically. To combine these non-recursive rules with the predefined recursive ones, we extend Datalog with high-order predicates as in HiLog [3]. Specifically, the predefined recursive Datalog rules can be encapsulated in high-order predicates and called later in non-recursive rules written by users, and vice versa.

2. Recursive Bidirectional Transformations in Datalog

2.1. Shredding tree-structured data into relations

For tree-structured data such as XML or JSON documents, we use the Edge shredding approach [4, 5] to store all the edges of the graph that represents the document. For simplicity, we consider unordered XML documents. Figure 1 shows an XML document of employees that is represented by a tree and stored in a single relation `edge(parent_id,label,id)`. Each tuple in `edge` represents a node of the XML document, where `id` and `parent_id` are the ids of the node and its parent node, respectively, and `label` is the node's tag name. We consider the value of a node as a tag name of a special child node `Nil`. For example, a tuple `edge(3,'Bob',Nil)` represents that the value of node 3 is '`Bob`'. Querying over the XML document now can be written in Datalog, a well-known relational data query language. Datalog can be considered as

```
1  % splitting the source
2  subedge_l(X, L, Y) :- edge(X, L, Y),
                         L = 'Employee', ¬ edge(_, _, X).
3  subedge_l(X, L, Y) :- subedge_l(_, _, X),
                         edge(X, L, Y).
4  subedge_r(X, L, Y) :- edge(X, L, Y),
                         ¬ subedge_l(X, L, Y).
5  % two inner forward transformations
6  subedge_l^view(X, L, Y) :- subedge_l(X, L, Y).
7  subedge_r^view(X, L, Y) :- subedge_r(X, L, Y),
                              subedge_r(_, _, X).
8  % merging
9  edge^view(X, L, Y) :- subedge_l^view(X, L, Y).
10 edge^view(X, L, Y) :- subedge_r^view(X, L, Y).
```

```
11 % splitting the view
12 subedge_l^view(X, L, Y) :- edge^view(X, L, Y),
                              L = 'Employee', ¬ subedge_r(_, _, X).
13 subedge_l^view(X, L, Y) :- subedge_l^view(_, _, X),
                              edge^view(X, L, Y).
14 subedge_r^view(X, L, Y) :- edge^view(X, L, Y),
                              ¬ subedge_l^view(X, L, Y).
15 % two inner backward transformations
16 subedge_l^new(X, L, Y) :- subedge_l^view(X, L, Y)
17 subedge_r^new(X, L, Y) :- subedge_r^view(X, L, Y).
18 subedge_r^new(X, L, Y) :- subedge_r(X, L, Y),
                              ¬ subedge_r(_, _, X).
19 % merging
20 edge^new(X, L, Y) :- subedge_l^new(X, L, Y).
21 edge^new(X, L, Y) :- subedge_r^new(X, L, Y).
```

Figure 2: Forward (on the left) and backward (on the right) transformations.

Prolog without functions. The recursion in Datalog makes it a suitable language for queries over the `edge` relation since recursive computation is required for tree traversals.

Example 2.1. The following recursive Datalog rules extract from the tree in Figure 1 the subtree (the left subtree, stored in relation $\texttt{subedge}_l$) that contains only employees:

```
(r1)  subedge_l(X, L, Y) :- edge(X, L, Y), L = 'Employee', ¬ edge(_, _, X).
(r2)  subedge_l(X, L, Y) :- subedge_l(_, _, X), edge(X, L, Y).
```

Each Datalog rule represents a first-order logic sentence where the conjunction of all predicates in the rule body (right-hand side) implies the predicate in the rule head (left-hand side), and all variables are universally quantified [6]. Computing the relation in the rule head is computing the smallest instance that satisfies all the Datalog rules. For $\texttt{subedge}_l$ in these two rules, the first rule gives $\texttt{subedge}_l$ an initial instance of all outgoing edges `edge(X,L,Y)` of label 'Employee' from the root node `X` that has no parent node (¬`edge(_,_,X)`). The second rule recursively adds to $\texttt{subedge}_l$ all the outgoing edges from a node X that is reachable in $\texttt{subedge}_l$. □

2.2. Bidirectional Transformations over the Tree

We shall use a tree lens combinator `xfork` presented in [7] to demonstrate how Datalog can be used to implement such a bidirectional transformation over a tree stored in a relation `edge`. As in `xfork`, we first split the original tree in Figure 1 into two subtrees (a left subtree and a right subtree) according to the labels of the outgoing edges from the root. For the left subtree, we apply an identity transformation. For the right subtree we apply a simple bidirectional transformation (similar to the `hoist` lens in [7]) that removes the outgoing edge from the root `edge(1,'manager',11)` (to consider an expert, who is a manager, as an employee). Finally, the two tree views obtained from the two inner bidirectional transformations are merged into a single tree. Figure 2 shows the forward and backward transformations written in Datalog.

The forward direction works as follows. To split the original tree, we use two recursive Datalog rules in Example 2.1 to extract the left subtree $\texttt{subedge}_l$ (the tree of nodes 1 to 10).

Then the right subtree $\texttt{subedge}_r$, which is the remaining part, is the one that has all the edges in edge but not in $\texttt{subedge}_l$ and is computed by a rule in Line 4. Line 6 calculates a view as an identity instance of $\texttt{subedge}_l$. Line 7 calculates a view by excluding the edge from the root of $\texttt{subedge}_r$. In this rule, the second predicate of the body makes sure that node X has a parent node, i.e., node X is not the root. Lines 9 and 10 merge the two views into one $\texttt{edge}^{view}$ by taking the union. If the root nodes of these views have different IDs, we write an additional rule to unify them.

In the backward direction, we have similar operations. Lines 12, 13, and 14 split the view $\texttt{edge}^{view}$. Line 16 calculates the first new source $\texttt{subedge}_l^{new}$ for $\texttt{subedge}_l^{view}$ as the same instance. Lines 17 and 18 compute the second new source $\texttt{subedge}_r^{new}$ by taking all the edges of the view $\texttt{subedge}_r^{view}$ and the outgoing edge from the root in the original source $\texttt{subedge}_r$. Predicate $\neg\texttt{subedge}_r\texttt{(\_,\_,X)}$ in Line 18 is to ensure that node X has no parent node, i.e., X is the root. Lines 20 and 21 merge the two new sources.

The forward and backward transformations in Figure 2 can be considered as program templates for writing more bidirectional transformations over the tree. The Datalog rules of splitting and merging operations can be adjusted by changing the constant '`Employee`' to another label. The two inner bidirectional transformations (Rules 6, 7, 16, 17, and 18) can be arbitrarily implemented. Since the splitting and merging operations are fixed, the well-behavedness of the program templates is guaranteed if the two inner bidirectional transformations are well-behaved.

Our observation is that an inner bidirectional transformation works on a component that is structurally smaller than the original tree, and thus is easier to manually implement. Meanwhile, writing decomposing operations, e.g., splitting, requires more knowledge about recursion. This suggests a new approach to make bidirectional transformations over tree-structured data programmable in Datalog. The key idea is to predefine a variety of program templates that employ the recursive computation power of Datalog to work on recursive data structures like trees. Programmers can reuse these templates by manually implementing inner bidirectional transformations without recursion.

Our example shows a typical program over a recursive data structure with three operations: decomposing (splitting) the tree, then processing each component, and merging the results. By recursively decomposing all components, we obtain structural recursions over the tree. In our program templates, we predefine recursive Datalog rules to deal with these recursion forms. We consider only a fixed number of commonly used recursive patterns that always terminate and are sufficiently expressive in practice. To enhance the reusability of program templates, we shall encapsulate Datalog rules in high-order predicates that can be easily recalled.

2.3. Encapsulating Datalog rules in high-order predicates

We can consider $\texttt{subedge}_l$ and $\texttt{subedge}_r$ of the splitting operations (Rules 2, 3, and 4 in Figure 1) as a pair resulting from a `split` function over a relation edge and a constant '`Employee`'. The pair can be represented by a function $\{\texttt{'l'} \mapsto \texttt{subedge}_l, \texttt{'r'} \mapsto \texttt{subedge}_r\}$. Therefore, we have: $\texttt{subedge}_l$ `= split(edge,'Employee')('l')` and $\texttt{subedge}_r$ `= split(edge,'Employee')('r')`. Rules 2, 3, and 4 can be parameterized with `src` and `Label` as follows:

```
split(src, Label)('l')(X, L, Y) :- src(X, L, Y), L = Label, ¬ src(_, _, X).
split(src, Label)('l')(X, L, Y) :- split(src, Label)('l')(_, _, X), src(X, L, Y).
```

```
split(src, Label)('r')(X, L, Y) :- src(X, L, Y), ¬split(src, Label)('l')(X, L, Y).
```

Where `split` can be considered as a high-order logic predicate as in the HiLog syntax [3].

Similarly, we can have a predicate `merge` for the merging operation. By encapsulating the two inner backward transformations in two predicates put_1 and put_2, the whole backward transformation can also be encapsulated in a single predicate:

```
fork_put(put1, put2, Label)(src, view)(X, L, Y) :- merge(
        put1(split(src, Label)('l'), split(view, Label)('l')),
        put2(split(src, Label)('r'), split(view, Label)('r')) )(X, L, Y).
```

More interestingly, the corresponding forward transformations get_1, get_2, and $fork_{get}$ can be uniquely determined from their backward transformations put_1, put_2, and $fork_{put}$, respectively [8, 2]. Therefore, the high-order predicates of put_1, put_2, and $fork_{put}$ are sufficient to capture the corresponding bidirectional transformations.

3. Validation

As presented in Section 2, a Datalog program template consists of predefined Datalog rules and user-written rules. Validating the well-behavedness of bidirectional transformations specified by these rules plays a central role in our approach. While the predefined part can be pre-validated, an automated validator is required for statically checking the inner bidirectional transformation programs, which are arbitrarily written.

Although many properties of Datalog rules can be reduced to the satisfiability of Datalog queries, it is well known that the satisfiability problem of a Datalog query is undecidable [9]. Fortunately, the predefined Datalog rules have performed more complicated computations such as recursion so that the user-written inner bidirectional transformations are much simpler, and thus do not require the full syntax of the Datalog language. The more sophisticated the predefined part is, the simpler the user-written part is, and thus it can be expressed in a restricted class of Datalog where the program's well-behavedness is decidable. Indeed, for example, all the Datalog rules of the inner bidirectional transformations in Figure 2 are in guarded-negation Datalog (GN-Datalog) where the satisfiability problem is decidable [10].

Recursive Datalog rules in program templates are predefined to deal with common forms of structural recursion rather than arbitrary recursions. Therefore, to validate a property of these recursive rules, we can apply structural induction on the data structure to construct a proof for the property. The main challenge to applying structural induction is that the data structure is not explicitly described by an inductive type but shredded into an `edge` relation. The structure is preserved by constraints on the `edge` relation, e.g., primary/foreign keys on `id` and so forth.

4. Conclusion

We have presented our ideas to use Datalog as a programming language for bidirectional transformations over tree-structured data. Although this short paper has demonstrated a simple example, we believe Datalog is expressive enough to cover many interesting recursive view update strategies over trees and even some restricted graphs.

Acknowledgments This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099.

References

[1] V.-D. Tran, H. Kato, Z. Hu, Programmable view update strategies on relations, PVLDB 13 (2020) 726–739.

[2] S. Fischer, Z. Hu, H. Pacheco, The essence of bidirectional programming, Science China Information Sciences 58 (2015) 1–21.

[3] W. Chen, M. Kifer, D. S. Warren, HILOG: A foundation for higher-order logic programming, J. Log. Program. 15 (1993) 187–230.

[4] D. Florescu, D. Kossmann, Storing and querying XML data using an RDMBS, IEEE Data Eng. Bull. 22 (1999) 27–34.

[5] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, C. Zhang, Storing and querying ordered XML using a relational database system, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002, 2002, pp. 204–215.

[6] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about datalog (and never dared to ask), TKDE 1 (1989) 146–166.

[7] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt, Combinators for bi-directional tree transformations: A linguistic approach to the view update problem, in: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005, pp. 233–246.

[8] S. Fischer, Z. Hu, H. Pacheco, A clear picture of lens laws, in: International Conference on Mathematics of Program Construction, 2015, pp. 215–223.

[9] O. Shmueli, Equivalence of datalog queries is undecidable, The Journal of Logic Programming 15 (1993) 231 – 241.

[10] V. Bárány, B. ten Cate, M. Otto, Queries with guarded negation, Proc. VLDB Endow. 5 (2012) 1328–1339.