A Visual Query Language for Complex-Value Databases

Christoph Koch
Saarland University Database Group, Saarbrücken, Germany
koch@infosys.uni-sb.de

ABSTRACT

In this paper, a visual language, VCP, for queries on complex-value databases is proposed. The main strength of the new language is that it is purely visual: (i) It has no notion of variable, quantification, partiality, join, pattern matching, regular expression, recursion, or any other construct proper to logical, functional, or other database query languages and (ii) has a very natural, strong, and intuitive design metaphor. The main operation is that of copying and pasting in a schema tree.

We show that despite its simplicity, VCP precisely captures complex-value algebra without powerset, or equivalently, monad algebra with union and difference. Thus, its expressive power is precisely that of the language that is usually considered to play the role of relational algebra for complex-value databases.

1. INTRODUCTION

Even though most modern database query languages are based on logical or algebraic foundations (or a combination of these, as is the case for SQL) to allow for the declarative or at least abstract specification of queries, inexperienced users are often overwhelmed by the task of writing queries. This has motivated efforts to develop easy-to-use *visual* query languages.

The essential issue in defining good visual query languages is to provide strong visual metaphors for the constructs in a query and the steps to be taken to define it. Only by such strong visual metaphors can a visual language become easy to use for inexperienced users. Examples of such metaphors for data management operations in the wider sense are, e.g., deletion by dragging an icon representing a data object onto a garbage can or data relocation by dragging and dropping icons in a directory tree.

We distinguish between (1) graphical (or graph-based) query languages such as QBE [19] and Graphlog [9] and (2) visual languages in which the specification of the query is a process, i.e., the query is defined by the interaction with the user rather than by the static graphical outcome.[1] For example, manipulating a directory hierarchy in an operating system window manager usually involves a sequence of insertion, copy, and deletion steps which together define the transformation on the file system to be performed. By just considering the outcome – a directory structure – it is not possible to tell which transformation was carried out.

The interaction of a user with a system may give a surprising amount of expressive power while allowing the visual approach to remain intuitive. Static graphical languages tend to offer more powerful constructs (such as variables and quantifiers) to compensate for this. Even if such constructs are provided via graphical objects, queries still closely correspond to traditional textual query languages, and much of the appeal of visual specification is lost.

QBE [19] is a visual query language for relational databases and an attractive alternative to relational algebra. In its original form, it is widely taught but has failed to have a large impact on database practice. One reason for this may be its reliance on variables, which are required to perform joins, and may be a concept hard to deal with for non-expert users. One seemingly close cousin of QBE, the visual language employed in Microsoft Access, avoids the use of variables by replacing them by lines that connect table columns to be joined. Still, lines are an unsuitable metaphor for multi-argument and multi-way joins. The otherwise intuitive design metaphor of QBE (and equally the visual query language of Access) also breaks when one wants to go beyond conjunctive queries to employ union and difference in queries.

Even greater problems are encountered when a visual language is sought for nested relational or complex-value databases [12, 1, 2]. QBE-like tables can be visually nested within each other easily, but there seems to be no clear – and sufficiently expressive – semantics to multiple occurrences of variables (which give us joins in QBE). Such a semantics would need to be able to express functionality beyond joins such as nesting and unnesting as well. Previous work on languages for complex-value databases has therefore either resulted in expressively rather weak languages or has compromised simplicity (the existence of a strong visual design metaphor) to obtain the right expressiveness.

Besides QBE, another example of a language of the first kind is visXCerpt [6], a GUI-based query language for XML that uses graphical primitives for notions such as variables, quantification, recursion, and partiality. The expressive po-

[1]Languages of the first class are traditionally called visual as well.

wer of visXCerpt has not been formally studied, but appears to be very high (Turing-complete). QURSED [14], another GUI-based XML query language, trades in expressiveness for simplicity (even though no formal study of its expressive power is available). There are no explicit graphical objects for variables, but the user is required to have a notion of concepts such as variables and Boolean combinations of conditions in order to successfully use the query builder. Both the visXCerpt and the QURSED tool use a number of visual operations to build queries (such as dragging and dropping data locations); however, the query is the static outcome of this process.

XQBE [3] and XML-GL [8] are graphical languages for queries on XML. In both, there is no notion of explicit variables; instead lines are drawn similarly to the visual tool of MS Access. A query consists of a source- and a construct-part (both based on graphical schema representations). No study of the expressive power of these languages is available, but the queries seem to correspond to the nested (XML) analogs of restricted classes of conjunctive queries.

In Graphlog [9], queries are graphs with node and edge annotations, which are to be matched against a graph database (in the spirit of finding subgraph homomorphisms). Graphlog handles a class of linear recursive queries by allowing for certain regular expressions over relation names on edges, where such relations can be defined by Graphlog query graphs and used cyclically. When recursion is used sparingly, Graphlog queries tend to be easy to read. The expressive power of Graphlog is studied formally in [9] and it turns out that the language has a number of nice characterizations. However, only binary relations can be defined, so no general data transformations are possible and the expressiveness does not match that required for complex-value databases. Similar approaches are taken in G-Log [15] and in GraphLog's predecessor G+ [10].

Lixto [5], a visual Web wrapper generator, is based on an intuitive metaphor for information extraction from Web pages. (The selection of regions in a Web page with the mouse and the association of "patterns" to such regions.) However, Lixto only covers unary (tree node-selecting) queries rather than data transformation queries, which is sufficient in the wrapping context. The core language of Lixto has been shown to capture precisely an important and well-studied class of queries, those definable in monadic second-order logic over trees [11]; however, to achieve this, Lixto has to resort to recursion [4], for which no visual metaphor is provided.

To this day, there is no truly visual language that captures the expressiveness of any of the clean theoretically-founded data transformation languages such as relational algebra or complex-value algebra [1, 2]. This paper aims to improve on this situation by proposing a language that appears to satisfy these desiderata. Our contributions are as follows.

- A visual language, VCP, for queries on complex-value databases is proposed. The main strength of the new language is that it is purely visual: (i) It has no notion of variable, quantification, partiality, join, pattern matching, regular expression, recursion, or any other construct proper to logical, functional, or other database query languages and (ii) has a very natural, strong, and intuitive design metaphor.

  The only operations available and needed in VCP are that of copying and pasting in a *schema tree* as well as inserting, renaming, and deleting nodes and filtering sets. A schema tree only uses three kinds of nodes, namely set-typed and tuple-typed nodes and atomic value leaves.

  The only (oblique) advanced notion that the user has to deal with is that of a collection (a set). Schema trees can be understood and treated similarly to directory trees as they have become commonplace in OS window managers (with directories as collections of files).

- We show that despite its simplicity, VCP precisely captures the complex-value algebra without powerset [1], or equivalently, monad algebra with union and difference [17, 7]. Thus, its expressive power is precisely that of the language that is usually considered to play the role of relational algebra for complex-value databases (cf. [1, 2]).

  These languages are also a foundation of – and probably very similar in expressive power too – to XML query languages such as XQuery. Even though this remains to be verified, this renders it likely that VCP may give rise to the very first well-founded visual XML query language.

VCP is a member of the second class of query languages defined above – queries are defined as an interactive process. It turns out that this interaction gives us so much expressiveness that we need only very simple query constructs and operations.

Example 1.1. We discuss an example VCP query informally to provide a first idea of the language. Consider a relational database with two relations $books(\underline{isbn}, title, year)$ and $authors(\underline{isbn, name})$, modeling books and their (possibly multiple) authors.

Throughout this paper, we will focus on a complex-value data model in which relations may be nested into each other. A graphical (tree-based) representation of the schema (a *schema tree*) is shown in Figure 1 (a), and detailed definitions will follow in the technical sections of the paper. "Dom" denotes the domain of atomic values such as strings and integers. A corresponding data tree – modeling two books and their authors – is shown in Figure 1 (b).

We will specify a query that maps every complex value database of our given schema to a complex value consisting of a set of book-tuples of the form $\langle isbn, title, authors\rangle$, where "authors" is the set of authors of the book. That is, this query *nests* the authors of each book into their book tuple. The intended query result and a corresponding schema tree can be found in Figures 2 (b) and (a), respectively.

In VCP, we can define this query visually, by a number of interactive modifications of the schema tree. We proceed as follows. (1) First we execute a bulk copy operation of the "authors" relation into each tuple of the "books" relation by simply dragging the "authors" subtree of the schema tree onto the node representing the "books" tuples (adding another relation-typed attribute/column to the "books" relation). This is shown in Figure 1 (c).

(2) Then we delete the "year" edge from the schema tree, which removes this attribute from "books" and thus the years from all book entries in the database. (Because of space limitations, not all steps have their own figure; this operation can be found in Figure 1 (c).)

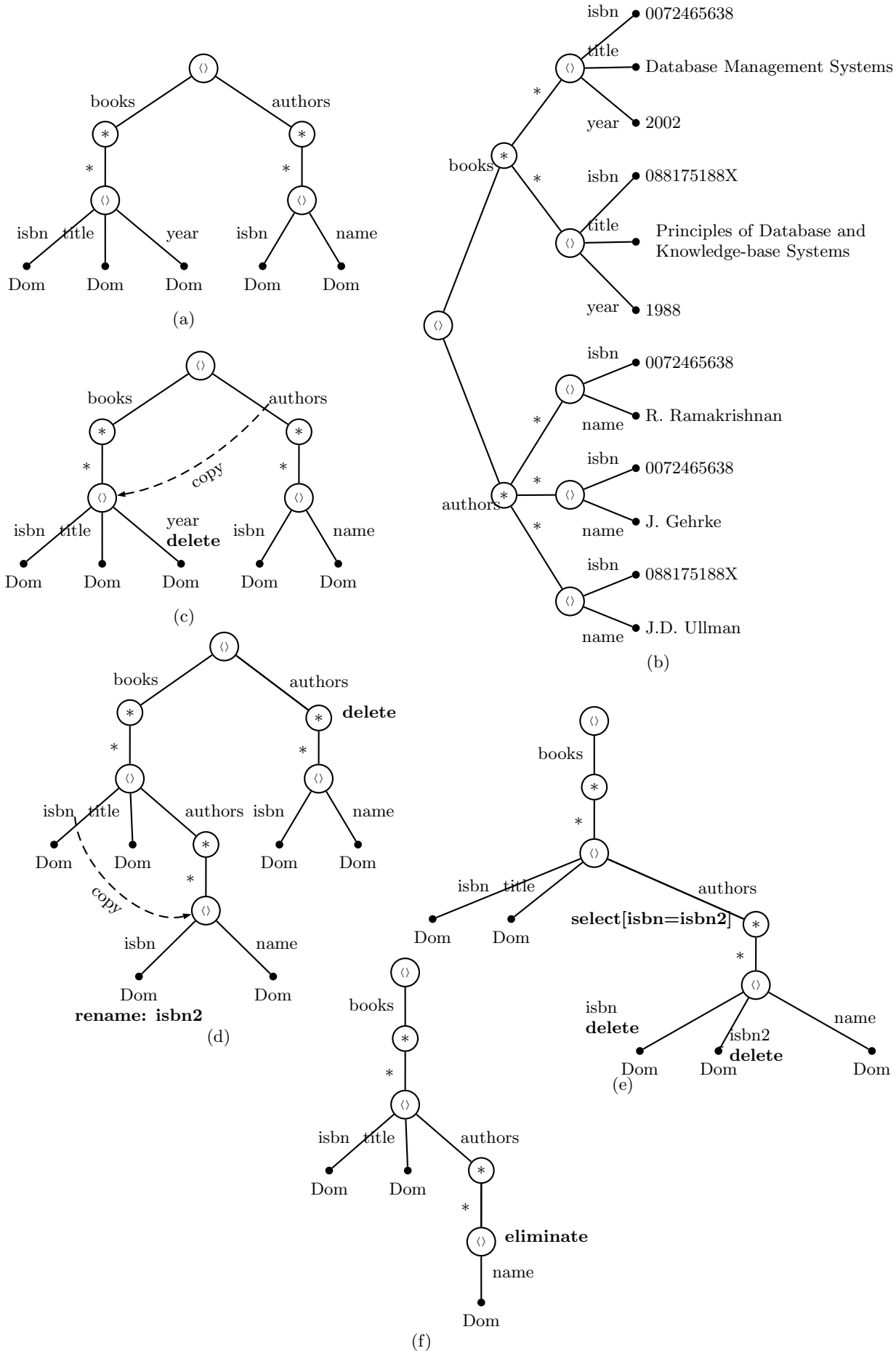


Figure 1: Initial data tree (b) and query steps (a), (c)-(f) of Example 1.1.

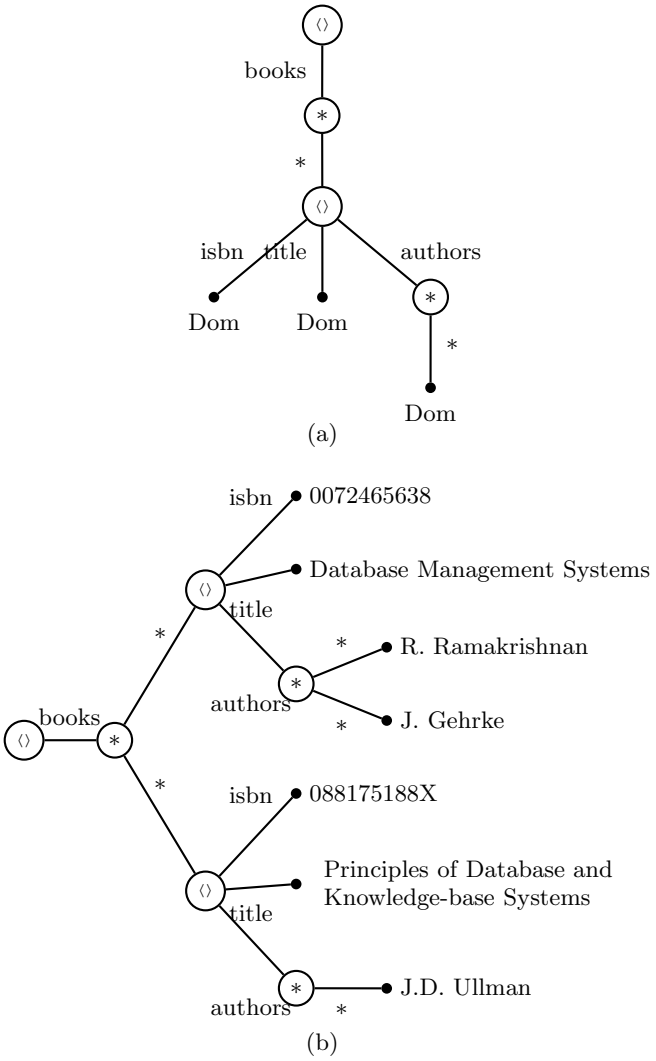


Figure 2: Conclusion of Example 1.1.

(3) Next we make a bulk copy of the "isbn" attribute of each book tuple t to each of the author tuples in the "authors" attribute of t. This is performed by dragging the isbn edge emanating from the node representing the book tuples to the node representing the author tuples nested inside books (see Figure 1 (d)). In order not to have two "isbn" attributes in the nested authors, though, we first rename the "isbn" attribute of the nested authors to "isbn2".

(4) We will not need the original authors relation anymore, so we can remove it from the complex value computed as the query result by deleting the top-level authors subtree from the schema tree (Figure 1 (d)).

(5) Now we apply a "select" operation on the schema tree node corresponding to the "authors" relation nested inside the books and filter out those authors for which isbn $\neq$ isbn2, i.e., for each book tuple t, we remove those author tuples that really belong to book t (Figure 1 (e)).

(6) Then we can eliminate the "isbn" and "isbn2" attributes of the nested author tuples by deleting their subtrees (Figure 1 (e)).

(7) Finally, we eliminate the "authors" tuple node – which has only one child (in other terms, a single attribute). This transforms, for each book tuple, the "authors" attribute from a set of unary tuples to a set of domain values (author names) (Figure 1 (e)). [2]

As stated above, the final schema tree and the query result are shown in Figure 2. □

Further examples of VCP queries can be found in Figures 3, 4, and 5. However, these examples fulfill a double duty, and serve to demonstrate certain expressiveness arguments. They are somewhat more abstract.

The structure of the paper is as follows. First, in Section 2, we introduce types for complex values and their corresponding schema trees. Moreover, we give an introduction to monad algebra. Section 3 presents the language VCP and gives a number of examples. In Section 4, it is shown that VCP precisely captures the expressive power of monad algebra. We conclude with a discussion of VCP and future work (Section 5).

2. PRELIMINARIES

2.1 Schema Trees

We model complex values constructed from sets, tuples, and atomic values from a single-sorted domain[3] in the normal way. Types are terms of the grammar

$$\tau ::= \text{Dom} \mid \{\tau\} \mid \langle A_1 : \tau_1, \ldots, A_k : \tau_k \rangle$$

where $k \geq 0$. Type terms have an obvious tree representation; we will call such trees *schema trees*.

Note that schema trees have three kinds of nodes – tuple, set, and atomic value type nodes – and two kinds of edges – tuple and set-edges, of which the tuple edges carry an attribute name as label. An example of a schema tree for the type $\langle R : \{\langle A : \tau_1, B : \tau_2 \rangle\}, S : \{\langle C : \tau_3, D : \tau_4 \rangle\}\rangle$ can be found in Figure 3 (a). If $\tau_1 = \tau_2 = \tau_3 = \tau_4 = \text{Dom}$, this type is an appropriate representation of relational schema $R(AB), S(CD)$. (Here and later we model a relational database as a *tuple* of relations to get a single complex value.)

2.2 Monad Algebra

Consider the query language on complex values consisting of expressions built from the following operations (the types of the operations are provided as well):

1. identity

$$\text{id} : x \mapsto x \qquad \tau \to \tau$$

2. composition

$$f \circ g : x \mapsto g(f(x)) \qquad \frac{f : \tau \to \tau',\ g : \tau' \to \tau''}{f \circ g : \tau \to \tau''}$$

3. constants from $\text{Dom} \cup \{\emptyset, \langle\rangle\}$ ($\langle\rangle$ is the nullary tuple)

4. singleton set construction

$$\text{sing} : x \mapsto \{x\} \qquad \tau \to \tau'$$

[2]Visually, this operation cuts out a node from a path in the schema tree, rather than deleting the subtree rooted by the node eliminated.

[3]All results in this paper immediately generalize to many-sorted domains.

5. application of a function to every member of a set

$$\text{map}(f) : X \mapsto \{f(x) \mid x \in X\}$$

$$\frac{f : \tau \to \tau'}{\text{map}(f) : \{\tau\} \to \{\tau'\}}$$

6. flatten: $X \mapsto \bigcup X \qquad \{\{\tau\}\} \to \{\tau\}$

7. pairing[4]

$$\text{pairwith}_{A_1} : \langle A_1 : X_1, A_2 : x_2, \ldots, A_n : x_n\rangle \mapsto \{\langle A_1 : x_1, A_2 : x_2, \ldots, A_n : x_n\rangle \mid x_1 \in X_1\}$$

$$\langle A_1 : \{\tau_1\}, A_2 : \tau_2, \ldots, A_n : \tau_n\rangle \to \{\langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle\}$$

8. tuple formation

$$\langle A_1 : f_1, \ldots, A_n : f_n\rangle : x \mapsto \langle A_1 : f_1(x), \ldots, A_n : f_n(x)\rangle$$

$$\frac{f_1 : \tau \to \tau_1, \ldots, f_n : \tau \to \tau_n}{\langle A_1 : f_1, \ldots, A_n : f_n\rangle : \tau \to \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle}$$

9. projection

$$\pi_{A_i} : \langle A_1 : x_1, \ldots, A_i : x_i, \ldots, A_n : x_n\rangle \mapsto x_i$$

$$\pi_{A_i} : \langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle \to \tau_i$$

The language has a strong and clean theoretical foundation from programming language theory, for which we have to refer to e.g. [17].

Returning to the definition of our operations, note that projection is applied to tuples rather than to sets of tuples as in relational algebra. For example, the relational algebra expression π_{AB} (on some relation with at least columns A and B) corresponds to $\text{map}(\langle A : \pi_A, B : \pi_B\rangle)$ in $\mathcal{M}$.

By *positive monad algebra* $\mathcal{M}_\cup$, we denote $\mathcal{M}$ extended by the set union operation. This language has a number of nice properties [17, 7], but it is known that it is incomplete as a practical query language because it cannot yet express selection, set difference, or set intersection.

However, if we extend $\mathcal{M}_\cup$ by any nonempty subset of the operations selection (of the form $\sigma_{A=B}$ on complex values of type $\{\langle A : \tau, B : \tau', \ldots\rangle\}$, where "$=$" denotes deep equality of complex values), set difference "$-$", set intersection $\cap$, or nesting[5], we always get the same expressive power. We will call any one of these extended languages *full monad algebra*.

THEOREM 2.1 ([17]). *$\mathcal{M}_\cup[\sigma] = \mathcal{M}_\cup[-] = \mathcal{M}_\cup[\cap] = \mathcal{M}_\cup[nest]$.*

Moreover, generalizing selections to test against constants or to support "$\in$", "$\subseteq$", or Boolean combinations of conditions does not increase the expressiveness of full monad algebra [17].

[4]Operations pairwith_{A_i} can be defined analogously.

[5]The "nest" operation of complex value algebra without powerset [1] groups tuples by some of their attributes. For example, $\text{nest}_{C=(B)}(R)$ on relation $R(AB)$ computes the value $\{\langle A\colon x, C\colon \{\langle B\colon y\rangle \mid \langle A\colon x, B\colon y\rangle \in R\}\rangle \mid (\exists y)\langle A\colon x, B\colon y\rangle \in R\}$.

EXAMPLE 2.2. The query of Example 1.1 can be phrased in $\mathcal{M}_\cup[\sigma]$ as

$$\begin{aligned}\langle books : \text{pairwith}_{books} \circ \\ \text{map}(\langle isbn : \pi_{books} \circ \pi_{isbn}, title : \pi_{books} \circ \pi_{title}, \\ authors : \langle 1 : \pi_{books} \circ \pi_{isbn}, 2 : \pi_{authors}\rangle \circ \\ \text{pairwith}_{authors} \circ \text{map}(\langle isbn : \pi_1, \\ isbn2 : \pi_2 \circ \pi_{isbn}, name : \pi_2 \circ \pi_{name}\rangle) \circ \\ \sigma_{isbn=isbn2} \circ \text{map}(\pi_{name})\rangle)\rangle\end{aligned}$$

That is, we first pair each book tuple with the set of all authors. Using the "map" operation, we then process each of these pairs as follows. For each book-author pair, we pair the book's isbn number with each of the author tuples. Then we are able to select those authors that belong to the book (using σ). Each book tuple is made a triple of an isbn number, a title, and a set of authors. The latter is a set of names – rather than tuples of isbn, isbn2, and name – created by mapping π_{name} onto the set of author tuples. □

Theorem 2.1 demonstrates that full monad algebra (say, $\mathcal{M}_\cup[\sigma]$) is a very robust notion. It can serve as an "expressiveness benchmark" for query languages on complex-value databases. Indeed, it has been shown that full monad algebra is a *conservative extension* of relational algebra:[6]

THEOREM 2.3 ([16]). *A mapping from a (flat) relational database to a (flat) relation is expressible in $\mathcal{M}_\cup[\sigma]$ if and only if it is expressible in relational algebra.*

It was discovered (see e.g. [17] for a detailed discussion) that virtually all the complex value query languages developed in the eighties and nineties that were intended for practical use are expressively equivalent. Monad algebra is one of them, and so are *nested relational algebra* [12] and *complex value algebra without the powerset operation* [1, 2]. Thus, the expressiveness results obtained in this paper for VCP show that this language is equivalent to all of them.

3. THE VCP LANGUAGE

In this section, we introduce VCP, a query language that transforms complex value databases through a sequence of operations on a schema tree. VCP has two dynamic aspects, the first being how VCP operations transform the schema tree at the time of specification (discussed in Section 3.1), and the second being the semantics of VCP queries on data (Section 3.2). We provide three larger examples in Section 3.3.

3.1 Query Specification as a Process

Let $\text{nca}(v, e)$ denote the nearest common ancestor node of node v and edge e in the schema tree.

The operations of the VCP language are presented in Table 1. Operations 8 and 9 are implemented visually by drag&drop in the schema tree. The other operations are local to a node or edge in the schema tree.

We have been parsimonious with the number of operations that have been introduced in Table 1; to make query specification faster and more convenient, further visual operations can be added, such as a "delete subtree" operation on set edges (which empties sets). [7]

[6]A generalized version of Theorem 2.3 can be found in [18].

[7]This operation is redundant with tuple insertion, subtree

	Name & arguments	Applied to	Conditions	Function	Example
1	new constant (A, c)	tuple node v	-	Adds a new attribute A with constant value $c \in \text{Dom} \cup \{\emptyset, \langle\rangle\}$ to v.	Fig. 4 (d)
2	insert tuple (A)	node v	-	Replaces the subtree rooted by v by $\langle A : v\rangle$, i.e., by a new tuple-node with v as A-child.	Fig. 6 (a)
3	insert set	node v	-	Replaces the subtree rooted by v by $\{v\}$, i.e. by a new set-node with v as child	
4	rename (A)	tuple edge e	-	Renames the label of e (= an attribute of a tuple) to A	Fig. 4 (b)
5	eliminate	set node v	The parent node of v is also a set node	cuts v: let $v = \{\tau\}$. Then, v is replaced in the schema tree by τ.	Fig. 3 (e)
6	eliminate	tuple node v	The subtree rooted by v is of the form $\langle A : \tau\rangle$, i.e., it has precisely one attribute	Replaces v by τ.	Fig. 3 (d)
7	delete [subtree]	tuple edge e	-	Deletes e and its subtree; reduces arity of tuple by one	Fig. 4 (f)
8	copy to (v)	tuple edge e	The label of e must not exist yet in the destination tuple node v. The path from $\text{nca}(v, e)$ to e must be $*$-free.	Adds e and its subtree to tuple node v.	Fig. 4 (c)
9	copy to (v)	set edge e	The types below e and destination set node v must be equal. The path from $\text{nca}(v, e)$ to e must contain precisely one $*$ (i.e., the label of e).	Does not modify schema tree.	
10	select (A, B)	set node v	The child of v must be a tuple of type $\langle A : \tau_A, B : \tau_B, \ldots\rangle$	Does not modify schema tree.	Fig. 4 (e)

Table 1: VCP operations.

3.2 Informal Semantics

In order to obtain a simple informal semantics for VCP, it is convenient to think of complex data values as trees themselves. There are two main differences between schema trees and data trees. In the former, $*$-nodes have precisely one child, while in the latter, $*$-nodes may have many (one for each of the members of the set). Moreover, leaf nodes of schema trees are either labeled "Dom", $\emptyset$, or $\langle\rangle$, while data trees have atomic values instead of "Dom" at the leaves. (See Figure 1 a or 2 a for an example of a schema tree and Figure 1 b resp. 2 b for a data tree compatible to the schema tree. In order to avoid confusion – but also to be economic with space – all data trees in this paper are turned by 90 degreed compared to schema trees.)

By the path between two nodes v, w (where v is an ancestor of w), we denote the sequence of edge labels through which w is reachable from v. Note that in a schema tree, for any node v, each path π uniquely identifies a node reachable from v via π. In other words, schema trees are deterministic trees. We can identify nodes of the schema tree with their paths from the root node (e.g., we can talk of "the node books.$*$.isbn" in Figure 1 (a)). This is generally not the case for data trees. A given path in the data tree may match a set of nodes, e.g., relative to the root node of Figure 1 (b), books.$*$.year matches "1988" and "2002".

In the following, we say that a VCP operation is *schema-local* to a subtree of the schema tree if all the visual steps necessary to specify the operation can be exclusively taken in the subtree.

There are three kinds of operations in VCP, (a) operations on nodes (1, 2, 3, 5, 6, and 10 of Table 1), (b) operations on edges (4, 7, and 8 of Table 1), and (c) copy-paste operations (8 and 9 of Table 1).

deletion on tuple edges, addition of constants, and tuple elimination.

By the *context node* of an operation o, denoted $\text{ctx}(o)$, we refer to

- the node of the schema tree that o is applied to if o is a node operation (a), with the exception of *set note elimination* where $\text{ctx}(o)$ is its parent (a $*$-node as well),
- the node that the edge emanates from which o is applied to if o is an edge operation (b), and
- the node $\text{nca}(w, e)$ for a copy-operation (c) that copies from edge e to node w.

Of course, o is schema-local w.r.t. the subtree of $\text{ctx}(o)$.

The *local semantics* function $L[\![o]\!]$ maps from a complex value to a complex value, or in other words, from a data tree to a data tree: $L[\![\text{select(A=B)}]\!](u)$, where u is a $*$-node of the data tree, removes all those members of set u (= children of u with their subtrees) that do not satisfy the selection condition $A = B$. $L[\![\text{eliminate}]\!](u)$ flattens the set of sets u, i.e., replaces the subtree rooted by u by $\bigcup u$. For all other operations o of types (a) and (b), $L[\![o]\!]$ on u is basically the same operation as described in Table 1 for the schema tree applied to node u (or an emanating edge) in the data tree.

The local semantics of copy-operations (c) is as follows. For a copy-paste operation o (type 8 or 9 in Table 1) from source edge e to destination node w, let π_{from} be the path from $\text{ctx}(o) = \text{nca}(w, e)$ to e and let π_{to} be the destination path from $\text{ctx}(o)$ to w. If π_{to} contains $*$-edges, we say that o is a bulk-copy operation. If o is on tuples (type 8), π_{from} consists only of tuple-nodes. $L[\![o]\!](u)$ is obtained from u by copying the source $u.\pi_{from}$ to each of the tuple nodes matching $u.\pi_{to}$. If o is on sets (type 9), π_{from} consists only of tuple-nodes and one set-node s (the node which e directly emanates from). $L[\![o]\!](u)$ is obtained from u by copying each of the members of the source-set reachable from u via path

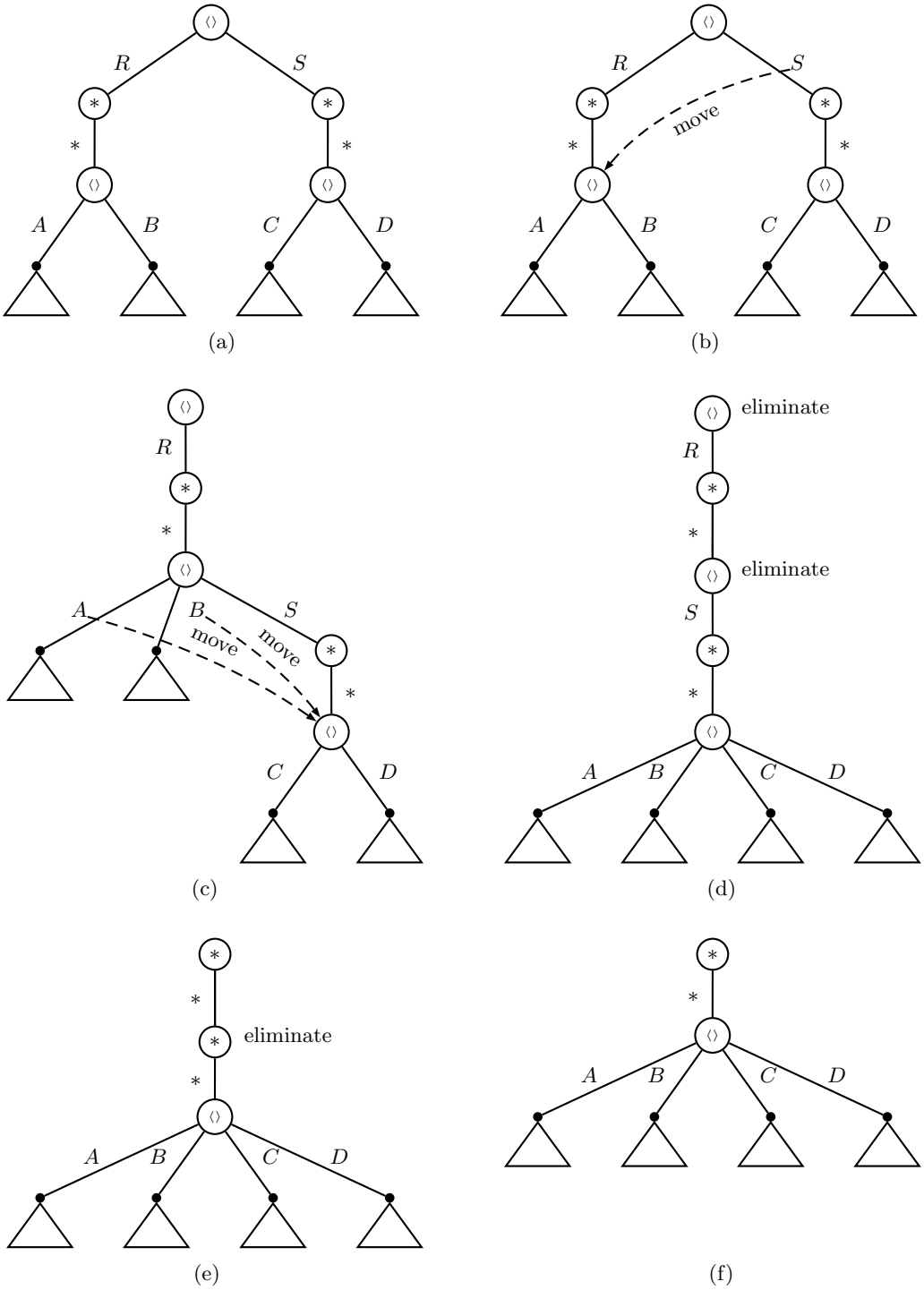


Figure 3: Simulating the cartesian product $R \times S$ of flat relational algebra in VCP.

π_{from} (excluding the final "$*$") to each of the destination sets matching $u.\pi_{to}$.

If the path π from the root of the schema tree to $ctx(o)$ contains $*$-nodes, we call o a *bulk operation*. Operation o is executed by replacing each node u reachable through π from the root of the data tree by $L[\![o]\!](u)$.

Example 3.1. Consider again the schema and the data tree of Figure 1 (a) and (b), respectively, The operation "insert tuple (A)" on the schema tree node books.$*$.year replaces the values of year attributes in tuples reachable through path books.$*$ in the data tree by a unary tuple node with the year as the value of attribute A. For an edge manipulation example, "rename to publyear" on the schema tree node books.$*$.year renames each of the "year" edges of tuple nodes reachable through path books.$*$ of the data tree to "publyear". □

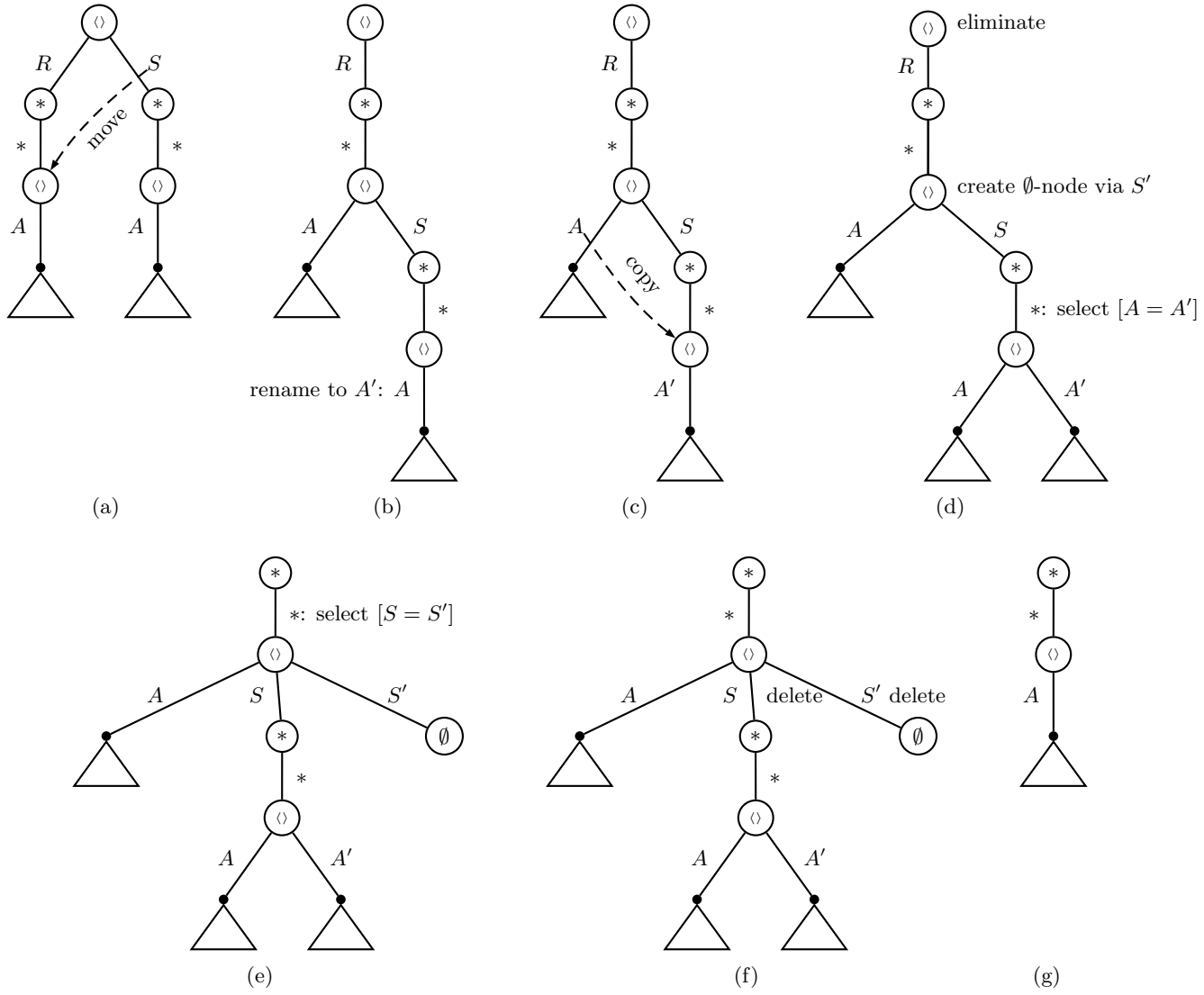


Figure 4: Modeling difference $R - S$ using selection in VCP.

3.3 Examples

Figure 3 shows how the cartesian product $R \times S$ for two binary relations $R(A, B)$, $S(C, D)$ can be encoded in VCP. Note that we employ two bulk copy operations (or, here, "move" to safe space) to pair the tuples of the two relations.

The operation "move" is obtained by first copying and subsequently deleting the source. It is employed here to save space but with considerable headache; a "bulk copy" operation is really much more intuitive than a "move to many places" operation.

In Figure 4, we show how difference $R - S$ for unary relations $R(A)$ and $S(A)$ can be encoded in VCP. The idea of the mapping is the same as in Example 2.2, but we have gone to the full length of not assuming a selection operation of the form $\sigma_{A=\emptyset}$. We use only a single form of selections, $\sigma_{A=B}$. This query may also serve as a template for proving that adding "-" to VCP does not extend its expressive power.

Of course, difference "-" could also be directly supported in a GUI based on VCP to allow to realize this query in a single step. Difference "−" may not have the clean visual metaphor of the other VCP operations, but it may be available as an option for expert users.

Figure 5 shows how to implement the nest operation of complex value algebra without powerset [1, 2] in VCP with selection. We encode $\text{nest}_{C=(B)}(R)$ on relation $R(AB)$.

It it important to assert that each VCP query step is assumed to consist of a single operation, even though we have sometimes resorted to annotating schema trees in Figure 3, 4, and 5 with several independent operations to safe space.

4. EQUIVALENCE OF VCP AND MONAD ALGEBRA

In this section, we characterize the expressive power of VCP by showing that it captures full monad algebra $\mathcal{M}_\cup[\sigma]$. We also show that VCP without selection captures precisely positive monad algebra and that VCP without the copy operation on sets and without selection coincides with $\mathcal{M}$. These expressiveness results not only show that VCP captures the "benchmark expressiveness" for complex-value

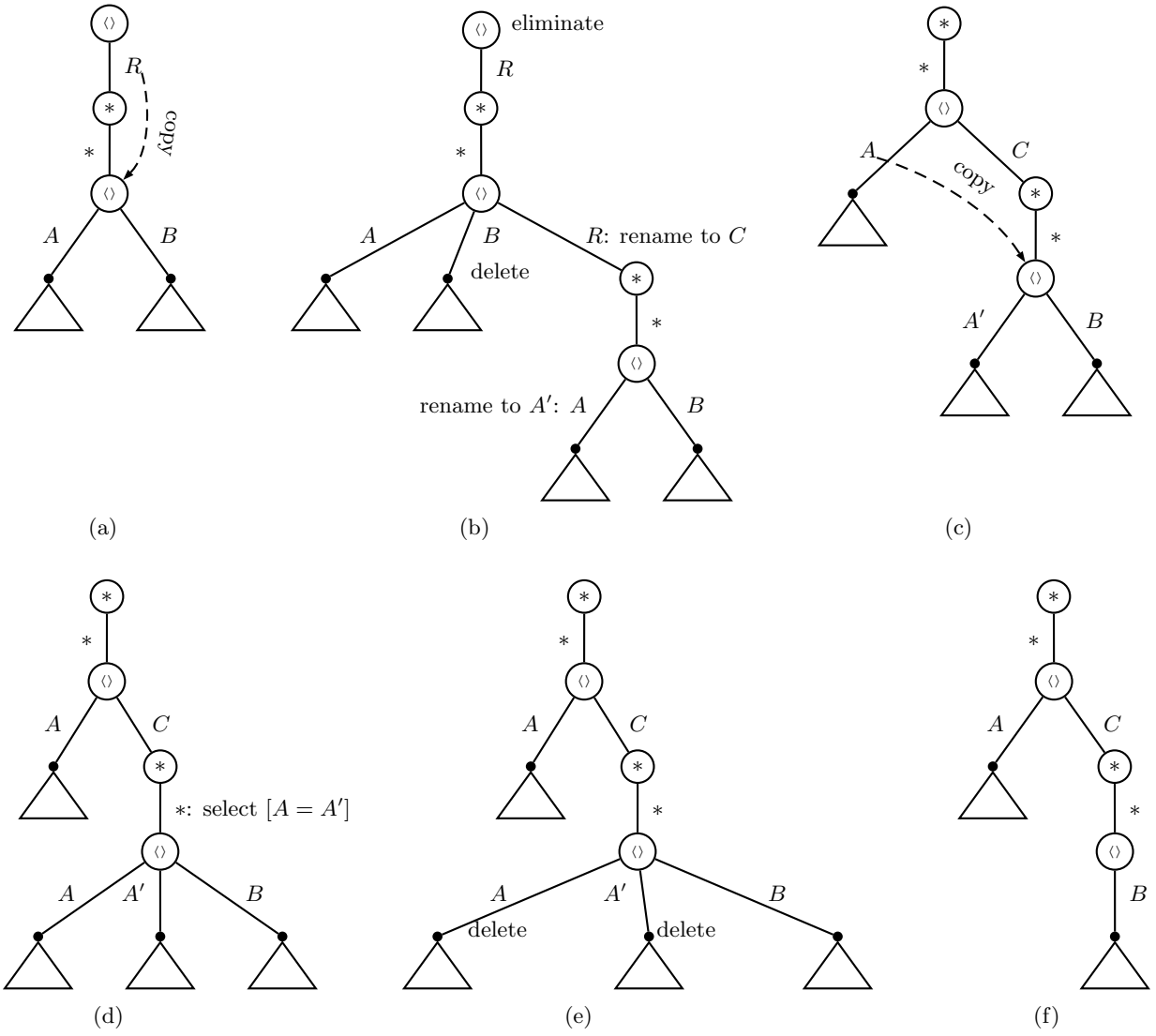


Figure 5: $\mathbf{Nest}_{C=(B)}(R)$ in VCP.

databases. The proofs of the direction $VCP \subseteq \mathcal{M}_{\cup}[\sigma]$ also provide us with a translation from VCP to full monad algebra that allows us to use existing query engines and results on query evaluation for the latter language. The proofs are straightforward and provide us with an alternative formal semantics for VCP (through the monad algebra framework).

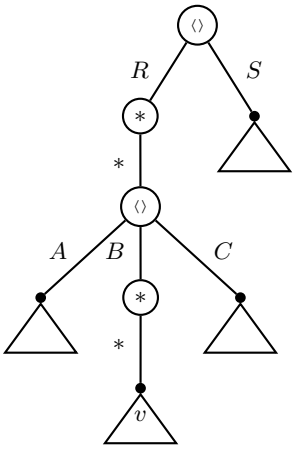
THEOREM 4.1. *VCP[wo 'select'] $\subseteq \mathcal{M}_{\cup}$.*

Proof Sketch: The proof is by induction. We define a function $\mathcal{M}[\![\ldots]\!]$ that maps any VCP query to an equivalent $\mathcal{M}$-expression.

- If operation o on schema $\langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle$ is schema-local to τ_1, then $\mathcal{M}[\![o]\!] = \langle A_1 : \pi_{A_1} \circ \mathcal{M}[\![o@A_1]\!], A_2 : \pi_{A_2}, \ldots, A_n : \pi_{A_n}\rangle$ where $o@A_1$ denotes o modified to apply to the A_1 branch of the schema tree (i.e., the subtree corresponding to τ_1).

  If operation o on schema $\tau = \{\tau'\}$ is schema-local to τ', then $\mathcal{M}[\![o]\!] = \text{map}(\mathcal{M}[\![o@*]\!])$ where $o@*$ denotes the operation o modified to apply to the subtree corresponding to τ'.

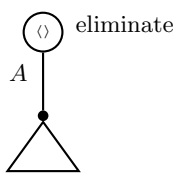
For example, if o is an operation schema-local to the subtree rooted by node v in the schema tree



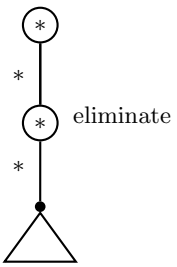
then

$$\mathcal{M}[\![o]\!] = \langle R : \pi_R \circ \mathrm{map}(\langle A : \pi_A, B : \pi_B \circ \mathrm{map}(\mathcal{M}[\![(((o@R)@*)@B)@*]\!]), C \circ \pi_C \rangle), S : \pi_S \rangle.$$

- The deletion of subtrees, the renaming of tuple-edges, and the addition of new constants to tuples is handled by tuple creation, constants, and projection.
- Tuple-node and set-node elimination is encoded using projection and "flatten", respectively:

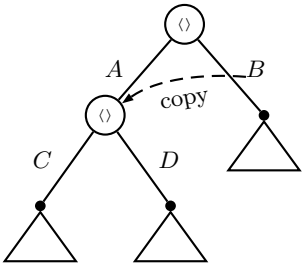


$\mathcal{M}[\![o]\!] = \pi_A$

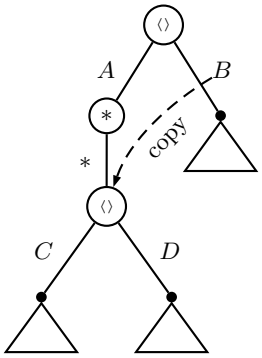


$\mathcal{M}[\![o]\!] = \mathrm{flatten}$

- A tuple-node with edge "A" is inserted by $\langle A : \mathrm{id} \rangle$.
- A set-node is inserted by "sing".
- Copy-paste expressions on tuples are mapped to $\mathcal{M}_\cup$ as shown for two important cases:



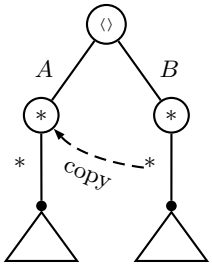
$$\mathcal{M}[\![o]\!] = \langle A : \langle B : \pi_B, C : \pi_A \circ \pi_C, D : \pi_A \circ \pi_D \rangle, B : \pi_B \rangle$$



$$\mathcal{M}[\![o]\!] = \langle A : \mathrm{pairwith}_1(\pi_A, \pi_B) \circ \mathrm{map}(\langle B : \pi_2, C : \pi_1 \circ \pi_C, D : \pi_1 \circ \pi_D \rangle), B : \pi_B \rangle$$

In general, the path from $\mathrm{nca}(v, e)$ to e can consist of of sequence of tuple edges, from which be can extract the source node by a sequence of projections. On the path from $\mathrm{nca}(v, e)$ to destination node v, there can be a sequence of tuple nodes (handled using tuple construction as shown in the first example above) and $*$-nodes, handled using "pairwith" as in the second example.

- Copy-paste expressions on sets are mapped to union. For example,



$$\mathcal{M}[\![o]\!] = \langle A : \pi_A \cup \pi_B, B : \pi_B \rangle$$

- A sequence of VCP operations $O_1, \ldots, On$ where each O_i corresponds to an $\mathcal{M}$-expression f_i simply corresponds to composition $((f_1 \circ f_2) \circ \ldots) \circ f_n$. □

The previous translation also yields

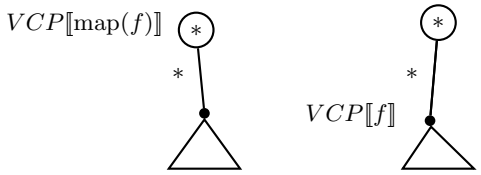
Corollary 4.2. *VCP[wo 'copy to set node', 'select'] $\subseteq \mathcal{M}$.*

For the other direction,

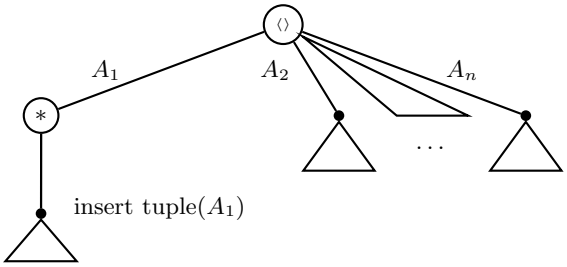
Theorem 4.3. *$\mathcal{M}_\cup \subseteq$ VCP[wo 'select'].*

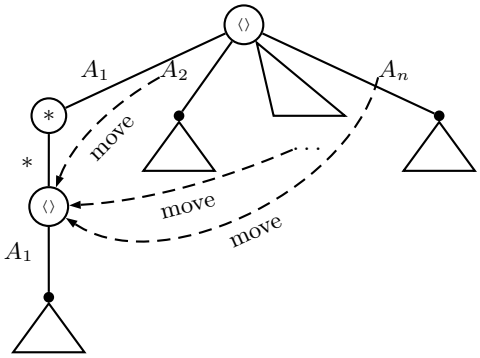
Proof Sketch: The proof employs an inductive definition of a function $VCP[\![\ldots]\!]$ that maps $\mathcal{M}_\cup$ expressions to VCP queries. Translating most operations is obvious, so we will just discuss the most interesting:

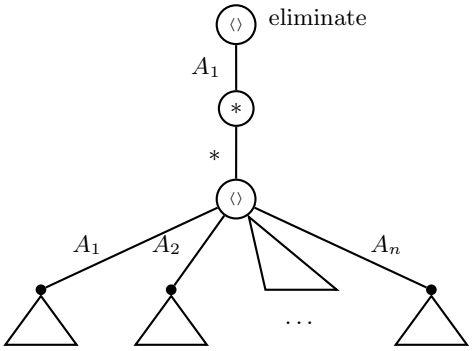
- map



- pairwith. The essential operation here is copy-paste. We implement $\mathrm{pairwith}_{A_1}$ in VCP as follows.







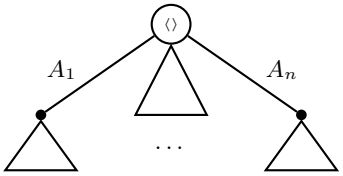
- tuple creation. We rewrite each $\mathcal{M}$-expression of the form $\langle A_1 : f_1, \ldots, A_n : f_n\rangle$ into

$$\langle A_1 : \mathrm{id}, \ldots, A_n : \mathrm{id}\rangle \circ \langle A_1 : \pi_{A_1} \circ f_1, \ldots, A_n : \pi_{A_n} \circ f_n\rangle.$$

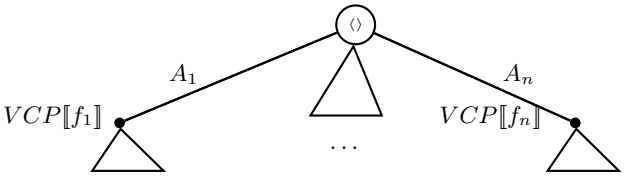
Now, $\langle A_1 : \mathrm{id}, \ldots, A_n : \mathrm{id}\rangle$ simply means to make a tuple of n copies of the input value which can be realized in VCP as shown in Figure 6 (for $n = 3$).

The second step, $\langle A_1 : \pi_{A_1} \circ f_1, \ldots, A_n : \pi_{A_n} \circ f_n\rangle$, just requires to push the f_i down into the A_i branches, for each $1 \leq i \leq n$. That is, we transform

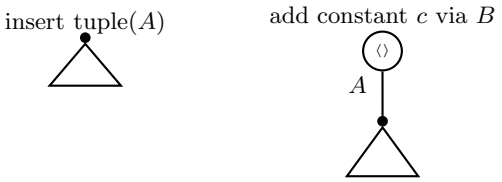
$VCP[\![\langle A_1 : \pi_{A_1} \circ f_1, \ldots, A_n : \pi_{A_n} \circ f_n\rangle]\!]$

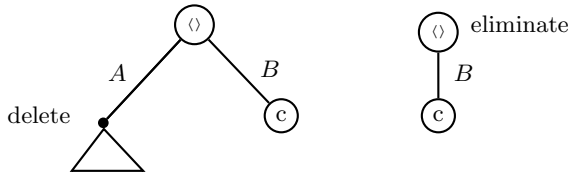


into



- union is implemented using the copy operation between sets.
- constants.





□

COROLLARY 4.4. *$\mathcal{M} \subseteq$ VCP[wo 'copy to set node', 'select'].*

The same selection operation is available in VCP and $\mathcal{M}_\cup[\sigma]$. From the previous results, we obtain

THEOREM 4.5. *$VCP = \mathcal{M}_\cup[\sigma]$.*

5. DISCUSSION AND CONCLUSIONS

In our presentation, we have implicitly assumed set-typed collections, but at least for positive monad algebra $\mathcal{M}_\cup$ and for VCP[wo selection], sets can be replaced by bags or lists and we immediately obtain that $\mathcal{M}_\cup = VCP$[wo selection] still holds. For the practical viewpoint, this implies that VCP can be used for visually specifying bag queries (as in SQL), but it also exhibits one oblique notion that users have to understand in order to successfully use the VCP language, namely that of collections. Under the set-interpretation, dragging a $*$-edge onto a $*$-node results in the addition of the members of the source sets to the destination set *with duplicate elimination*, while for bags, duplicates are not eliminated, and for lists, this operation means to append the source list to the destination list.

Regarding full monad algebra, there is unfortunately no agreed-upon notion of intersection or difference for bags and lists (several alternatives can be reasonably justified). Moreover, not all of these notions lead to the same expressive power when added to $\mathcal{M}_\cup$. For example, the probably most natural notion of difference on bags usually referred to as *monus* is known to yield the power of arithmetics, while this is not the case for intersection or selection [13].

Still, any of these differing "full" bag or list monad algebras can be simulated in VCP by just adding the desired operations (such as selection or difference) directly. Of course this is a pragmatic solution, but actually not more daring than what we have done in the set case earlier in the paper: positive monad algebra was realized using just insertion, renaming, deletion, and copying, while the move to the expressiveness of full monad algebra was made by adding a selection operation. The choice of which operations are made available in the language (in the case of sets, selection, intersection, and difference are interchangeable and even adding them all does not yield greater expressiveness) should depend on which operations appear natural given the design choices made for the GUI of the query editor.

One future area of research will be to provide a visual language for defining XML queries on the basis of VCP. Ordered, unranked XML trees can be viewed as nested lists, but to get a visual language for XML with the expressive power of (full) monad algebra on lists, also tuple-typed nodes are required. These do not exist in XML, but could be simulated using the XPath position() function; that is, the children of pseudo-tuple nodes in XML would only be accessed using paths of the form "child[position() = i]", yielding the i-the attribute of the pseudo-tuple.

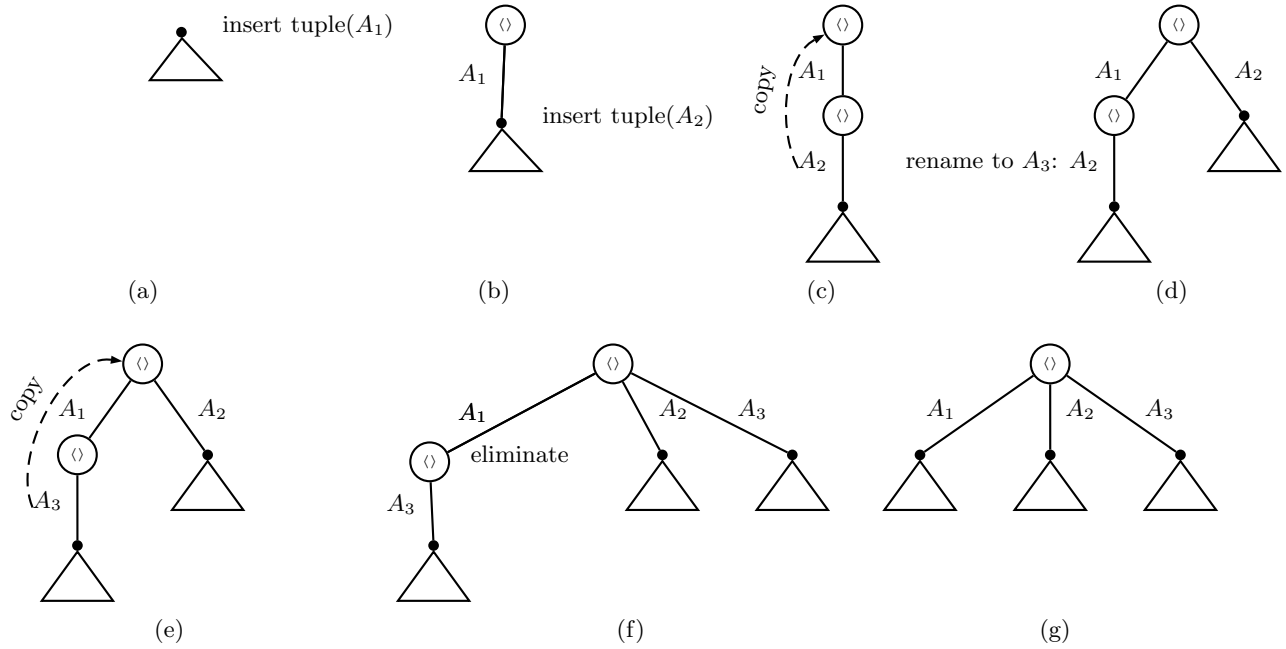


Figure 6: $VCP[\![\langle A_1 : \mathbf{id}, A_2 : \mathbf{id}, A_3 : \mathbf{id}\rangle]\!]$.

Schema information for XML is usually provided in the form of Document Type Definitions or XML Schemas, which correspond to possibly infinite schema trees (through recursion). However, this is not a problem in VCP or VCP extensions that correspond to XQueries that use no transitive axes. In VCP, each query only navigates into a schema tree up to a certain depth fixed with the query. Further work will be required for VCP to deal with navigation to the descendants of XML nodes.

VCP – in an extended form – seems to be a promising candidate for a bridging formalism between (relational) databases, file systems, and (XML or HTML) data trees. Such a bridge could lead to greatly enhanced usability of visual tools for Web site and Web service definition. This is one possible direction of future research.

6. REFERENCES

[1] S. Abiteboul and C. Beeri. "The Power of Languages for the Manipulation of Complex Values". *VLDB J.*, **4**(4):727–794, 1995.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] E. Augurusa, D. Braga, A. Campi, and S. Ceri. "Design and Implementation of a Graphical Interface to XQuery". In *Proc. SAC*, pages 1163–1167, Melbourne, Florida, 2003.

[4] R. Baumgartner, S. Flesca, and G. Gottlob. "Declarative Information Extraction, Web Crawling, and Recursive Wrapping with Lixto". In *Proc. LPNMR'01*, Vienna, Austria, 2001.

[5] R. Baumgartner, S. Flesca, and G. Gottlob. "Visual Web Information Extraction with Lixto". In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 119–128, Rome, Italy, 2001.

[6] S. Berger, F. Bry, S. Schaffert, and C. Wieser. "Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data". In *Proc. VLDB (Demo Track)*, pages 1053–1056, 2003.

[7] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. "Principles of Programming with Complex Objects and Collection Types". *Theor. Comput. Sci.*, **149**(1):3–48, 1995.

[8] S. Comai, E. Damiani, and P. Fraternali. "Computing Graphical Queries over XML Data". *ACM TOIS*, **19**(4):371–430, 2003.

[9] M. P. Consens and A. O. Mendelzon. "GraphLog: a Visual Formalism for Real Life Recursion". In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, 1990.

[10] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. "G+: Recursive Queries Without Recursion". In *Expert Database Conf.*, pages 645–666, 1988.

[11] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". *Journal of the ACM*, **51**(1):74–113, 2004.

[12] G. Jaeschke and H.-J. Schek. "Remarks on the Algebra of Non First Normal Form Relations". In *Proc. PODS'82*, pages 124–138, 1982.

[13] L. Libkin and L. Wong. "Query Languages for Bags and Aggregate Functions". *J. Comput. Syst. Sci.*, **55**(2):241–272, 1997.

[14] Y. Papakonstantinou, M. Petropoulos, and V. Vassalos. "QURSED: Querying and Reporting Semistructured Data". In *Proc. SIGMOD Conference*, pages 192–203, 2002.

[15] J. Paredaens, P. Peelman, and L. Tanca. "G-Log: A Graph-Based Query Language". *IEEE Trans. Knowl. Data Eng.*, **7**(3):436–453, 1995.

[16] J. Paredaens and D. Van Gucht. "Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions". In *Proc. PODS*, pages 29–38, 1988.

[17] V. Tannen, P. Buneman, and L. Wong. "Naturally Embedded Query Languages". In *Proc. ICDT*, pages 140–154, 1992.

[18] L. Wong. "Normal Forms and Conservative Extension Properties for Query Languages over Collection Types". *J. Comput. Syst. Sci.*, **52**(3):495–505, 1996.

[19] M. M. Zloof. "Query-by-Example: A Data Base Language". *IBM Systems Journal*, **16**(4):324–343, 1977.