**The VLDB Journal manuscript No.**
(will be inserted by the editor)

# UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion

**Peter Buneman**[1], **Mary Fernandez**[2], **Dan Suciu**[2]

[1] University of Pennsylvania
[2] AT&T Labs

Received: date / Revised version: date

**Abstract** This paper presents structural recursion as the basis of the syntax and semantics of query languages for semistructured data and XML. We describe a simple and powerful query language based on pattern matching and show that it can be expressed using structural recursion, which is introduced as a top-down, recursive function, similar to the way XSL is defined on XML trees. On cyclic data, structural recursion can be defined in two equivalent ways: as a recursive function which evaluates the data top-down and remembers all its calls to avoid infinite loops, or as a bulk evaluation which processes the entire data in parallel using only traditional relational algebra operators. The latter makes it possible for optimization techniques in relational queries to be applied to structural recursion. We show that the composition of two structural recursion queries can be expressed as a single such query, and this is used as the basis of an optimization method for mediator systems. Several other formal properties are established: structural recursion can be expressed in first order logic extended with transitive closure; its data complexity is PTIME; and over relational data it is a conservative extension of the relational calculus. The underlying data model is based on value equality, formally defined with bisimulation. Structural recursion is shown to be invariant with respect to value equality.

## 1 Introduction

The recent interest in semistructured data was sparked a few years ago with the development of the Object Exchange Model (OEM) [PGMW95], a data format that could be used to exchange arbitrary database structures between applications. What was novel about OEM was its ability to accommodate irregular (semistructured) data – data with no assigned schema. Because of this it found immediate application in data integration and in modeling other forms of irregular data that are to be found in scientific data formats and on the Web [AQM$^+$97,BDHS96,FFK$^+$98].

In a parallel development XML has emerged as a standard format for data exchange on the Web [Con98]. However the roots of XML are not in data models but in document markup languages in which the markup is used to convey the structure rather than the layout of a document. Despite the disparate origins and different superficial appearance of OEM and XML, the two approaches are remarkably close in their underlying data model; see, for example [ABS99]. The abstract model is extremely simple: it is nothing more than a labeled graph.

A new data model immediately invites the development of a query language for that model. Relational query languages are often taken as a yardstick for what is desirable in a query language: an underlying optimizable algebra, a simple surface syntax such as SQL or OQL, the ability to define views and compose them with queries, etc.[1]. Not surprisingly a number of languages have been developed or adapted for semistructured

[1] For a more extensive list of desiderata for an XML query language, see Maier's paper in [QL98]

data and XML: Graphlog [CM90], MSL [PAGM96], UnQL [BDHS96], Lorel [AQM+97], Strudel [FFK+98], XSL [Cla99b,Cla99a], XQL [Rob99], and XML-QL [DFF+99]. There are two aspects to these languages that set them apart from standard database query languages. They are all capable to some extent of treating schema information such as attribute names as data, and – because they need to perform searches over a graph – they all incorporate some form of *recursion*. The first aspect is not unusual: it is clearly available in relational database systems that present schema information in a table, and it is also available in complex object languages such as F-logic[KW93] also present the schema in a way that it can be queried. It is the second aspect that is challenging. It has yet to be established that recursive query optimization techniques developed for deductive databases work well in the context of semistructured data. Allowing arbitrary recursive programs makes optimization difficult and probably allows nonterminating programs. Some existing systems adopt *ad hoc* rules to bound recursive searches.

In UnQL (Unstructured data Query Language) [BDHS96] a new approach to querying semistructured data was introduced based on *structural recursion*. The idea is to limit the form of a recursive program by tying it closely to the recursive structure of the data. UnQL was one of the first query languages for semistructured data, and it met many of the criteria (an optimizable algebra, a simple query language, compositionality etc.) for a database query language. In addition to exploiting structural recursion, UnQL introduced the idea of using *patterns* and *templates* in a simple surface syntax. It also used a somewhat different model to the OEM. Roughly speaking, UnQL's model is, like the relational model, *value based* and this allows analogous optimizations to those in relational databases. The purpose of this paper is to explain in detail the various components of UnQL: its model, its query language, its internal algebra and its optimizations. We believe that these are of quite general use in the design and implementation of query languages for semistructured data.

Recent activity in the World Wide Web Consortium has brought structural recursion into the center of discussion for XML query languages. XML is best explained to a database person as a syntax for semistructured data. The only XML languages adopted so far by the industry are XSL [Cla99a,Cla99b] and its relative XQL [Rob99].

The original goal of XSL was to express transformations from XML to HTML[2], called *stylesheets*. The idea is for XML authors to associate a stylesheet with an XML document. A browser would execute the stylesheet and obtain an HTML representation of the document that can be displayed. XSL however evolved into a more general XML to XML transformation language. Being the first XML language to be shipped in commercial products, it is likely to be used for a while as a substitute for an XML query language, and it is likely to influence future XML languages. As a query language, XSL is nowhere close to any declarative query language a database person might be familiar with. It consists of a collection of rules, each with a pattern and a template. The input XML tree is processed top-down, by matching each pattern to the current node. If a match occurs, the template in that rule is evaluated, possibly triggering the rules to be applied recursively, on the node's children. Thus XSL's semantics is best understood in terms of recursive functions, rather than declarative semantics. As we show in this paper, this follows the same programming paradigm as structural recursion. We believe that structural recursion is in the position to bridge the gap between the document and the database communities working on a XML query language. At the same time, the optimization and evaluation techniques we show here for structural recursion also apply to XSL.

The use of patterns and templates in semistructured query languages was also pioneered in UnQL, and these are also important in the context of XML. Both XSL and XML-QL (a query language specifically designed for XML) have patterns and templates, those in XML-QL following UnQL's style quite closely.

UnQL's value-based data model is of general interest. In relational databases relations are treated as equivalent if they are observably equivalent. That is, they cannot be distinguished by any query. For relations queried with relational algebra, this means that the relations are *sets*, i.e. the order and multiplicity of the tuples in a relation is unimportant. For graph-like databases queried with UnQL, the relevant notion is *bisimulation*. Compared with graph isomorphism, which is the appropriate notion of observable equality for OEM, bisimulation has certain advantages. For example one can efficiently check the existence of a bisimulation, hence decide whether two graphs are observably equal. Also, duplicate elimination can be efficiently performed and, moreover, it can be performed at any place in the query, without affecting its semantics.

[2] HTML 4.0 is a subset of XML.

Again, interest in bisimulation for semistructured data goes beyond UnQL. A model based on bisimulation is adopted in [DGM98] to describe the relationship between semistructured data and description logics. More relevant to query languages is the fact that bisimulations are useful in building indexes. All languages for semistructured data (including XSL) incorporate some form of regular path expression, to evaluate them efficiently, researchers have investigated *path indexes*, i.e. data structures summarizing all paths in the data. The first example of such a structure was the "data guide" [GW97]. Assuming the data to be a non-deterministic automaton, the data guide is its deterministic powerset automaton. Data guides work well for certain data instances, especially trees, but do not scale well on highly connected instances. It has been shown however that bisimulation can successfully replace the powerset construct for the purpose of index construction. T-indexes [MS99], for example, scale better than data guides on cyclic data.

This paper is a complete description of UnQL's data model, query language, optimization techniques, and formal properties. Some of these were announced previously in [BDS95,BDHS96], but most details and proofs were never published before. Many techniques and properties transcend UnQL's scope and are applicable in other contexts too: for example the *conservativity result* (announced in [BDHS96] and proved here) was already used to prove similar properties for StruQL "by reduction" to UnQL [FFLS97]. To put these techniques to work, we show here how the optimizations for structural recursion can be used in a mediator-based data integration system to do query rewriting. Finally, the paper reports, for the first time, on UnQL's implementation.

Specifically, the paper makes the following contributions:

- It describes *structural recursion* as a query language for trees and explains its relationship to XML.
- It describes a declarative syntax for a subset of the language; the subset uses a `select-where` syntax with patterns and templates; a translation from this syntax into structural recursion is given.
- It describes an original semistructured data model based on labeled graphs. The model is value-based, as opposed to object-based, and value equivalence is given by a certain notion of bisimulation. A large class of results can be obtained for graphs (under bisimulation) simply by establishing them for finite trees.
- It shows how structural recursion can be lifted from trees to graphs, and defines a calculus for graphs, based on structural recursion.
- It describes two different evaluation techniques for structural recursion (top-down, and bulk), and proves that they are equivalent. Bulk evaluation can be implemented with traditional database operators, while top-down evaluation, when restricted to trees, is typical for XSL processors.
- It describes an optimization for structural recursion, and shows how it can be applied in mediator systems. The optimization relies on certain algebraic laws, whose validity is proved.
- It establishes and proves the following formal properties of the calculus: all queries are invariant under bisimulation, the calculus can be expressed in first order logic extended with transitive closure, and the calculus is a conservative extension of the relational calculus over relational data.
- it reports an implementation on UnQL in ML.

*Related Work* As a graph query language, Graphlog [CM90] and GOOD [GPVdBVG94,GPVdBVG90] deserve credit as possibly the first two languages for querying semistructured data. Graphlog is based on datalog and makes use of recursive datalog queries for graph queries. GOOD uses a graph model of data that conforms to a functional data model and has a query/transformation language that exploits pattern matching of graphs. If one removes the distinction between two kinds of edge labels in GOOD and forgets the schema, one arrives at a semistructured data model that is close to the model we describe in this paper. Several more recent query languages have been considered for semistructured data. Lorel [AQM+97] is the query language in Lore, a semistructured data management system. Lorel adapts OQL to query semistructured data, by extending it with regular path expressions and implicit coercions. Recently Lorel has been extended to query XML and its implementation is described in [MW99]. MSL [PAGM96] is a logic-based language for semistructured data, extending datalog to semistructured data and is used in the Tsimmis data integration project [PGMW95]. It was the first language to use Skolem functions in semistructured data. StruQL is the query language in the Strudel project, a Web Site Management System [FFLS97,FFK+98]. It was especially designed to allow complex graphs to be easily constructed declaratively: for that it uses Skolem functions and a block structure, a unique feature in StruQL. XML-QL [DFF+99] is a query language for XML, combining UnQL's patterns and templates with StruQL's Skolem functions and block structure.

Structural recursion has been known to the functional programming community for many years and appears as the "reduce" or "fold" operators of languages such as Lisp, ML and Haskell. It was first proposed for database languages in FAD [BBKV87] and Machiavelli [OBB89]. The semantics of recursive programming constructs for collection types was first described by Tannen and Subrahmanyam in [BTS91]. Phil Wadler first observed the connection between comprehension for collection types (lists, bags, sets) in [Wad92] and structural recursion. A query language built around comprehension is described in [BLS+94], and the properties of such languages are described in [BTBN91]. Structural recursion as it is described in this paper extends those principles to trees and graphs.

Skolem functions where first discussed in the context of databases by Maier [Mai86]. They were later used in [AK89], in F-logic [KW93] and in ILOG [HY90].

Courcelle [Cou90] considers hypergraphs with *sources*, i.e. distinguished nodes in the hypergraph. He defines six hypergraph operators and proves that every hypergraph can be constructed using these operators. This is related to our data model which consists of a graph with distinguished input nodes and output nodes, on which we define nine constructors. Unlike Courcelle's operators however, ours are directed: they distinguish between the input and the output nodes.

Structural recursion is related to top-down tree transducers [RS97], but it is more powerful than these because it can express joins and cartesian products which are not expressed by top-down transducers. Top-down transducers are not closed under composition, in other words they do not enjoy an optimization similar to Theorem 4. The reason for that is the top-down transducers can express non-terminating queries (a form of side-effect), which structural recursion cannot.

Bisimulation was considered in process algebra [Mil89]. An efficient algorithm for computing a bisimulation is described by Paige and Tarjan in [PT87]: the best known algorithm for computing a simulation is in [HHK95]. Bisimulation was applied to indexes for semistructured data in [MS99].

*Organization* The paper is organized as follows. Sec. 2 introduces the semistructured data model and structural recursion restricted to trees. Sec. 3 describes three applications of the structural recursion paradigm: querying ordered trees, querying and transforming XML, and the usage in mediator systems. In Sec. 4 the model is extended to arbitrary graphs, and a calculus for graphs, UnCAL, is presented in Sec. 5. Some formal properties and optimization techniques are established in Sec. 6 and 7. Two evaluation strategies are presented in 8, and an implementation is reported in 9. We conclude in Sec. 10.

## 2 Semistructured Data and Structural Recursion

We start by describing a simple tree model for semistructured data and show how structural recursion works on trees. We also introduce the query language UnQL, which can traverse data to arbitrary depth with regular path patterns, and describe its semantics on trees. Restricting the data to trees makes structural recursion and UnQL queries much easier to define and understand. Of course, real data is often cyclic: we will show in later sections how the operations extend naturally to arbitrary graphs.

### *2.1 Semistructured Data Modeled as Trees*

Fig. 1 illustrates a typical example of semistructured data, adapted from `http://www.odci.gov/cia/publications/factbook/index.html`, which contains information about countries, e.g., geography, people, government, etc[3]. The data can be viewed as a tree in which nodes correspond to objects and leaves to atomic values. We use a simple syntax for a textual representation of semistructured data; our example in text form appears in Fig. 2.

Atomic values include quoted strings such as `"Ireland"`, numbers such as `70280`, and possibly other basic types. Structured values (the internal nodes of the tree) are denoted by $\{l_1 : e_1, \ldots, l_n : e_n\}$ in which the $l_1 \ldots l_n$ are *labels* such as `country`, `name`, etc., and the $e_1, \ldots, e_n$ are atomic values or other structured values.

[3] We apologize to the Central Intelligence Agency for the liberties we have taken in this excerpt from that most useful database.

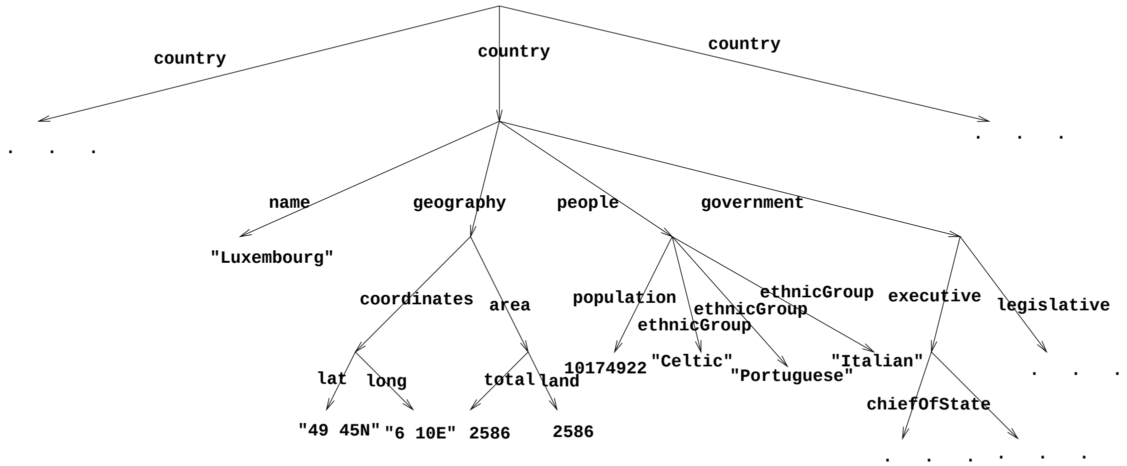


**Fig. 1** An example of a semistructured data instance.

```
FactBook:
 { country: { name: "Ireland",
              geography: { coordinates: {long: "53 00N", lat: "8 00W"},
                           area: { total: 70280, land: 68890: water: 1390}},
              people: {population: 3619480,
                       ethnicGroup: "Celtic",
                       ethnicGroup: "English"},
              government: {executive: {chiefOfState: "McAleese",
                                       headOfGovernment: {name: "Ahern"
                                                          function: "prime minister"}},
                           legislative: {...}}},
   country: { name: "Luxembourg",
              geography: { coordinates: {long: "49 45N", lat: "6 10E"},
                           area: { total: 2586, land: 2586}},
              people: {population: 425017,
                       ethnicGroup: "Celtic",
                       ethnicGroup: "Portuguese",
                       ethnicGroup: "Italian"},
              government: {executive: {chiefOfState: {name: "Jean",
                                                      function:"Grand Duke"},
                                       headOfGovernment: {name: "Juncker",
                                                          function:"prime minister"}},
                           legislative: {...}}},
    country: { name: "Belgium",
              geography: { coordinates: {long: "50 50N", lat: "4 00E"},
                           area: { total: 30510, land: 30230, water: 280}},
              people: {population: 10174922,
                       ethnicGroup: "Fleming",
                       ethnicGroup: "Walloon"},
              government: {executive: {chiefOfState: {name: "Albert II",
                                                      function:"King"},
                                       headOfGovernment: {name: "Dehaene",
                                                          function:"prime minister"}},
                           legislative: {...}}},
   ...
 }
```

**Fig. 2** Textual representation of the semistructured data in Fig. 1

This is different from a syntax for tuples, because labels may be repeated; duplicate labels are used to represent sets (as in the example above), without having to introduce a special set notation. Semistructured data is often called "schema-less" or "self-describing", because data attributes and data values are intermingled, which permits representation of irregular data. For example, repeated attributes (e.g. `ethnicGroup`) and missing attributes (e.g. `water`) are permitted, and attributes may have different types in different objects (`chiefOfState`). More importantly, the structure of data may evolve over time, e.g., `ethnicGroup` could occur at level 3 as well as level 4, and the data would still be valid. To summarize, the syntax for trees in UnQL is given by:

$$t ::= a \mid \{l: \ t, \ldots, l: \ t\}$$

where $a$ ranges over the atomic values and $l$ ranges over labels.

As another example, the same syntax can describe regular data, such as a relation:

```
{ student: { id: 123, name: "L. Simpson", age: 19 },
  student: { id: 345, name: "T. Quail",   age: 22 },
  student: { id: 789, name: "E. Vader",   age: 32 } }
```

in which we use a repeated label, `student`, to represent a set of subtrees.

The OEM [PGMW95] model defines semistructured data as a labeled graph, much like that in Fig. 1. The fundamental distinction between the UnQL and OEM models is that in UnQL $\{l_1 : \ t_1 \ldots, l_n : \ t_n\}$ denotes a *set* of label/tree pairs, not an object. By contrast OEM associates a separate object identifier with each node in the tree. Thus the UnQL model, like the relational model, is "value based". For example {`a: 1, a: 1`} and {`a: 1`} mean the same in UnQL but are different in OEM.

Since non-atomic values are sets of label/value pairs, they can be built from basic constructors for sets: the empty set {}, the singleton set, $\{l : \ v\}$ and the union of two sets $s_1 \cup s_2$. The notation $\{l_1 : t_1, \ldots, l_n : \ t_n\}$ is shorthand for $\{l_1 : \ t_1\} \cup \ldots \cup \{l_n : \ t_n\}$. For example, {`a: 5, b:` {`c: 8, d: 9, a: 7`}`, e: 4`} is shorthand for {`a: 5`} ∪ {`b:` ({`c: 8`} ∪ {`d: 9`} ∪ {`a: 7`})} ∪ {`e: 4`}. To summarize, there are four *constructors for trees*:

$$t ::= t \cup t \mid \{l : t\} \mid \{\} \mid a$$

where $a$ ranges over atomic values and $l$ over labels. Obviously, the operation $t_1 \cup t_2$ only makes sense when both $t_1$ and $t_2$ are sets, i.e. not atomic values.

### *2.2 Structural Recursion on Trees*

Recursion is the usual programming idiom for tree-like data. We introduce a restricted form of recursion, *structural recursion*, that is declarative and permits optimizations expected of a database query language. The key idea of structural recursion is that the form of the program follows the structure of the data. The restrictions, which are syntactic, ensure that the recursion always terminates. We introduce structural recursion by example, then give a formal definition. Structural recursion is only one operator of a query language, and may be combined with other operators.

Assume we want to retrieve all ethnic groups in our database. With structural recursion we write this query as a recursive function:

```
fun f1(T1 U T2) = f1(T1) U f1(T2)
  | f1({L:T})   = if L = ethnicGroup then {result: T} else f1(T)
  | f1({})      = {}
  | f1(V)       = {}
```

Uppercase symbols are variables. The left-hand sides of each equation are *patterns*. Note how the first three patterns follow the constructors for sets. When this function is applied to a value, that value is matched against each pattern in turn. If the set is constructed by a union operation, we bind the variables `T1`, `T2` to its two components and evaluate the right-hand side. If the set is a singleton label/value pair, bind the variables `L` and `T` to the label and value respectively. For an empty set, no variables are bound. The last line has a pattern consisting of the variable `V`. This is a "catch-all" clause, since the pattern `V` matches anything; in particular it matches atomic values. Each of the constructors for semistructured data is therefore matched by at least one pattern. When a value is matched by more than one pattern, we follow the convention of many functional programming languages and take the first matching pattern. Thus the order of the equations is significant.

Taken together these patterns define a function. Our syntax for pattern matching is borrowed from ML from which we also borrow the keyword `fun`, as in `fun f1 = ...` for defining the function `f1`.

The result of this query, applied to the database in Figure 2 is

```
{ result: "Celtic",  result: "English", result: "Portuguese",
  result: "Italian", result: "Fleming", result: "Walloon", ...}
```

Duplicates are eliminated because we have a set-based semantics.

Note that recursion is used twice in the definition of `f1`. First, it is used on the *horizontal* structure of a value in

```
f1(T1 U T2) = f1(T1) U f1(T2)
```

Second it is used on the *vertical* structure of the tree in `f1({L:T})  = if ... else f1(T)`. Unlike traditional (relational and object-oriented) query languages, query languages for semistructured data can operate in the vertical dimension, i.e., search the tree to an arbitrary depth.

If we allow arbitrary expressions on the right-hand side of clauses in structural recursion it is easy to construct programs that do not terminate or are nondeterministic (see [BTS91]). We therefore impose additional restrictions:

1. The right-hand side of the clause with the union pattern should always have the form `f(T1 U T2) = f(T1) U f(T2)`.
2. The only recursive calls in the right hand side of a singleton clause, `f({L:T})`, are for the argument `T`. Moreover, the results of the recursive calls can only be used in constructors, and not passed as arguments to other functions or predicates. In particular recursive calls of the form `f({a:{b:T}})` or `g(f(T))` are not allowed.
3. The result of `f({})` should always be `{}`.

Restrictions 1 and 3 mean that lines 1 and 3 in the program above are always the same, therefore we may omit them. These two restrictions deal with determinism: no matter how we decompose a set into a union of smaller sets, including empty ones, we get the same result. Restriction 2 guarantees that every recursive call will be on some argument strictly smaller than the input — hence the function terminates. As we will show, all three constraints guarantee that structural recursion is deterministic and always terminates, even on graphs with cycles.

Also, as a further syntactic abbreviation, if line 4 is missing, the empty set is returned by default. Our example `f1` above now reduces to

```
sfun f1({L:T}) = if L=ethnicGroup then {result: T} else f1(T)
```

where `sfun` indicates that we are defining a function with these defaults. We may also use constants in patterns and use them in place of conditionals:

```
sfun f1({ethnicGroup:T}) =  {result: T}
   | f1({L:T})           =  f1(T)
```

This is the notation we use in this paper.

*Examples of structural recursion* Structural recursion allows us to express complex transformations on a data structure. As a first simple example, structural recursion allows us to copy a tree:

```
sfun f2({L:T}) = {L:f2(T)}
   | f2(V) = V
```

This example returns an isomorphic copy of an input tree, in which all labels are `a`:

```
sfun f3({L:T}) = {a:f3(T)}
   | f3(V) = V
```

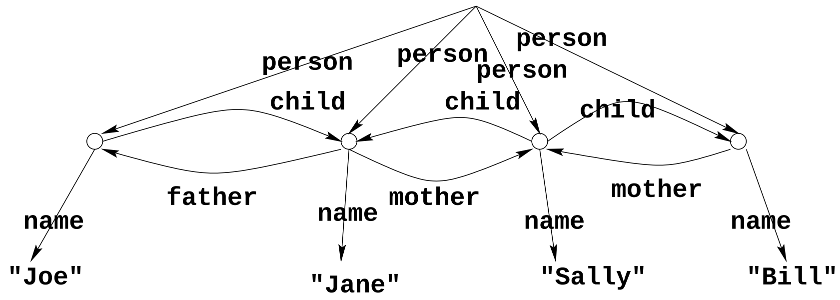


**Fig. 3** A data graph with cycles

The function below can double the children of each node giving a result that is apparently exponential in the size of the input.:

```
sfun f4({L:T}) = {a:f4(T)} U {b:f4(T)}
   | f4(V)     = V
```

For example, on the tree `{a:{b:{c:1}}}` the function returns

```
{a:{a:{a:1, b:1}, b:{a:1, b:1}},b:{a:{a:1, b:1}, b:{a:1, b:1}}}
```

We will show that such power does not result in expensive computation: an encoding of the result as a directed acyclic graph guarantees that any structural recursion function can always be computed efficiently (in PTIME).

In practice, it is useful to write multiple recursive functions that may call each other. In Fig. 2, we construct a view that is an isomorphic copy of the data, but in which all numbers directly or indirectly reachable from an `area` edge are converted from $km^2$ to $mi^2$. This is achieved with a recursive function `f5` that calls a recursive function `g5`.

```
sfun f5({area:T}) = {area:g5(T)}
   | f5({L:T})    = {L:f5(T)}
   | f5(V)        = V

sfun g5(V)     = if isInt(V) then 0.3861 * V else V
   | g5({L:T}) = {L:g5(T)}
```

`f5(t)` starts processing a tree `t` from the root, and copies it until it reaches `area`: here it calls `g5` on the subtree, which copies the subtree and converts all numbers in that tree. On branches where `f5` does not encounter `area`, it simply leaves the numbers and all other atomic values unchanged.

Mutually recursive functions are also possible, as shown by the even/odd example in Sec. 5.1.1. In general, any number of mutually recursive functions $f_1, f_2, \ldots, f_k$ can be defined simultaneously. In such definitions, restrictions 1 and 3 remain the same, hence, $f_i(t_1 \cup t_2) = f_i(t_1) \cup f_i(t_2)$ and $f_i(\{\}) = \{\}$, for every function $f_i$. Restriction 2 is relaxed to allow in the right hand side of the clause $f_i(\{l : t\})$ any of the recursive calls $f_1(T), f_2(T), \ldots, f_k(T)$ (or several of them).

*Cyclic structures* We digress with a brief discussion of why restrictions on recursion are needed. (A complete explanation is deferred to later sections.) Consider the cyclic structure shown in figure 3.

Here is a query to extract all the names that occur in the figure:

```
fun f6(T1 U T2) = f6(T1) U f6(T2)
  | f6({L: T})  = if L = name then {answer: T} else f(T)
  | f6({})  = {}
  | f6(V) = {}
```

If we view this definition as a program, then it will "loop infinitely". Alternatively, we can view it as a set of equations for which we must find an instance of the function `f6` that satisfies both the left-hand side and right-hand sides of each clause. One possible solution, in fact the minimal one, is

```
f6(r)= {answer: "Joe", answer: "Jane", answer: "Sally", answer: "Bill"}.
```

Note that more complicated expressions in the right-hand side of the first clause could mean that there is no finite solution to these equations. Suppose that the union in `f6(T1) U f6(T2)` were replaced by a list append operation. Any solution to these equations now contains an "infinite" list.

The ability to deal with cyclic structures in a clean and principled manner is one advantage of the UnQL model. In Lorel, cycles in a query are arbitrarily terminated when the query revisits a node it has already visited. In XSL, as we shall see, certain queries do indeed cause an infinite loop, even on acyclic data.

### *2.3 A Query Language*

In addition to structural recursion, UnQL has a simple `select ...where ...` surface syntax with pattern matching:

```
query Q1 :=
   select {result: E}
   where  {country: {name: "France",  people: {ethnicGroup: E}}} in db
```

This query finds all the ethnic groups in France, but unlike the function `f1`, it assumes that the label `ethnicGroup` will occur at a certain position in the database. The pattern

```
{country: {name: "France", people: {ethnicGroup: E}}}
```

can be viewed as a tree. It is matched in all possible ways (as a subtree) to the tree `db`. Each matching binds the variable `E` to some node in the database. For each such binding, the result `{result: E}` is constructed and the union of all such results is returned.

In general, patterns have a syntax similar to trees:

$$Pat ::= \{PE_1 : Pat_1, ..., PE_n : Pat_n\}$$
$$Pat ::= Var \mid Const$$
$$PE ::= Label \mid LabelVar \mid RPP$$

Tree variables, *Var*, are bound to nodes in the tree, and denote the subtree dominated by that node. *Const* denotes an atomic constant. $PE_1, \ldots, PE_n$ are path patterns: each can be a label, a regular path pattern (*RPP*), or a label variable (*LabelVar*). The example above contains labels (`name, people`). Regular path patterns are described further in Sec. 2.4; an example is `people.ethnicGroup`. Finally, label variables are variables that are bound to label constants, for example,

```
select { result: L }  where { country: { L: X } } in db
```

The result consists of all labels used in `country`. Regular path patterns are described in detail later.

*Conditions and Multiple Patterns* The `where` clause may include conditions. For example,

```
query Q2 :=
   select { result: N }
   where  { country: { name: N }, people: { population: P } } in db,
          P > 50000000
```

The condition `P > 50000000` evaluates to true only if `P` is bound to an atomic integer value greater than 50000000. Comparison operators e.g. `X` = `Y`, and predefined or user defined unary predicates, e.g., `isString(N)`, are supported. Predicates always return false if an argument is bound to a non-atomic value. Therefore, the condition `X` = `Y` tests equality of atomic values, not of trees. For efficiency, UnQL has no primitive to check tree equality, although it does provide the `isEmpty(X)` predicate, which tests whether `X` is bound to a leaf node, i.e., an atomic value or {}.

Multiple patterns in the same `where` clause are permitted:

```
query Q2' :=
   select { result: {name: N, people: X }}
   where  { country: { name: N }, people: X } in db,
          { population: P } in X,
          P > 50000000
```

The query returns both the country name `N`, and all information about its people, in `X`. Each tree variable in the `where` clause is bound to a node in the tree. The predefined variable `db` is always bound to the root. When a tree variable such as `X` occurs in the `select` clause the subtree it is bound to is substituted into the result.

*Joins* Multiple patterns and variable equality are used to express joins. The following query is a self-join, which returns all countries that have some ethnic group in common with France:

```
query Q3 :=
   select {result: C}
   where  { country: {name: "France", people.ethnicGroup: E}} in db,
          { country: {name: C, people.ethnicGroup: F}} in db,
          E = F
```

*Nested Queries* We can use nested queries in the `select` clause, to group results or to express outer joins. The following query finds all ethnic groups and associates the set of countries in which that group occurs:

```
query Q4 :=
   select { result: ( { ethnicGroup: E}
                      U
                      (select { country: C }
                       where  { country: {name: C, people.ethnicGroup: E}} in db)
                    )
          }
   where  { country.people.ethnicGroup: E} in db
```

Note that `E` occurs in both the inner and outer queries, which is an implicit join. The result contains:

```
{ result: { ethnic: "Celtic", country: "Ireland", country: "Luxembourg", ...}
  result: { ethnic: "Portuguese", country: "Luxembourg", country: "Portugal", ...}
  ... }
```

Nested queries are also used to retrieve optional fields, a form of outer join. The following query retrieves all land areas and, where available, the water area:

```
query Q4' :=
   select { result: { country: C, landarea: L,
                      (select { waterarea : W}
                       where { water: W} in X)}
          }
   where  { country: { name: C, geography.area: X}} in db, { land: L} in X
```

*General Syntax and Semantics* The general syntax for UnQL queries is:

$$
\begin{aligned}
\mathit{Query} &::= \mathsf{select}\ \mathit{Template}\ \mathsf{where}\ BC_1, \ldots, BC_n \\
\mathit{Template} &::= \{\mathit{TE}_1 : \mathit{Template}_1, \ldots, \mathit{TE}_m : \mathit{Template}_m\} \mid \mathit{Var} \mid (\mathit{Query}) \\
\mathit{TE} &::= \mathit{Label} \mid \mathit{LabelVar} \\
\mathit{BC} &::= \mathit{BindCond} \mid \mathit{BoolCond} \\
\mathit{BindCond} &::= \mathit{Pat}\ \mathsf{in}\ \mathit{Var}
\end{aligned}
$$

A *BoolCond* can be any boolean combination of equality or inequality conditions and externally defined predicates applied to variables.

Each variable in an UnQL query has a *scope*. Given a query select *Template* where $BC_1, \ldots, BC_n$, let $BC_i = Pat_i$ in $Var_i$ be the first of of the $BC_1, \ldots, BC_n$ in whose pattern the variable $X$ occurs. The scope of $X$ is defined to be $Pat_i$, $BC_{i+1}, \ldots, BC_n$ and *Template*. All occurrences of $X$ in within this scope are taken as the same variable. For example in `select ...where {a: X} in db, {b: X} in db` the two occurrences of `X` denote the same variable, and the effect of this query is to bind `X` only to those values that occur both under an `a` and a `b` edge. In contrast, in `select {c: (select X where {a:X} in db), b: (select X where {b: X} in db)} where ...` the two `X`'s have disjoint scopes and they are different variables. There exists a predefined variable `db`, bound to the database of interest, whose scope is the entire query. Finally, a bind condition of the form *Pat* in *Var* may occur only within the scope of *Var*. For the remainder of this section we assume all queries to be written such that each variable $X$ occurs at most once in a pattern. For that we replace each subsequent occurrence of $X$ with a fresh variable $X'$, and add the condition $X = X'$. For example the query above with two occurrences of the same variable $X$ will be transformed into `select ...where {a: X} in db, {b: X'} in db, X = X'`.

We define below the result of a query on some tree database, by defining the semantics for one query at a time (i.e. sub-queries are defined independently). We say that a tree $t$ is *included* in a tree $t'$ if either both are the same atomic value, or if $t = \{l_1 : t_1, \ldots, l_m : t_m\}$, $t' = \{l'_1 : t'_1, \ldots, l'_n : t'_n\}$, and for each $i = 1, \ldots, m$ there exists $j = 1, \ldots, n$ s.t. $l_i = l'_j$ and $t_i$ is included in $t'_j$.

Let $Q$ be a query select *Template* where ..., and let $t$ be an input tree. Consider $V$ to be the set of all variables $X$ for which *Template* is in the scope of $X$. An *assignment*, $\theta$, is a mapping from $V$ to the nodes and the atomic values in $t$, such that the following three conditions hold. First, the distinguished variable *db* is mapped to the root node in $t$. Next, all *BindCond* and all *BoolCond* in the query are satisfied, as follows. A *BindCond* of the form *Pat* in *Var* is satisfied if $\theta(Var)$ is a node, and the tree $\theta(Pat)$ is included in the subtree of $t$ whose root is $\theta(Var)$. A boolean condition *BoolCond* is satisfied if all variables occurring in the condition are mapped to atomic values, and the boolean condition holds on those atomic values. Finally, we require that $\theta(Template)$ not be an atomic value (this can only happen when *Template* is a single variable *Var* and $\theta(Var)$ is an atomic value). For such an assignment $\theta$, we evaluate every subquery $Q'$ occurring in *Template*: in doing so we only consider assignments which extend $\theta$ to the new variables in $Q'$. Let $\theta(Template)$ denote the tree obtained by replacing each subquery $Q'$ with its value, and each variable *Var* with $\theta(Var)$. Now we can define the result of $Q$ on $t$. Let $\theta_1, \ldots, \theta_k$ be all possible assignments for the variables in $Q$. Then the semantics of the query $Q$ on the tree $t$ is defined to be $\theta_1(Template) \cup \ldots \cup \theta_k(Template)$.

### *2.4 Translating Queries into Structural Recursion*

We can show that pattern matching can be translated into structural recursion. To simplify presentation, we describe the case for queries without regular path patterns or sub-queries, using `Q1` above. In the first step, we transform the query into a query with only basic label/value patterns by repeatedly applying the following two rules to the query's `where` clause:

$$\text{where } \{PE_1 : Pat_1, ..., PE_n : Pat_n\} \text{ in } Var \rightarrow \text{where } \{PE_1 : Pat_1\} \text{ in } Var, ..., \{PE_n : Pat_n\} \text{ in } Var$$
$$\text{where } \{PE : Pat\} \text{ in } Var \rightarrow \text{where } \{PE : NewVar\} \text{ in } Var, Pat \text{ in } NewVar$$

In the second rule, *Pat* is a pattern that is neither a variable nor a constant, and *NewVar* is a fresh variable name. On `Q1`, the rules produce:

```
select { result: E }
where  { country: C } in db,
       { name: "France" } in C,
       { people: P } in C,
       { ethnicGroup: E } in P
```

Next, we describe nested functions in UnQL, which follow ML's syntax for introducing local names, as in:

```
let fun hypot(x,y) = sqrt(x*x + y*y)
in  hypot(hypot(3,4),12)
```

Using this notation we can define a query on a database `db` by:

```
let sfun f1({ethnicGroup:T}) =  {result: T}
       | f1({L:T})           =  f1(T)
in  f1(db)
```

where we have created a local scope in which the function name `f1` is defined so that we can apply it to `db`.

In the second translation step, we apply the following two rules repeatedly:

$$\mathsf{select}\ e\ \mathsf{where}\ (\{PE:T\}\ \mathsf{in}\ e', \mathit{rest}) \rightarrow \mathsf{let}\ \mathit{sfun}\ h(\{PE:T\}) = (\mathsf{select}\ e\ \mathsf{where}\ \mathit{rest})\ \mathsf{in}\ h(e')$$

$$\mathsf{select}\ e\ \mathsf{where}\ () \rightarrow e$$

In these rules, *rest* is a syntactic meta-variable which stands for the remaining clauses in the where component, and $h$ is a new function name that is different from any other name in scope.

For example, the query:

```
select {result: N} where {country: { name: N } } in db
```

is translated into:

```
let sfun h1({ country: C }) =
     let sfun h2({ name: N }) = { result: N }
     in h2(C)
in h1(db)
```

When applied to `Q1`, the result is:

```
let sfun h1({ country : C }) =
     let sfun h2({ Name: "France" }) =
          let sfun h3({ people: P }) =
               let h4 ({ ethnicGroup: E}) = { result: e }
               in h4(P)
          in h3(C)
     in h2(C)
in h1(db)
```

There are four functions defined by structural recursion, but none is recursive. Note that by our rules for pattern matching, `h2({Name : "France"})` returns the empty set when applied to a structure with no such label/value pair.

*Regular Path Patterns* So far our examples of `select ... where ...` queries have all used only horizontal structural recursion, i.e., when translated into structural recursion they produce non-recursive functions. It is also possible to express certain "deep" queries in the `select ... where ...` with regular path patterns. We have seen simple instances of regular path patterns. A regular path pattern is simply a regular expression on the alphabet of edge labels, including a "wild-card" label. For example the pattern `country : {geography : {coordinates : {long : X}}}` is abbreviated to `country.geography.coordinates.long : X`. In the following query,

```
select {vip: N}
where {country.government.executive._.name: N} in db
```

the underscore `_` is a "wild card". It matches any label, and is equivalent to introducing a new variable that is not used anywhere else in the query. The following two queries illustrate the use of alternation and optional labels:

```
select {vip: N}
where {country.government.executive.(chiefOfState|headOfGovernment).name: N} in db
```

```
select {vip: N}
where {country.government.executive.chiefOfState.name?: N} in db,
      isString(N)
```

In this example `name?` means zero or one occurrence of the label `name`, and `isString` is a predicate that tests whether its argument is a string. This query accounts for the irregular fashion in which names of chiefs of state are represented.

With a Kleene star we may match paths in the data of unbounded depth:

```
select { name: N }
where { _*.name: N } in db
```

In summary, regular path patterns are given by:

$$RPP ::= Label \mid \_ \mid (RPP.RPP) \mid (RPP \mid RPP) \mid RPP? \mid RPP*$$

As usual, we omit parentheses when the operator's precedence is clear.

Any regular path pattern can be translated into structural recursion, by expressing first the regular expression as an automaton (which may be non-deterministic) and associating a function with each state. For example, consider the regular expression is a.((b | c).d)∗.b? as used in the query `select T where {a.((b|c).d)*.b?: T} in db`. An equivalent (nondeterministic) automaton has four states and the following transitions:

$$s_1 \xrightarrow{a} s_2, s_2 \xrightarrow{b} s_3, s_2 \xrightarrow{c} s_3, s_2 \xrightarrow{b} s_4, s_3 \xrightarrow{d} s_2$$

The initial state is $s_1$, and $s_2, s_4$ are terminal states. The query is equivalent to the function `h1` below:

```
sfun h1({a:T}) = h2(T) U T
sfun h2({b:T}) = h3(T) U h4(T) U T
   | h2({c:T}) = h3(T)
sfun h3({d:T}) = h2(T) U T
sfun h4({L:T}) = {}
```

Each function corresponds to a state, and has one pattern for each symbol occurring on some transition from that state. Since $s_2, s_4$ are terminal states, whenever `h2(T)` or `h4(T)` occur in some right hand side their result is unioned with `T`. This translation generalizes easily to any regular expression $R$ whose language does not include $\varepsilon$ (the empty string). When $\varepsilon \in lang(R)$, then we first encode $R - \{\varepsilon\}$ as a structural recursive function $h'$, then we define $h(t) = h'(t) \cup t$.

*Summary of UnQL* In summary UnQL consists of the following operators:

- the tree constructors $\{\}, \{l : t\}, t_1 \cup t_2, \{l_1 : t_1, \ldots, l_n : t_n\}$
- functions defined by structural recursion
- queries defined with select − where and pattern matching, which can be translated into structural recursion.

### *2.5 Tree Equality*

We have stated that values are *sets* of label/value pairs, and our brief discussion of cyclic structures shows the importance of a set-based semantics interpretation of structural recursion on graphs.

One question is when are two structured values equal. For example, we interpret $\{1, 2, 3\}$, $\{1, 3, 2\}$, and $\{1, 3, 3, 2\}$ as the same set. Similarly, `{Name: "Joe", Tel: 1234, Tel: 3251}`, `{Tel: 1234, Name: "Joe", Tel: 3251}` and `{Name: "Joe", Tel: 1234, Tel: 1234, Tel: 3251}` should denote the same value, which means UnQL can discard duplicates. (In SQL, which allows both sets and bags, duplicates are explicitly removed with `unique`.)

The definition of equality is less obvious when we deal with deeper trees. For example

```
{a: {c: 3, b: 2}, a: {b: 2, c: 3}}
```

is the same value as

```
{a: {b: 2, c: 3}}
```

This prompts the general inductive definition of value equality for trees.

**Definition 1** Value-equality *for trees, $t \equiv t'$, is defined as follows. When $t$ and $t'$ are atomic values, $t \equiv t'$ if those values are the same. If $t = \{l_1 : t_1, \ldots, l_m : t_m\}$ and $t' = \{l'_1 : t'_1, \ldots, l'_n : t'_n\}$, and if*

- *for each $i \in \{1, \ldots, m\}$, there exists $j \in \{1, \ldots, n\}$ s.t. $l_i = l'_j$ and $t_i \equiv t'_j$.*
- *for each $j \in \{1, \ldots, n\}$, there exists $i \in \{1, \ldots, m\}$ s.t. $l_i = l'_j$ and $t_i \equiv t'_j$.*

*then $t \equiv t'$:*

It is this definition of tree equality that we will generalize to a definition of equality (bisimulation) on graphs in Sec. 4.

## 3 Applications of UnQL

Before extending UnQL to cyclic data, we discuss three applications of UnQL.

### *3.1 Querying Ordered Trees*

Throughout Sec. 2 we assumed that trees are sets: order and duplicates do not matter. However applications such as XML are based on an ordered tree model, one in which the children of any node have a prescribed order. The operations we have already discussed can be interpreted so that they make perfect sense on an ordered tree data model, and many of the optimizations developed in later sections also work on ordered trees. There are limitations on which techniques we can migrate from the unordered to the ordered model. Some optimizations do not work (for example idempotence $x \cup x = x$ fails), and we do not know how to extended structural recursion to ordered graphs. The extension to unordered graphs is discussed in Sec. 4.

To change the syntax for ordered trees, a tree is now a list:

$$t ::= a \mid [l : t, \ldots, l : t]$$

The tree constructors have the same form, but append @ replaces union:

$$t ::= t@t \mid [l : t] \mid [] \mid a$$

Structural recursion is defined as before, but the clause for `f(T1 U T2)` is replaced by `f(T1 @ T2)`. For example, the ethnic groups are retrieved by:

```
fun f1'(T1 @ T2) = f1'(T1) @ f1'(T2)
  | f1'([L : T]) = if L = ethnicGroup then [result: T] else f1'(T)
  | f1'([])      = []
  | f1'(V)       = []
```

### *3.2 Querying and Transforming XML*

*XML.* XML is a standard recently approved by the W3C as a data exchange format on the Web and can be viewed (with some approximations) as a syntax for semistructured data. For example, the data in Figure 2 is written in XML as:

```
<FactBook>
   <country>  <name> Ireland </name>
              <geography>  <coordinates>  <long> 53 00N </long> <lat> 8 00W </lat>
                           </coordinates>
                           <area> <total> 70280 </total> <land> 68890 </land>
                                  <water> 1390 </water>
                           </area>
              <people> ... </people>
              <government> ... </government>
   </country>
   <country> ... </country>
   ...
</FactBook>
```

*XSL.* There is no standard query language for XML. The only XML language supported by the W3C is XSL, an Extensible Stylesheet Language [Cla99a, Cla99b]. It is available in several commercial products, including Microsoft's Internet Explorer. While originally designed to express XML to HTML transformations (hence the name *stylesheet* language), XSL evolved into a general-purpose XML to XML transformation languages. To some extent it can serve as a limited query language. For example, consider the XSL program below:

```
<xsl:template> <xsl:apply-templates/> </xsl:template>
<xsl:template match="ethnicGroup">
       <result> <xsl:value-of/> </result>
</xsl:template>
```

This program consists of two template rules. Each rule has a match pattern, and a template: the match pattern in the first rule is empty (i.e., it can be applied anywhere), and in the second, the pattern is `ethnicGroup`. XSL proceeds recursively, from the root of the document and tries to match the current node with the patterns in the rules, starting from the last rule. Whenever a match occurs, the corresponding template is evaluated. When applied to the above XML data it returns:

```
<result> Celtic  </result>  <result> English </result>
<result> Celtic  </result>  <result> Portuguese </result>
<result> Italian </result>  <result> Fleming </result>
<result> Walloon </result>
```

Without going into the details of XSL, we remark that this XSL program is equivalent to the structural recursion function `f1` (modulo the different syntax and duplicate elimination). In fact, a large fragment of XSL can be rewritten in terms of structural recursion. For example:

```
<xsl:template>
       <a> <xsl:apply-templates/> </a>
       <b> <xsl:apply-templates/> </b>
</xsl:template>
<xsl:template match="c">
       <result> <xsl:value-of/> </result>
</xsl:template>
```

is equivalent to (note the reversed order of the cases):

```
sfun f({c:T}) = T
   | f({L:T}) = {a: f(T), b: f(T)}
```

Note that, like the function `f4` of Sec. 2.2, this can create an output that is exponentially larger than the input.

XSL templates can also express mutually recursive functions, through *execution modes*. This suggests a rather strong connection between XSL and structural recursion. However, there are important differences too. One is that XSL does not restrict its form of recursion and, hence, can express nonterminating programs even on trees. For example, consider the XSL program obtained by adding the following rule to the three rules above:

```
<xsl:template match="b">
        <xsl:apply-templates select="/"/>
</xsl:template>
```

When a `b` element is encountered, the templates are applied recursively to the root: the `select="/"` focuses the scope of `xsl:apply-templates` to the root of the tree. This results in a non-terminating computation on any input tree which has a `b` element.

A second difference is that XSL does not permit joins, which makes it inadequate for database applications.

*XML-QL.* A proposal for a query language for XML can be found in [DFF+99]. XML-QL incorporates ideas from UnQL and StruQL [FFLS97], such as pattern matching and Skolem functions. For example, query `Q1` above is written in XML-QL:

```
where <FactBook> <country> <name> France </name>
                           <people> <ethnicGroup> $E </ethnicGroup> </people>
                 </country>
      </FactBook>  in db
construct <result> $E </result>
```

XML-QL has joins, nested queries, and regular path patterns, like UnQL. However, XML-QL does not have a form of recursion, similar to structural recursion. While any function expressed with structural recursion can be expressed in XML-QL too, this is done using a complicated encoding with Skolem functions: such an encoding does not generalize to ordered data, like XML.

In summary UnQL, when applied to XML data, extends in a natural way both XSL and XML-QL.

### *3.3 UnQL Optimizations with Applications to Mediators*

Structural recursion admits surprisingly powerful optimizations. A full description is in Sec. 7. Here, we give a simple example.

Recall that for our example (Fig. 2), we can define a view (Sec. 2) in which all areas in $km^2$ are translated into $mi^2$ (functions `f5, g5`). We assume the view to be virtual.

Assume now that a user wants to find France's land area in $mi^2$. The combined query (view plus query) is:

```
let sfun f5({area:T}) = {area:g5(T)}
       | f5({L:T})    = {L:f5(T)}
       | f5(V)        = V

    sfun g5(V)     = if isInt(V) then 0.3861 * V else V
       | g5({L:T}) = {L:g5(T)}
in query Q :=
      select {result: A}
      where  { country: {name: "France", area.land: A}} in f5(db)
```

Of course, we don't want to materialize the entire view just to retrieve one value; we want to compose the two queries and obtain the simpler:

```
query Q' :=
      select {result: 0.3861 * A}
      where  { country: {name: "France", area.land: A}} in db
```

The optimization requires composing `f5, g5` with the query `Q` to obtain the query `Q'`. The more general problem is to compose two queries expressed with structural recursion, and obtain a single structural-recursion query (recall that `select-where` queries are non-recursive instances of structural recursion). Traditional optimization techniques for recursive queries (e.g. datalog) do not solve this problem. We will describe in Sec. 7 the general optimization technique for structural recursion.

This example is typical for mediator systems. In such systems, a mediator offers a virtual view over one or several data sources, e.g., an integrated view over several sources, or a restructuring of the data in one source. In such systems, most or all of the data from the sources is copied and restructured into the mediated view: when the sources are semistructured, this means that the mediator may need to traverse recursively the entire data tree. Thus, mediators are recursive, like the function `f5` above. User queries are in general `select-where` queries and only retrieve a tiny fraction of the data, like the query `Q` above. Optimizing the composed query (user query plus view definition) is critical for performance.

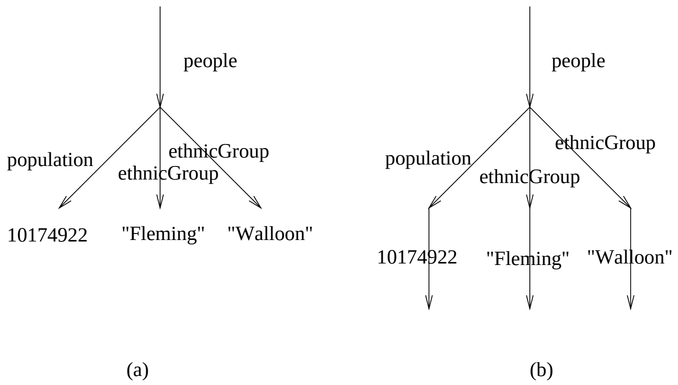


**Fig. 4** Representing data values on leaves (a) and on edges (b).

## 4 The Graph Data Model

The simple data model defined in Sec. 2 supports finite trees, but in practice, we need arbitrary graphs. In this section, we describe the graph data model and show that our language for trees extends to graphs as well.

First we need some notation. We assume an infinite set *Label* of symbols that label edges. Labels are typically strings, but for economy, we include all base types like integers, reals, etc., in *Label*. This eliminates the need for a second domain for leaf values: like labels, values can now be placed on edges, see Fig. 4. We allow an additional special symbol, $\varepsilon$, on the edges, and denote $Label_\varepsilon \stackrel{\text{def}}{=} Label \cup \{\varepsilon\}$. $Marker$ is an infinite set of symbols called *markers*. Each graph has certain nodes designated as inputs and certain nodes designated as outputs, and these nodes are labeled with markers. We need both the distinguished nodes and their markers in our model in order to describe the semantics of structural recursion on graphs. The top-down definition we used for trees does not work on graphs with cycles, of course. In our new semantics we split a graph into small cycle-free pieces, apply structural recursion independently on the pieces, then glue together the results. Markers are used for gluing: output nodes in some piece are glued with input nodes with the same marker in another piece. Markers are a unique feature of UnQL's data model, and somewhat related to object identifiers in object data models, like OEM. The difference is that not every node in a graph must have a marker, but only those designated as input or output nodes, and the same marker may be used in several graphs (this is useful, for example, when gluing nodes with the same marker). By contrast, in an object model every node must have a unique object identifier. Markers can also be used in queries, e.g., as entry points to the data. Markers are denoted $\&x, \&y, \&z, \ldots$. There is a distinguished marker $\& \in Marker$, typically used as the default input marker: an input node labeled & is called a *root*.

Let $\mathcal{X}, \mathcal{Y}$ be two finite sets of markers.

**Definition 2** *A labeled graph with input markers $\mathcal{X}$ and output markers $\mathcal{Y}$ is $g = (V, E, I, O)$, where $V$ is a (possible infinite) set of nodes, $E \subseteq V \times Label_\varepsilon \times V$ are the edges, $I \subseteq \mathcal{X} \times V$, and $O \subseteq V \times \mathcal{Y}$.*

When $(\&x, v) \in I$, then $v$ is called an *input node*; when $(v, \&y) \in O$, $v$ is called an *output node*. $I$ must be a one-to-one mapping from input markers to input nodes, and we denote with $I(\&x)$ the unique node associated with the marker $\&x$. It is possible for $I$ to be empty: in this case $\mathcal{X}$ is empty too, and the graph has no input nodes. Since we observe labeled graphs only through their inputs, all graphs with no input nodes ($\mathcal{X} = \emptyset$) are equivalent, and, hence, equivalent to the empty labeled graph $(\emptyset, \emptyset, \emptyset, \emptyset)$ (we will make this statement formal in Sec. 4.1). $O$ does not need to be one-to-one, and a marker $\&y \in \mathcal{Y}$ does not need to occur in $O$. In particular, $O$ may be empty. It also follows that every labeled graph with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$ is also a labeled graph with inputs $\mathcal{X}$ and outputs $\mathcal{Y}'$, for every $\mathcal{Y}' \supseteq \mathcal{Y}$. Finally, to complete the definition, input nodes must not have incoming edges, and output nodes must not have have outgoing edges, i.e., they are leaves. Note that a labeled graph may be infinite.

As our graph data model we use finite graphs, called *data graphs* or *databases*. We denote by $DB^{\mathcal{X}}_{\mathcal{Y}}$ the set of data graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. When $\mathcal{X} = \{\&\}$, then we abbreviate $DB^{\mathcal{X}}_{\mathcal{Y}}$ with $DB_{\mathcal{Y}}$. The set $DB_\emptyset$ is further abbreviated to $DB$.

*Trees* Trees, as defined in Sec. 2, are particular cases of data graphs with input & and no outputs. For example, the trees $\{a, b : \{c, d, d\}, b : \{c, d\}\}$ and $\{a, a, b : \{c, c, d\}\}$ are shown in Fig. 6 (a), and have 9 and 7 nodes respectively. The roots are labeled with the marker & (not shown in the figure) and there are no output markers. In this paper, we omit the input marker & when it is understood from the context. An example of a tree with output markers is $\{a, b : \&y, c : \{d\}\}$. The tree grammar in Sec. 2 is extended now to:

$$t ::= a \mid \&y \mid \{l : t, \ldots, l : t\}$$

*From finite trees to graphs* The guiding principle in UnQL is that concepts and properties are lifted from finite trees to graphs. To do this we show how a certain class of functions, defined on finite trees, naturally extend to work on graphs. To do this we first represent a graph as a certain type of infinite tree and then show how to lift our definitions for finite trees to such infinite trees. Representing the graph as an infinite tree is achieved through a process of *unfolding*.

*Unfoldings* Given some arbitrary data graph we unfold it, starting from each input node. We obtain a forest (or a tree, when there is only one input node), which may be infinite, hence our need to consider infinite graphs. For some data graph $d$, $\mathit{unfold}(d)$ denotes its unfolding. Fig. 6 (b) illustrates a data graph and its infinite unfolding. Note that the output marker $\&y$ occurs only once in $d$, but infinitely many times in $\mathit{unfold}(d)$. This is why we do not require $O$ to be one to one in Def. 2.

Formally, given $d \in DB_{\mathcal{Y}}^{\mathcal{X}}$, $d = (V, E, I, O)$, its unfolding is defined as $\mathit{unfold}(d) = (V', E', I', O')$ where:

- $V' = \{p \mid p$ is a path in $d$ starting at some $I(\&x)$, for $\&x \in \mathcal{X}\}$.
- $E' = \{(p, a, p') \mid p' = p.a$ where $p, p' \in V'$ and $a \in \mathit{Label}_\varepsilon\}$.
- $I' = \{(\&x, p) \mid p$ ends in $v$ and $(\&x, v) \in I\}$.
- $O' = \{(p, \&x) \mid p$ ends in $v$ and $(v, \&x) \in O\}$.

We remark that whenever $(\&x, p) \in I'$, $p$ is a path of length 0, i.e. consisting only of the node $I(\&x)$: this is because no edges enter any input node. In particular $I'$ is indeed a one-to-one mapping from $\mathcal{X}$ to the input nodes in $\mathit{unfold}(d)$.

Our plan is to show that $d$ is value equivalent with $\mathit{unfold}(d)$, for any data graph $d$. We will define value equivalence formally below. From this assumption it follows that any data graph $d$ is value equivalent to its *accessible* part (defined to be the set of nodes accessible by a path from some input node), because any data graph has the same unfolding as its accessible part.

We then need to show how operations and properties can be lifted from finite trees to data graphs via their unfoldings. For that we use the following fact, which will be proved in the Appendix. Let $f$ be a function from labeled graphs to labeled graphs. We call $f$ *compact* if, for any labeled graph $g$, $f(g)$ is determined by its actions on finite trees (formal definition is in Appendix). We call $f$ *unfolding* if $\mathit{unfold}(f(g)) = f(\mathit{unfold}(g))$ for any graph $g$. Then:

**Fact 1** *Let $f, g$ be two compact, unfolding functions. If $f(t) = g(t)$ for all finite forests $t$, then $f(d) = g(d)$ for all data graphs.*

*$\varepsilon$-Edges* One should think of $\varepsilon$-edges as of silent transitions in automata. They are necessary to define some of the operators on data graphs, including structural recursion. The meaning of $\varepsilon$ edges is captured by the following rule. Given a data graph $d = (V, E, I, O)$ and an edge $e = (u, \varepsilon, v) \in E$, then $d$ is value equivalent with the data graph $d_e$ obtained by deleting this $\varepsilon$-edge and adding the following:

- for each edge $(v, a, w)$ in $d$, a new edge $(u, a, w)$ is added in $d_e$;
- if $v$ is labeled with some output marker $\&y$ in $d$, then $u$ is labeled with that output marker in $d_e$.

For example Fig. 6 (c) illustrates that the data graph $d = \{a, \varepsilon : \{c, d\}, b\}$ is value equivalent to the data graph $d_e = \{a, c, d, b\}$. All nodes and edges that are not accessible from the root have been deleted from $d_e$.

### 4.1 Value Equivalence

We have discussed in Sec. 2 value equivalence for trees. The intuition was that two trees are value equivalent if they are equal when viewed as sets. Here we extend this notion to arbitrary graphs, and define a notion of value equivalence of two data graphs that satisfies three requirements. First, when both data graphs are trees, then value equivalence coincides with that for trees in Def. 1. Second, each data graph $d$ is value equivalent to *unfold*($d$). The latter may be infinite, hence it is not technically a data graph. For this reason our definition of value equivalence will be stated in terms of possibly infinite labeled graphs. Third, if $d_e$ is a data graph obtained by eliminating an $\varepsilon$-edge $e$ from $d$, as described above, then $d$ is value equivalent to $d_e$. Under this requirement, a path labeled $\varepsilon.a$ in $d$ is equivalent to an edge labeled $a$ hence, repeating the procedure, a path labeled $\varepsilon \ldots \varepsilon.a$, in notation $\varepsilon^*.a$, is equivalent to an edge labeled $a$. Similarly, if there exists an $\varepsilon^*$ path from some node $u$ to an output node with marker $\&y$, then we may consider $u$ to be labeled with $\&y$.

Such a definition for value equality can be obtained by extending a well known concept called graph bisimulation. We start by reviewing the traditional notion of graph bisimulation, and the related notion of graph simulation.

*Review of Graph Simulation and Bisimulation* Our review is based on [PT87,HHK95]. Given two graphs $G_1, G_2$, with $G_i = (V_i, E_i)$, $E_i \subseteq V_i \times V_i$, $i = 1, 2$, a *simulation* from $G_1$ to $G_2$ is a relation $S \subseteq E_1 \times E_2$ satisfying the following property: if $(x_1, x_2) \in S$ and $(x_1, y_1) \in E_1$, then there exists $y_2$ s.t. $(y_1, y_2) \in S$ and $(x_2, y_2) \in E_2$. One can check that the empty relation is always a simulation, that the union $S \cup S'$ of two simulations is a simulation, and that for any two graphs there exists a maximal simulation between them.

One way to understand a simulation is as a relaxation of the concept of a *graph morphism* from $G_1$ to $G_2$, which is a function $f : V_1 \to V_2$ s.t. for every edge $(x_1, y_1) \in E_1$, its image is also an edge: $(f(x_1), f(y_1)) \in E_2$. Namely a function $f : V_1 \to V_2$ is a graph morphism if and only if it is a simulation, when viewed as a binary relation. Computationally, however, simulations and graph morphisms behave differently. Deciding whether there exists a morphism from $G_1$ to $G_2$ is an NP-complete problem [GJ79], while the maximal simulation between them can be computed in PTIME, and quite efficiently [HHK95].

A *bisimulation* from $G_1$ to $G_2$ is a simulation $S$ for which $S^{-1} \stackrel{\text{def}}{=} \{(x_2, x_1) \mid (x_1, x_2) \in S\}$ is also a simulation. The empty relation is still a bisimulation, and there always exists a maximal bisimulation. The most efficient algorithm for finding a maximal bisimulation from $G_1$ to $G_2$ is described by Paige and Tarjan [PT87], and runs in $O(m \log n)$ time and $O(n)$ space, where $n = \mid V_1 \mid + \mid V_2 \mid$ and $m = \mid E_1 \mid + \mid E_2 \mid$.

Both definitions (of simulation and bisimulation), and all properties stated above carry over to the case when the graphs have labels on edges and/or nodes, by requiring that the relation $S$ preserves the labels. Specifically, when labels are on edges, then the definition of a simulation becomes: if $(x_1, x_2) \in S$ and $(x_1, a, y_1) \in E_1$, then there exists $y_2 \in V_2$ s.t. $(y_1, y_2) \in S$ and $(x_2, a, y_2) \in E_2$ (i.e. the edges $(x_1, a, y_1)$, $(x_2, a, y_2)$ have the same label $a$). As before, $S$ is a bisimulation if both $S, S^{-1}$ are simulations; this notion of bisimulation can be found in process algebras [Mil89]. When the labels are on nodes, then one requires that whenever $(x_1, x_2) \in S$, and $x_1$ is labeled with $a$, then $x_2$ be labeled with $a$ too. The problem of computing a bisimulation on labeled graphs reduces to that on unlabeled graphs. Paige and Tarjan's algorithm already handles the case when labels are placed on nodes. It does not mention labels explicitly, but instead, given a graph $G$ and an equivalence relation $P$, it computes the maximal bisimulation $S$ from $G$ to $G$ which is a subset of $P$; equivalently, the partition defined by $S$ is a refinement of that defined by $P$. Given two graphs $G_1, G_2$ with labeled nodes, one can compute the maximal bisimulation by taking their disjoint union $G_1 \cup G_2$, defining $P$ to be the equivalence relation on nodes given by the labels, applying Paige and Tarjan's algorithm, then retaining only those pairs $(x_1, x_2)$ in the bisimulation for which $x_1$ is in $G_1$ and $x_2$ is in $G_2$. It is easy to check that this procedure computes the maximal bisimulation between graphs with labeled nodes. When the graphs have labeled edges, then we need to reify them first. Namely we replace each edge $(u, a, v)$ labeled $a$ with a new node $\nu_{u,a,v}$ and two (unlabeled) edges: from $u$ to $\nu_{u,a,v}$ and from $\nu_{u,a,v}$ to $v$. The new node $\nu_{u,a,v}$ will be labeled with $a$. Applying this to $G_1$ and $G_2$ results in two new graphs with a total of $m + n$ nodes and $2m$ edges. These graphs have now labels on some of their nodes, and we can apply the previous procedure to find a maximal bisimulation in time $O(m \log(m + n))$ and space $O(m + n)$.

In the context of semistructured data we often deal with rooted graphs. A rooted graph has a distinguished node, called the root. A *rooted* bisimulation is a bisimulation $S$ between $G_1$ and $G_2$ s.t. $(r_1, r_2) \in S$, where

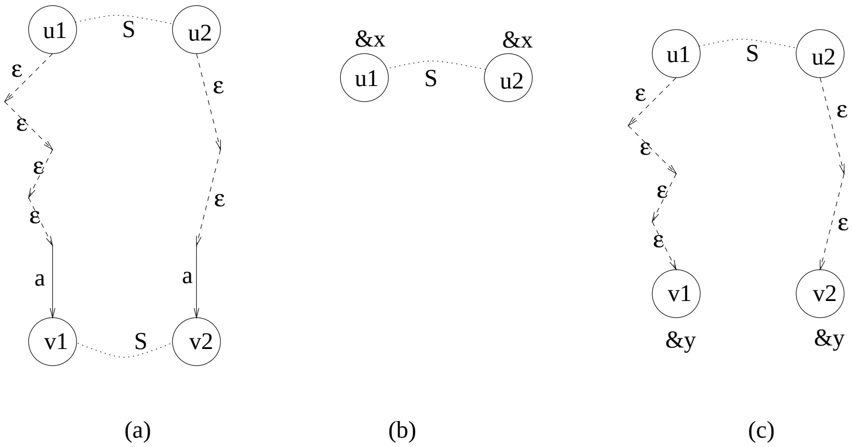


**Fig. 5** Illustration for the definition of extended simulations and bisimulations.

$r_1, r_2$ are the two roots. The empty relation is no longer a rooted bisimulation, and, in general, a rooted bisimulation may not exist. However it is still the case that if $S, S'$ are rooted bisimulations then $S \cup S'$ is a rooted bisimulation, and, hence, whenever some rooted bisimulation exists, then there exists a maximal one. It is easy to check if there exists some rooted bisimulation: compute the maximal bisimulation $S$, then check if $(r_1, r_2) \in S$.

*Value Equivalence* We extend now the notion of bisimulation to labeled graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, and define value equivalence in terms of this bisimulation. Recall that such labeled graphs may be infinite. We denote $(u, \varepsilon^*, v) \in E$ and $(u, \varepsilon^*.a, v) \in E$ whenever there exists a path from $u$ to $v$ labeled with $\varepsilon \dots \varepsilon$, or $\varepsilon \dots \varepsilon.a$ respectively.

**Definition 3** *Let $g_1 = (V_1, E_1, I_1, O_1), g_2 = (V_2, E_2, I_2, O_2)$ be two labeled graphs, both with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. An* extended simulation *from $g_1$ to $g_2$ is a relation $S \subseteq V_1 \times V_2$ such that:*

(a) *if $(u_1, u_2) \in S \wedge (u_1, \varepsilon^*.a, v_1) \in E_1$ with $a \neq \varepsilon$, then there exists a node $v_2$ s.t. $(u_2, \varepsilon^*.a, v_2) \in E_2$ and $(v_1, v_2) \in S$.*
(b) *if $(u_1, u_2) \in S \wedge (\&x, u_1) \in I_1$ then $(\&x, u_2) \in I_2$.*
(c) *if $(u_1, u_2) \in S \wedge (u_1, \varepsilon^*, v_1) \in E_1 \wedge (v_1, \&y) \in O_1$ then there exists a node $v_2$ s.t. $(u_2, \varepsilon^*, v_2) \in E_2 \wedge (v_2, \&y) \in O_2$.*
(d) *$(I_1(\&x), I_2(\&x)) \in S$, for every $\&x \in \mathcal{X}$.*

*An* extended bisimulation *from $g_1$ to $g_2$ is an extended simulation $S$ for which $S^{-1}$ is also an extended simulation.*

The first condition corresponds to the traditional definition of a simulation, but in our context in which a path labeled $\varepsilon^*.a$ is identified with an edge labeled $a$. The next two conditions treat both input and output markers as symbols labeling the nodes. Fig. 5 offers a graphical illustration of these three conditions. It says that if the top and left part hold, then so do the right and bottom. For example, in (a) if the top two nodes are in $S$ and the left path exists, then some path like that on the right exists and the bottom nodes are in $S$. For extended bisimulations the implication from right to left must hold too.

Finally, the last condition in Def. 3 requires the (bi)simulation to be rooted. As a consequence the empty relation is not an extended (bi)simulation any more.

We say that two labeled graphs $g_1$, $g_2$ are *value equivalent*, in notation $g_1 \equiv g_2$, if there exists an extended bisimulation from $g_1$ to $g_2$. It is easy to check that if two graphs are value equivalent then there exists a maximal extended bisimulation from $g_1$ to $g_2$.

Note that if $\mathcal{X} = \emptyset$, then condition (d) is true for any relation $S$: in this case the empty relation *is* a (bi)simulation, and all graphs with $\mathcal{X} = \emptyset$ are value equivalent, and equivalent to the labeled graph $(\emptyset, \emptyset, \emptyset, \emptyset)$, which we denote $()$.

Figure 6 shows some instances of equivalent graphs, and their extended bisimulations. The figure illustrates the three requirements for value equivalence spelled out at the beginning of this section:

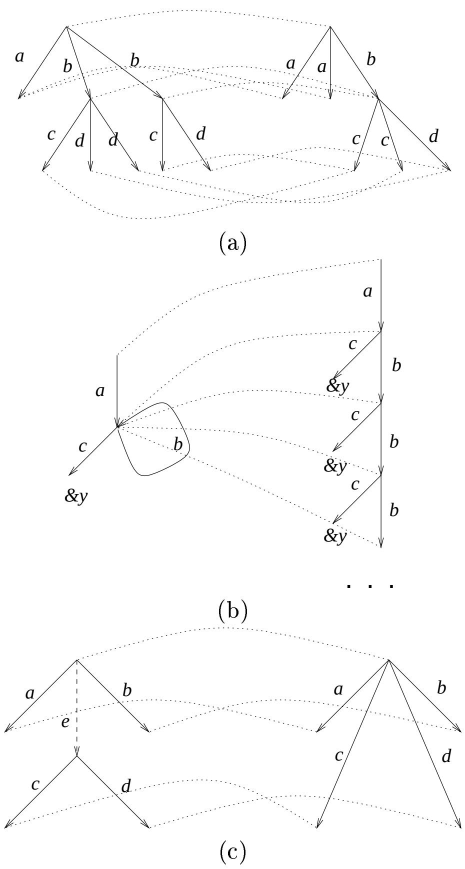


**Fig. 6** Examples of extended bisimulations between two graphs.

- any two finite trees $t_1, t_2$ are equivalent in the sense of Definition 1 iff they are equivalent in the sense that there exists an extended bisimulation from $t_1$ to $t_2$.
- any data graph $d$ is bisimilar to its infinite unfolding *unfold*$(d)$:

$$d \equiv \mathit{unfold}(d)$$

- any data graph $d$ is equivalent to $d_e$, where $d_e$ is obtained from $d$ by eliminating some $\varepsilon$ edge $e$, as explained earlier.

**Definition 4** *A function $f$ from labeled graphs to labeled graphs is called* bisimulation generic*, if, whenever $g \equiv g'$, then $f(g) \equiv f(g')$. A function $f$ with several arguments is called bisimulation generic if $g_1 \equiv g_1', g_2 \equiv g_2', \ldots$ implies $f(g_1, g_2, \ldots) \equiv f(g_1', g_2', \ldots)$.*

We end this sub-section explaining how value equivalence can be efficiently tested. The idea is to eliminate $\varepsilon$ edges first, then apply Paige and Tarjan's algorithm for labeled graphs. Let $d = (V, E, I, O)$ be a data graph. We construct a value equivalent data graph $d' = (V, E', I, O')$ without $\varepsilon$ edges. Its set of nodes is the same $(V)$, and so are its input nodes $(I)$. Edges $E'$ are defined as follows: $(u, a, w) \in E'$ for $a \neq \varepsilon$ if $(u, \varepsilon^*.a, v) \in E$. Outputs are defined as follows: $(u, \&y) \in O'$ if there exists $v$ s.t. $(u, \varepsilon^*, v) \in E$ and $(v, \&y) \in O'$. It is easily checked that $d \equiv d'$ (the identity relation is an extended bisimulation). The computation time for $d'$ is dominated by the time needed to compute the transitive closure of $\varepsilon$-edges (i.e. all $\varepsilon^*$ paths). Finally, given two data graphs $d_1, d_2$, one can check if they are value equivalent by computing $d_1', d_2'$ then checking whether there exists a rooted bisimulation from $d_1'$ to $d_2'$ with the procedure for labeled graphs.

### *4.2 Data Constructors*

There are three[4] constructors for trees: $t_1 \cup t_2$, $\{l : t\}$, and $\{\}$. As we saw, the tree constructors played a crucial role in the definition of structural recursion, and they are also useful operators in a query language like UnQL. Here we extend the tree constructors to a set of graph constructors: while some of the new constructors we introduce may find little use in the front syntax of a query language, they are useful both in the definition of structural recursion, and in a query optimizer, as internal operators.

Constructors for graphs and hypergraphs have been studied by Courcelle [Cou90]. He defines hypergraphs with *sources* and defines six constructors, proving that every hypergraph can be expressed using these constructors. Both the constructors and the result immediately carry over from hypergraphs to graphs. The sources in a graph are an ordered set of distinguished nodes, and resemble the inputs and outputs in our data graphs. Unfortunately the constructors do not include the tree constructors $t_1 \cup t_2$ and $\{a : t\}$ as primitives, but one has to express them as derived operations; only $\{\}$ is a primitive, denoted $\mathbf{1}$. This, together with the mismatch between the data models, necessitated different collection of constructors from those in [Cou90].

The data graph constructors are the following.

| | |
|---|---|
| $\{\}$ | empty tree |
| $\{l : d\}$ | singleton tree |
| $d_1 \cup d_2$ | union of two trees |
| | |
| $\&x := d$ | label the root node with some input marker |
| $\&y$ | data graph with one output marker |
| $()$ | empty data graph |
| $d_1 \oplus d_2$ | disjoint union |
| | |
| $d_1 @ d_2$ | append of two data graphs |
| $cycle(d)$ | data graph with cycles |

We explain the constructors here, their formal definition is in Fig. 7. There is a certain type discipline with respect to the input and output markers (similar to the *profile* in [Cou90]). Recall our convention by which a tree has the "default" input marker &.

The first three are the tree operators: $\{\}$, $\{l : d\}$, $d_1 \cup d_2$. All three constructors expect $d, d_1, d_2 \in DB_{\mathcal{Y}}$ and return results in $DB_{\mathcal{Y}}$. They are not polymorphic, hence should be written as $\{\}_{\mathcal{Y}}$, $\{l : d\}_{\mathcal{Y}}$, and $d_1 \cup_{\mathcal{Y}} d_2$ respectively; however we will always omit the subscript $\mathcal{Y}$ to avoid clutter. Note in Fig. 7 how union is defined with the aid of $\varepsilon$-edges. For example $\{a, b\} \cup \{c, d, e\} = \{\varepsilon : \{a, b\}, \varepsilon : \{c, d, e\}\}$ which is value equivalent to $\{a, b, c, d, e\}$.

The next four constructors allow us to create and add input and output markers: $\&x := d$ takes $d \in DB_{\mathcal{Y}}$ and relabels the root with the input marker $\&x$, hence the result is in $DB_{\mathcal{Y}}^{\{\&x\}}$. The constructor $\&y$ returns a tree with a single node labeled with the output marker $\&y$ and default input marker &: hence $\&y \in DB_{\mathcal{Y}}$, where $\&y \in \mathcal{Y}$. This is like $\{\}$, but now we have an output marker on the unique node. The asymmetry between the input marker constructor, $\&x := d$, and the output marker constructor, $\&y$, is due to the fact that we want to construct trees bottom-up: we start with a single node, possibly labeled with one output marker, but have to complete the entire tree before adding the input marker. The empty graph is denoted by $()$: it has no nodes, no edges, and is, up to value equivalence, the unique data graph in $DB_{\mathcal{Y}}^{\emptyset}$. Note the distinction between the empty graph $()$ and the empty tree $\{\}$ which contains a single node. The disjoint union $d_1 \oplus d_2$ requires $d_1$ and $d_2$ to have disjoint sets of input markers $\mathcal{X}_1, \mathcal{X}_2$, and the same set of output markers $\mathcal{Y}$; then $d_1 \oplus d_2 \in DB_{\mathcal{Y}}^{\mathcal{X}_1 \cup \mathcal{X}_2}$.

Finally, the last two constructors deal with the vertical structure of the data graphs. The append operator $d_1 @ d_2$ is defined when $d_1 \in DB_{\mathcal{Y}}^{\mathcal{X}}$ and $d_2 \in DB_{\mathcal{Z}}^{\mathcal{Y}}$, and results in a graph in $DB_{\mathcal{Z}}^{\mathcal{X}}$. Append is essentially performed by gluing each output node in $d_1$ with the input node in $d_2$ labeled with the same marker: formally, however, this is achieved by adding $\varepsilon$ edges, see Fig. 7. It corresponds to list concatenation, if linear trees are identified with lists: when $d_1 = \{a_1 : \{a_2 : \ldots \{a_n : \&z\} \ldots\}\}$ and $d_2 = (\&z := \{b_1 : \{b_2 : \ldots\}\})$, then

[4] There are four such constructors defined at the end of Sec. 2.1. The last one, $a$, is not needed any more, since we model now atomic values as labels on edges.

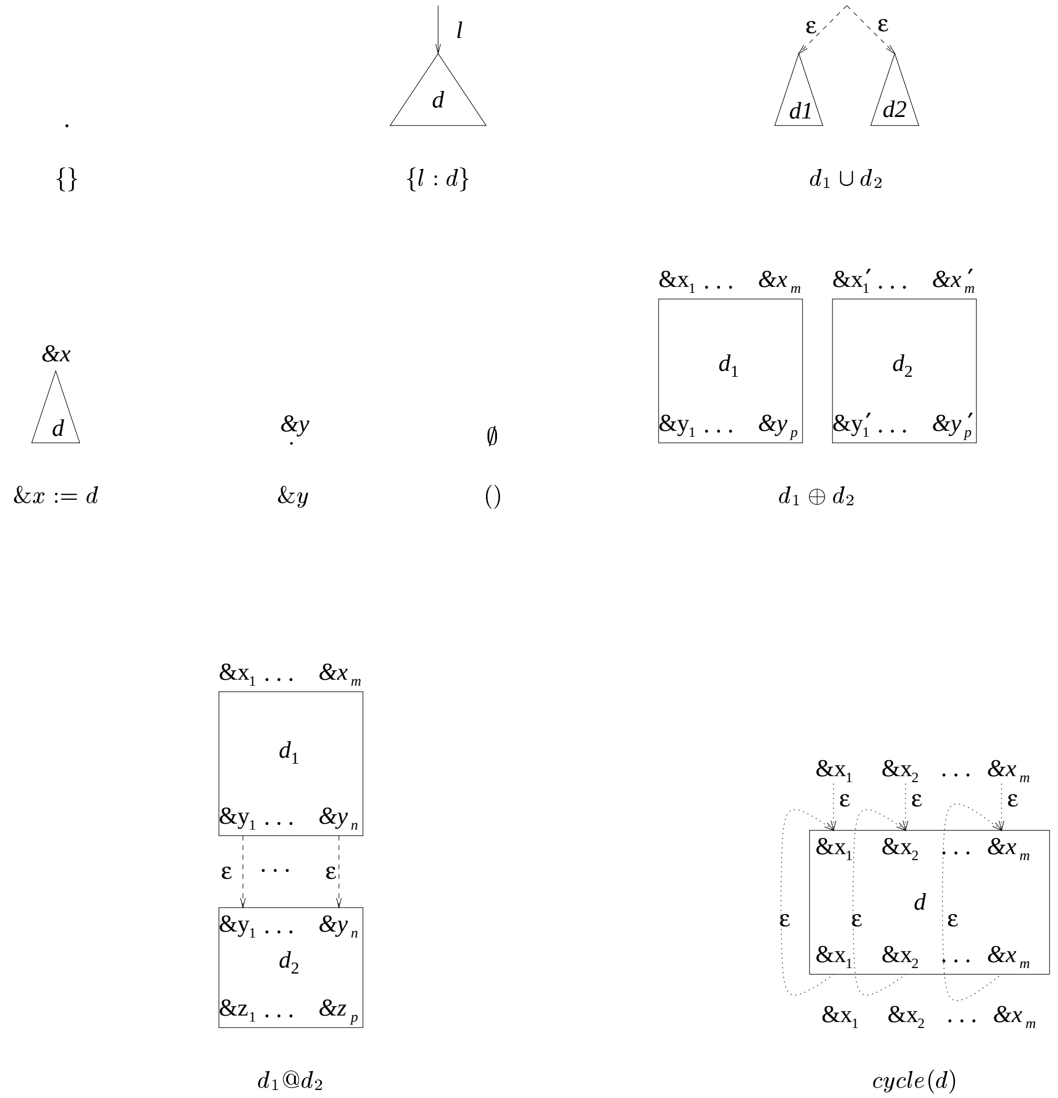


**Fig. 7** Definition of the constructors.

$d_1 @ d_2 = \{a_1 : \{a_2 : \ldots \{a_n : \{b_1 : \{b_2 : \ldots\}\}\} \ldots\}\}$. For another illustration, consider $d_1 = \{a : \&y_1,\ b,\ c : \&y_2\}$ and $d_2 = (\&y1 := \{d\},\ \&y2 := \{e, f\})$. Then $d_1 @ d_2$ is value equivalent[5] to $\{a : \{d\},\ b,\ c : \{e, f\}\}$. The effect is that of simultaneously substituting each output marker in $d_1$ with the value of the corresponding input marker in $d_2$. The last operator allows us to introduce cycles: when $d \in DB^{\mathcal{X}}_{\mathcal{X} \cup \mathcal{Y}}$, then $cycle(d) \in DB^{\mathcal{X}}_{\mathcal{Y}}$.

Each operator has a certain "type" given by the combination of allowed input and output markers. As said earlier, the operators are not polymorphic: we assume that each occurrence of an operator is subscripted with the set of input/output markers it expects, such as $\{\}_{\mathcal{Y}}$, or $d_1 @_{\mathcal{X},\mathcal{Y},\mathcal{Z}} d_2$, but we drop the subscripts to avoid clutter.

As a remark, recall that in Sec. 2 the union operator was undefined when one of the operands was an atomic value: e.g. $\{a\} \cup 5$ made no sense. In our definition here all constructors are well defined, as long as the types are correct. As an illustration, $\{a\} \cup \&y$ is defined and equal to $\{a, \varepsilon : \&y\}$ (recall that we do not have atomic values any more in our model).

[5] When we apply the definition in Fig. 7, $d_1 @ d_2$ results in $\{a : \{\varepsilon : \{d\}\},\ b,\ c : \{\varepsilon : \{e,\ f\}\}\}$.

We will use the following notations and abbreviations. As before, we abbreviate $\{l : \{\}\}$ with $\{l\}$, and use $\{l_1 : d_1, \ldots, l_n : d_n\}$ for $\{l_1 : d_1\} \cup \ldots \{l_n : d_n\}$. We write $(d_1, \ldots, d_n)$ for $d_1 \oplus \ldots \oplus d_n$. Finally, if $d, d' \in DB_{\mathcal{Y}}^{\mathcal{X}}$, and

$$d = (\&x_1 := d_1, \ldots, \&x_m := d_m)$$
$$d' = (\&x_1 := d'_1, \ldots, \&x_m := d'_m),$$

we extend the meaning of $\cup$ so that

$$d \cup d' = (\&x_1 := d_1 \cup d'_1, \ldots, \&x_m := d_m \cup d'_m)$$

That is, we take the union component-wise. This extends the union operator in a natural way from rooted data graphs (with unique input marker $\&$) to data graphs with arbitrary input markers.

We show next that the constructors are sufficient to enable us to express any data graph $d$, up to value equivalence. Consider first the case of a rooted data graph, i.e. with a single input marker $\&$, and let $d = (V, E, I, O)$, $V = \{v_1, \ldots, v_n\}$. We use $n$ distinct markers, $\&x_1, \ldots, \&x_n$, one corresponding to each vertex. For each vertex $v_i$, $i \in \{1, \ldots, n\}$, let $e_i$ be the union of the following expressions: $\{a : \&x_j\}$, for every edge $(v_i, a, v_j)$ with source $v_i$, and $\&y_k$, for every output marker $\&y_k$ labeling the vertex $v_i$. Assuming $v_1$ to be the root of $d$, the data graph is value equivalent to:

$$\&x_1 @\, cycle(\&x_1 := e_1, \ldots, \&x_n := e_n)$$

For an illustration, consider the graph in Fig. 6 (b), with three nodes and one output marker $\&y$; it can be expressed as:

$$\&x_1 @\, cycle(\&x_1 := \{a : \&x_2\}, \&x_2 := \{b : \&x_2, c : \&x_3\}, \&x_3 := \&y)$$

Consider now the case when the data graph $d$ has multiple input markers $\&x_1, \ldots, \&x_m$: then it is value equivalent to a graph of the form $(\&x_1 := d_1, \ldots, \&x_m := d_m)$ where $d_1, \ldots, d_m$ are rooted data graphs for which we apply the previous construction.

Thus, for any data graph $d$ we have a canonical expression denoting $d$. Of course, that expression is not unique: in general there are many equivalent expressions denoting the same graph.

We show next that all constructors are bisimulation-generic, and that this result is not accidental, but a consequence of a careful choice of both constructors and the definition of value-equivalence.

**Proposition 1** *All data constructors are bisimulation-generic.*

*Proof* This is done by applying the definition directly. We illustrate only for the operator $d_1 \cup d_2$, others are handled similarly. So let $d_1 \equiv d'_1$, $d_2 \equiv d'_2$. Denoting $d \stackrel{\text{def}}{=} d_1 \cup d_2$ and $d' \stackrel{\text{def}}{=} d'_1 \cup d'_2$ we have to prove that $d \equiv d'$. Let $S_1$ be the extended bisimulation relation from $d_1$ to $d'_1$ and $S_2$ the extended bisimulation relation from $d_2$ to $d'_2$. Note that $d$'s nodes are those in $d_1$ union those in $d_2$ plus a new node (the root): call it $u$. Similarly, the nodes of $d'$ are those in $d'_1$ union those in $d'_2$ plus a new node, denoted $u'$. (We may safely assume that $d_1, d_2, d'_1, d'_2$ have disjoint sets of nodes: we can always rename.) Define the relation $S$ to be $S_1 \cup S_2 \cup \{(u, u')\}$. Now we verify that $S$ is an extended bisimulation. For that we need to check conditions 1-4 of Def. 3. Take condition 1. Consider a $\varepsilon^*.a$ path in $d = d_1 \cup d_2$. It can either (a) be entirely in $d_1$, then we use the fact that $S_1$ is an extended bisimulation; or, (b) be entirely in $d_2$, then use the fact that $S_2$ is an extended bisimulation; or (c) start at the root and continue, say, in $d_1$ (the case when it continues in $d_2$ is similar). Here we apply $S_1$ to the remainder of the path in $d_1$ and obtain a similar path in $d_1$' (for that we use condition 4 on $S_1$, which implies that the roots of $d_1$ and $d'_1$ are in $S_1$). Conditions 2 and 3 are checked similarly. Condition 4 holds trivially, since $(u, u') \in S$. □

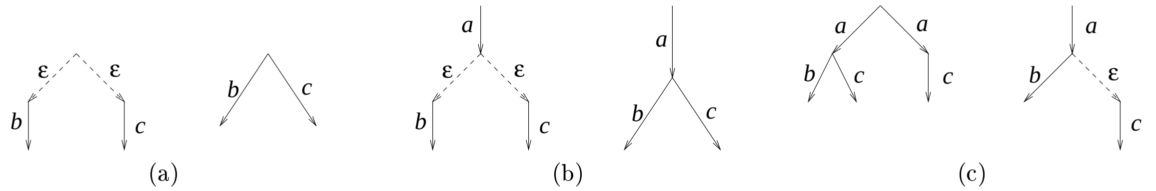


**Fig. 8** Relation between extended bisimulation and weak bisimulation.

$$
\begin{aligned}
E ::= & \{\} \mid \{L : E\} \mid E \cup E \mid \\
& \&x := E \mid \&y \mid () \mid E \oplus E \\
& \mid E@E \mid cycle(E) \mid \\
& Var \mid \text{if } B \text{ then } E \text{ else } E \mid rec(\lambda(LabelVar, Var).E)(E) \\
L ::= & LabelVar \mid a \; a \in Label \\
B ::= & isempty(E) \mid L = L
\end{aligned}
$$

**Fig. 9** UnCAL

*Weak bisimulation* Extended bisimulation is related to, but distinct from, the "weak bisimulation" used in process algebra [Mil89], where the "silent transitions" ($\tau$) correspond to our $\varepsilon$ edges. Namely two graphs $d_1, d_2$ are "weakly bisimilar" [Mil89] if there exists some relation $S$ as in Definition 3, but in which the path $\varepsilon^*.a$ in the first condition is replaced with a path $\varepsilon^*.a.\varepsilon^*$. We cannot use weak bisimulation as our notion of value equivalence, because the constructors would not preserve this equivalence. We illustrate this with the singleton constructor $\{a : d\}$. Consider the two data graphs $d_1, d_2$ in Figure 8 (a): they are weakly bisimilar, but $\{a : d_1\}$ and $\{a : d_2\}$ (Fig. 8 (b)) are not (in the left graph the root has a path $a.\varepsilon$ to a node that is not weakly bisimilar to any node in the right graph). Weak bisimulations and extended bisimulations are incomparable. The two graphs in Fig. 8 (b) are extended bisimilar but not weakly bisimilar, while the two graphs in Figure 8 (c) are weakly bisimilar but not extended bisimilar.

## 5 UnCAL: a Query Language for Graphs

UnCAL (Unstructured Calculus) is UnQL's internal algebra, and its syntax is depicted in Fig. 9. The name "calculus" should be understood here in the sense of $\lambda$-calculus, and not of relational calculus: UnCAL is in fact closer in spirit to the relational algebra than to the relational calculus. It consists of the graph constructors introduced earlier, variables, conditionals, and structural recursion. We have defined the data constructors already, while variables and conditionals are self explanatory. Structural recursion in UnCAL, in notation $rec(e)$, will be described next.

### 5.1 Structural Recursion on Graphs

Extending structural recursion to arbitrary graphs seems impossible at first, since recursive functions run into infinite loops when applied to graphs with cycles. We give two solutions to this problem. One is memoization: remember all recursive calls and avoid entering infinite loops. We call this the *recursive semantics*. Another is an entirely different view on structural recursion: apply the recursive functions in parallel, on all graph's edges. Hence each function will be applied only as many times as edges in the graph, and infinite loops are avoided. We call this the *bulk semantics*. Moreover the two semantics turn out to be equivalent – an evidence of the robustness of structural recursion. A third choice is to do an inductive definition based on a constructor expression for the given data graph: such a definition is also possible (and we will show it to be equivalent to the other two), however it can only be done under some technical restrictions.

*Encoding Several Recursive Functions as One* Before we define the semantics of structural recursion, we argue that multiple recursive functions can always be encoded as a single recursive function. Let us start by illustrating on the functions `g, h` below:

```
sfun g({a:T}) = {a:h(T)}            sfun h({b:T}) = {c:h(T)}
   | g({L:T}) = g(T)                   | h({L:T}) = {L:h(T)}
```

On a tree `T`, `g(T)` erases all edges until it reaches an `a`. After that it copies the tree, but replaces every `b` with a `c`. To rewrite the two functions as a single recursive function, we define an auxiliary function `e(L)`, where `L` is a label:

```
fun e(L) = case L of
             a:  (&z1 := {a: &z2}, &z2 := {a: &z2})
             b:  (&z1 := &z1,      &z2 := {c: &z2})
             _:  (&z1 := &z1,      &z2 := {L: &z2})
```

For a label `L`, `e(L)` returns a graph with two input markers, `&z1, &z2`, and the same two output markers. Fig. 10 (a) illustrates the three possible shapes for `e(L)`. Recall that when a graph has the same set of input markers and output markers (`{&z1, &z2}` in our example), each marker occurrence is clearly distinguished as either input or output.

Consider now the following structural recursion function.

```
fun f(T1 U T2) = f(T1) U f(T2)
  | f({L:T})   = e(L) @ f(T)
  | f({})      = (&z1 := {}, &z2 := {})
```

From the last line we read that the "type" of `f(T)` is a graph with input markers `&z1` and `z2`. Technically this definition is an extension of our syntax for structural recursion, in that it allows the function to return a forest instead of a tree: a forest is a particular instance of a graph with multiple input nodes. We will allow this extension of structural recursion in the sequel. In the first line we use our convention that a union applied to two graphs with input markers `&z1` and `&z2` computes the union of the `&z1` trees in both graphs, and separately the union of the `&z2` trees in both graphs (Sec 4.2). As before we write the definition of the structural recursion function as:

```
sfun f({L:T}) = e(L) @ f(T)
```

The relationship between `f` on one hand and `g, h` on the other, is the following. For any finite tree `T`:

```
f(T) = (&z1 := g(T),  &z2 := h(T))
```

We sketch the proof by induction on `T`. For `T` $= \{\}$, both sides of the equation are `(&z1:={}, &z2:={})`. For `T` $=$ `T1 U T2`, the left hand side is `f(T1) U f(T2)` which, by induction hypothesis is:

```
(&z1 := g(T1), &z2 := h(T1)) U (&z1 := g(T2), &z2 := h(T2))
```

The right hand side of the equation is:

```
(&z1 := g(T1) U g(T2),  &z2 := h(T1) U h(T2))
```

The two are equal. Finally, when $\mathtt{T} = \{\mathtt{L} : \mathtt{T}'\}$, then one makes a case analysis on `L`. When $\mathtt{L} = \mathtt{a}$, then the left hand side is $\mathtt{e(a)}@\mathtt{f(T')}$; since the induction hypothesis holds, after substituting[6] in the body of `e(a)`, the left hand side becomes $(\&\mathtt{z1} := \{\mathtt{a} : \mathtt{h(T')}\}, \&\mathtt{z2} := \{\mathtt{a} : \mathtt{h(T')}\})$, which, by the definitions of `g` and `h` is equal to $(\&\mathtt{z1} := \mathtt{g}(\{\mathtt{a} : \mathtt{T'}\}), \&\mathtt{z2} := \mathtt{h}(\{\mathtt{a} : \mathtt{T'}\}))$. The other cases are similar.

It follows that we can replace the two mutually recursive functions `g, h` with a single recursive function `f`. To obtain `g(T)`, one computes `&z1 @ f(T)`; for `h(T)` one computes `&z2 @ f(T)`.

This construction can be generalized to $k$ mutually recursive functions $g_1, \ldots, g_k$, as follows. Let $\mathcal{Z} = \{\&z_1, \ldots, \&z_k\}$ be a set of $k$ markers. Let each function $g_i$ be defined by *sfun* $g_i(\{l : t\}) = e_i$. Recall that the expression $e_i$ may contain $l$, $t$, and any of the recursive calls $g_1(t), \ldots, g_k(t)$: substitute the latter with the markers $\&z_1, \ldots, \&z_k$ respectively, and call the resulting expression $e'_i(l,t)$. Define $e(l,t) \stackrel{\text{def}}{=} (\&z_1 := e'_1(l,t), \ldots, \&z_k := e'_k(l,t))$. Then the single structural recursive function expressing all $k$ functions $g_1, \ldots, g_k$ is:

$$\textit{sfun}\ f(\{l,t\}) = e(l,t)@f(t) \tag{1}$$

---

[6] Recall that the `@` operator is analogous to marker substitution.

Note that, if any of the expressions $e_i$ used the argument $t$ explicitly, then $e$ will also use $t$ explicitly; in our previous example neither `g` nor `h` used `t` explicitly, hence it did not occur in `e` either.

One can easily establish the following equation, on all finite trees $t$, by induction on $t$:

$$f(t) = (\&z_1 := g_1(t), \ldots, \&z_k := g_k(t))$$

In particular we can obtain $g_1(t)$ as $\&z_1 @ f(t)$.

Hence, from now on, we will only consider a single structural recursive function $f$, of the form (1). The syntax for it in UnCAL (Fig. 9) is $rec(\lambda(l,t).e)$, or simply $rec(e)$, when the label and tree variables $l, t$ are understood. As for other UnCAL operators, the set $\mathcal{Z}$ of markers used in $e$ should normally be written as a subscript to $rec$, but we drop it to avoid clutter.

*Structural Recursion and Markers* Before defining structural recursion on graphs, we need to be precise about the "type", i.e. the set of input and output markers in the result of structural recursion. It follows from the discussion above that, if $t$ is a tree (i.e., a single input marker & and no output markers), then $rec(e)(t)$ is in $DB^{\mathcal{Z}}_{\emptyset}$, where $\mathcal{Z}$ is the set of markers used in $e$. The case of $rec(e)(d)$, when $d$ has an arbitrary set of input markers $\mathcal{X}$ and an arbitrary set of output markers $\mathcal{Y}$ requires more care however. We will encounter such uses of structural recursion later (Sec. 7), when we describe optimization techniques, and it is important to define the semantics of structural recursion correctly, in order for those techniques to work.

Specifically, we need a way to generate new markers, and for that we introduce the following marker constructor: given two markers $\&x, \&y$, $\&x \cdot \&y$ denotes a new marker. The operation $\cdot$ returns a different marker for every pair $\&x, \&y$. One can think of $\cdot$ as a Skolem function on markers, but with the following difference: we assume $\cdot$ to be associative, $(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z)$ and & to be its identity: $\& \cdot \&x = \&x \cdot \& = \&x$. In other words, our universe of all markers, denoted $Marker$, is now a monoid, with & the identity[7] Given two sets of markers $\mathcal{X}, \mathcal{Y}$, we denote $\mathcal{X} \cdot \mathcal{Y}$ the set $\{\&x \cdot \&y \mid \&x \in \mathcal{X}, \&y \in \mathcal{Y}\}$.

We can now explain the types in structural recursion. Let $e$ be a function $e : Label \times DB_{\mathcal{Y}} \to DB^{\mathcal{Z}}_{\mathcal{Z}}$. Then the type of $rec(e)$, is defined to be a function $DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}}$. Normally we should have denoted $rec_{\mathcal{X},\mathcal{Y},\mathcal{Z}}(e)$, but, as usual, we drop the subscripts. The type of $rec(e)(t)$ when $t$ is a tree, follows from this general rule: here $\mathcal{X} = \{\&\}$, $\mathcal{Y} = \emptyset$, hence the type of $rec(e)(t)$ is $DB^{\mathcal{Z}}_{\emptyset}$, because $\{\&\} \cdot \mathcal{Z} = \mathcal{Z}$ and $\emptyset \cdot \mathcal{Z} = \emptyset$.

*5.1.1 Bulk Semantics of Structural Recursion* Given $f = rec(e)$ and an input data graph $d$, we want to define the result $f(d)$ of computing $f$ on $d$. In the bulk semantics we apply the body $e$ independently on all edges in $d$, then join the results with $\varepsilon$ edges (as in the @ operation). Continuing our example, the result of the bulk semantics is illustrated in Fig. 10. The figure illustrates first the effect of applying the function `e` on a single edge in (a), then the bulk semantics of `f(d)` for the tree $\mathtt{d} = \{\mathtt{b}, \mathtt{c} : \{\mathtt{a} : \{\mathtt{b}, \mathtt{d}\}, \mathtt{b}\}, \mathtt{a} : \{\mathtt{e}\}\}$ in (b): this result reads

$$\mathtt{f(d)} = (\&\mathtt{z}_1 := \{\mathtt{a} : \{\mathtt{c}, \mathtt{d}\}, \mathtt{a} : \{\mathtt{e}\}\}, \&\mathtt{z}_2 := \{\mathtt{c}, \mathtt{c} : \{\mathtt{a} : \{\mathtt{c}, \mathtt{d}\}, \mathtt{c}\}, \mathtt{a} : \{\mathtt{e}\}\})$$

The reader may compute `g(d)` and `h(d)` separately, as recursive functions on the tree `d`, and check that indeed $\mathtt{f(d)} = (\&\mathtt{z}_1 := \mathtt{g(d)}, \&\mathtt{z}_2 := \mathtt{h(d)})$.

We define the bulk semantics in general. Fix a data graph $d = (V, E, I, O)$, $d \in DB^{\mathcal{X}}_{\mathcal{Y}}$. Given a node $v \in V$, we denote $d_v$ to be "$d$ with root $v$", i.e. $d_v = (V, E, I_v, O)$, with $I_v = \{(\&, v)\}$; notice that $d_v \in DB_{\mathcal{Y}}$. Consider now a function defined by structural recursion, $f = rec(e)$, where the body $e$ is $e : Label \times DB_{\mathcal{Y}} \to DB^{\mathcal{Z}}_{\mathcal{Z}}$. In the bulk semantics, the data graph $d' = f(d)$ is constructed as follows. First make $\mid \mathcal{Z} \mid$ disjoint copies of $V$: each such node will be identified as $S1(u, \&z)$, for $u \in V$ and $\&z \in \mathcal{Z}$. Next apply $e$ on each edge $(u, a, v) \in E$, where $a \neq \varepsilon$, more precisely, compute the data graph $e(a, d_v)$, and take the disjoint union of all these $\mid E \mid$ data graphs. Their nodes will be uniquely identified as $S2(u, a, v, w)$, where $(u, a, v) \in E$ and $w \in e(a, d_v).V$: here and in the sequel we use the notation $g.V, g.E, g.I, g.O$ for the vertices, edges, inputs, and outputs of some graph $g$. Referring to Fig. 10, the $S1$ nodes are outside the boxes, while the $S2$ nodes are inside the boxes. Finally we add edges connecting $S1$ nodes to $S2$ or to other $S1$ nodes, based on the

[7] Associativity of "·" is needed for Theorem 4 to hold. In particular, Theorem 4 (1) is related to a "associativity" law in monads, see [Wad92]. If we don't require "·" to be associative, then we have to do some messy marker renamings for Theorem 4 to hold.

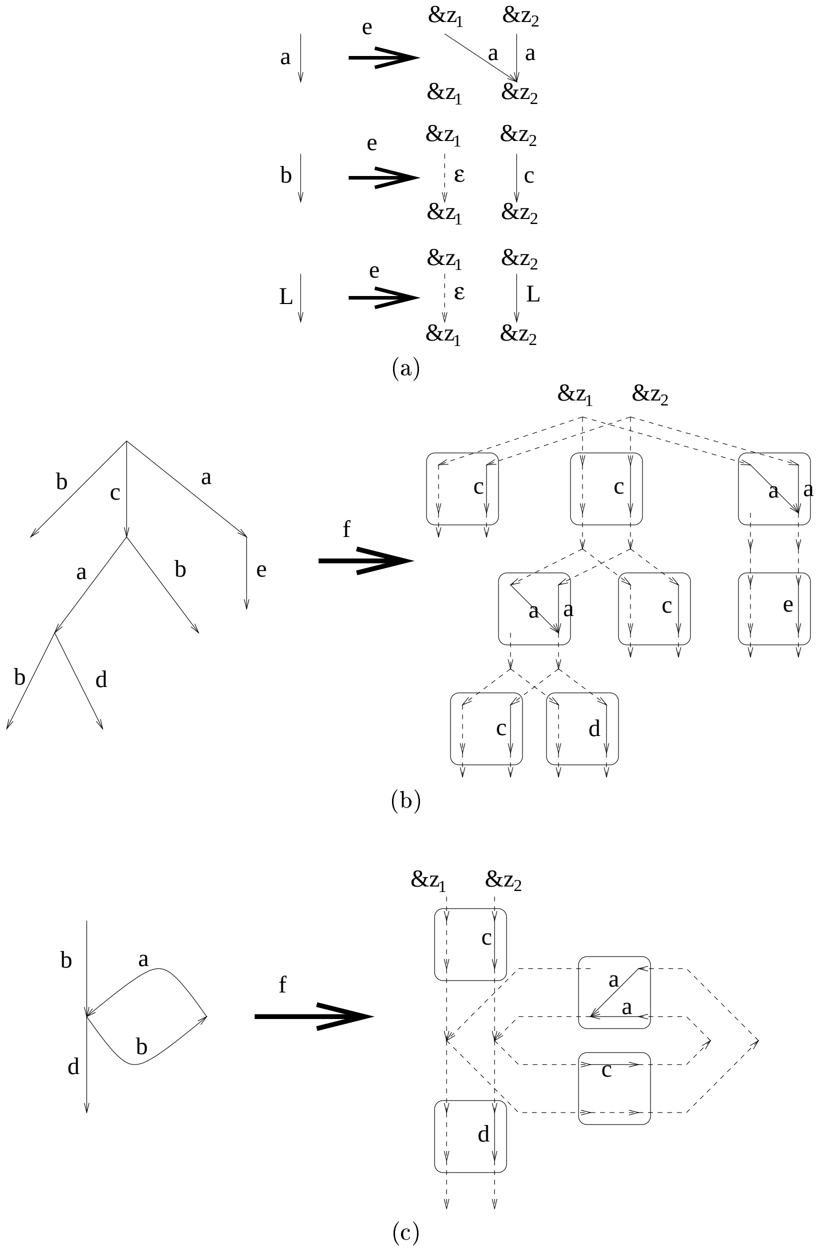


**Fig. 10** Illustration for the bulk semantics of structural recursion.

edges in $d$. For an edge $(u, a, v) \in E$ with $a \neq \varepsilon$, for every $(\&z, w) \in e(a, d_v).I$, where $\&z \in \mathcal{Z}$, we add an $\varepsilon$ edge from $S1(u, \&z)$ to $S2(u, a, v, w)$; similarly for every $(w, \&z) \in e(a, d_v).O$ we add an $\varepsilon$ edges from $S2(u, a, v, w)$ to $S2(v, \&z)$. For an $\varepsilon$ edge $(u, \varepsilon, v) \in E$, for every $\&z \in \mathcal{Z}$ we add an $\varepsilon$ edge from $S1(u, \&z)$ to $S1(v, \&z)$, for every marker $\&z \in \mathcal{Z}$.

The above discussion leads us to the following formal definition of the *bulk semantics* for structural recursion. Given $rec(e)$ and a data graph $d$, the data graph $d' = rec(e)(d)$ is:

$$
\begin{aligned}
d'.V \stackrel{\text{def}}{=} & \{S1(u, \&z) \mid u \in V, \&z \in \mathcal{Z}\} \cup \\
& \{S2(u, a, v, w) \mid (u, a, v) \in E, a \neq \varepsilon, w \in e(a, d_v).V\} \\
d'.E \stackrel{\text{def}}{=} & \{(S1(u, \&z), \varepsilon, S2(u, a, v, w)) \mid \&z \in \mathcal{Z}, (u, a, v) \in E, a \neq \varepsilon, (\&z, w) \in e(a, d_v).I\} \cup \\
& \{(S2(u, a, v, w), b, S2(u, a, v, w')) \mid (u, a, v) \in E, a \neq \varepsilon, (w, b, w') \in e(a, d_v).E\} \cup
\end{aligned}
$$

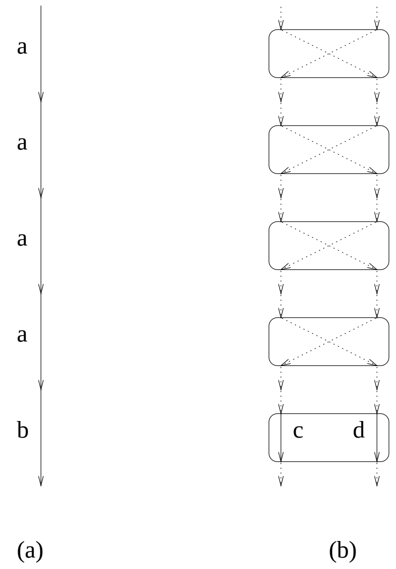


**Fig. 11** A chain data graph (a) and the bulk semantics of `even-odd` (b).

$$\{(S2(u,a,v,w),\varepsilon,S1(v,\&z)) \mid \&z \in \mathcal{Z}, (u,a,v) \in E, a \neq \varepsilon, (w,\&z) \in e(a,d_v).O\} \cup$$
$$\{(S1(u,\&z),\varepsilon,S1(v,\&z)) \mid \&z \in \mathcal{Z}, (u,\varepsilon,v) \in E\}$$
$$d'.I \stackrel{\text{def}}{=} \{(\&x \cdot \&z, S1(u,\&z)) \mid (\&x,u) \in I, \&x \in \mathcal{X}, \&z \in \mathcal{Z}\}$$
$$d'.O \stackrel{\text{def}}{=} \{(S1(u,\&z), \&y \cdot \&z) \mid (u,\&y) \in O, \&y \in \mathcal{Y}, \&z \in \mathcal{Z}\}$$

Here, $S1$ and $S2$ are Skolem functions. Each of these sets is definable by a first order formula. Notice also that, when $d \in DB^{\mathcal{X}}_{\mathcal{Y}}$, then according to this definition $d' \in DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}}$, which is the same as the type defined earlier.

It is interesting to notice that the bulk semantics is expressed entirely in first order logic extended with Skolem functions (in fact, only non-recursive datalog is needed). This may sound surprising since, after all we are expressing recursion ! It does so by introducing additional $\varepsilon$ edges in the result, with the aid of the markers. If we want to eliminate these edges, then, in general, we have to compute a transitive closure. In effect, the bulk semantics delays computing the recursion and isolates it in a single computation of a transitive closure (of $\varepsilon$ edges).

To appreciate the power of such a semantics, consider the following example:

```
sfun even({a:T}) = odd(T)          sfun odd({a:T}) = even(T)
   | even({b:T}) = {c}                 | odd({b:T}) = {d}
```

The function `even` runs down a chain of `a`-edges until it finds a `b`-edge, and outputs `c` if the `a`-chain was even in length, and `d` otherwise. On a chain `T` like in Fig. 11 (a), the bulk semantics returns a pair of zigzagging, crossing chains, Fig. 11 (b), which apparently do not make us much wiser. If we really want to know if the unique `b` was at an odd depth or an even depth, we need to eliminate the $\varepsilon$ edges starting from the left input marker (corresponding to the function `even`).

It is also important to note that a query processor does not have to eliminate $\varepsilon$ edges after every application of structural recursion. All UnCAL's operators, except one, can cope with $\varepsilon$ edges. For example, assume we have to compute `g(even(T))`, where `T` is the chain in Fig. 11 (a), and the function `g` is:

```
sfun g({c:T}) = {e}
   | g({d:T}) = {}
```

We can apply `g` to the data graph in Fig. 11 (b) without having to eliminate the $\varepsilon$ edges first: the result will be a similarly shaped graph with an `e` label instead of `c`, and with the `d` edge deleted. Thus, the query processor can postpone $\varepsilon$ elimination until the end of the computation. The only operator requiring $\varepsilon$ elimination is *isempty*: for example, to compute `isempty(g(even(T)))`, we have no choice but to check if the unique non-$\varepsilon$ edge (`e`) is reachable from the left marker.

There is some similarity here to duplicate elimination in relational queries without aggregates, where an optimizer also has a choice when to perform duplicate elimination.

In our definition of the bulk semantics we introduced $\varepsilon$ edges quite generously. This was convenient for the definition, but in many practical cases one can save several, or even all the $\varepsilon$ edges in the definition. For example in Fig. 11 (b) the connecting $\varepsilon$ edges, between the output nodes of one graph and the input nodes of the next could be spared, and the nodes glued instead: this is a form of optimization which can be generalized to the case when the input data graph has no loops (edges from some node to itself). Alternatively, one could glue the input and output nodes in each graph: in Fig. 11 one would glue the input `&z1` to the output `&z2`, and conversely, in each of the graphs in the boxes. Of course, both optimizations can not be applied together in this example, unless we compute a transitive closure.

Finally note that the result of any structural recursion computation is polynomial in the size of the input data. Exponential blowup resulting from recursive processing of trees are avoided by representing the results as directed acyclic graphs. For example the function `f4` in Sec. 2.2, when applied to the tree $u_1 \stackrel{a}{\to} u_2 \stackrel{b}{\to} u_3 \stackrel{c}{\to} u_4$ results essentially in the graph:

$$v_1 \underset{b}{\overset{a}{\rightrightarrows}} v_2 \underset{b}{\overset{a}{\rightrightarrows}} v_3 \underset{b}{\overset{a}{\rightrightarrows}} v_4$$

*5.1.2 Recursive Semantics of Structural Recursion* The idea here is very simple. Given a function $f = rec(e)$ or, equivalently, *sfun* $f(\{l : t\}) = e@f(t)$, we compute $f(d)$ recursively on $d$, essentially by unfolding $d$, as if the input were a tree, but memoize all recursive calls to avoid infinite loops. The algorithm is depicted in Fig. 12; we explain it next. Let $p$ be the number of input and output markers of $e$, i.e. its markers are $\&z_1, \ldots, \&z_p$. The algorithm traverses recursively the graph using the function $r(u)$ (whose argument is a node in the graph). Whenever we visit a node $u$ we check if it is in a list *visited* of nodes we have seen so far. Initially *visited* $= \emptyset$. If $u$ is not there, then, before any recursive call is made, we create a new data graph $s1 = (\&z_1 := \{\}, \ldots, \&z_p := \{\})$, and insert in *visited* the pair $(u, s1)$. The data graph $s1$ is a mutable object: it is similar to a record with $p$ components, in which each component is initially $\{\}$. Each of its $p$ input nodes will represent some node in the result graph (an $S1$ node, in the terminology of the bulk semantics), and its set of edges is initially empty. Next, we iterate over all edges $(u, a_i, v_i)$ and update $s1$ to $s1 \cup e(a_i, d_{v_i})$: as before, $d_{v_i}$ denotes the graph $d$ with node $v_i$ designated as input. If $u$ however is already in *visited* then we retrieve the graph $s1$ from there and return it. More precisely we return the $p$ input nodes of $s1$ (and their set of outgoing edges may not be fully computed yet).

When the input is a tree, then the recursive semantics coincides with that described in Sec. 2. Indeed, on some non-empty node $\{a_1 : t_1, \ldots, a_n : t_n\}$ the function $r$ returns $e(a_1, f(t_1)) \cup \ldots \cup e(a_n, f(t_n))$, which is precisely what the structural recursive function $f$ does, since $f(t \cup t') = f(t) \cup f(t')$ and $f(\{a : t\}) = e(a, t)@f(t)$.

*5.1.3 Equivalence of the Bulk and Recursive Semantics* We have given two distinct definitions for the semantics of structural recursion. Here we establish their equivalence.

**Proposition 2** *The recursive semantics is value equivalent to the bulk semantics.*

*Proof* For some structural recursive function $rec(e)$ let $f_1$ denote its bulk semantics and $f_2$ its recursive semantics. Consider some data graph $d$. We have to show that $f_1(d) \equiv f_2(d)$. Let $d'$ be the accessible part of $f_1(d)$ (i.e. all nodes accessible from some of its input nodes). We show that $f_2(d)$ constructs a graph isomorphic to $d'$, in notation $d' = f_2(d)$; graph isomorphism, of course, implies value equivalence, hence this proves the proposition. For this strong claim to hold we need to explain in more detail what graph exactly the Algorithm in Fig. 12 constructs. First, the statement:

$$s1 := (\&z_1 := \{\}, \ldots, \&z_p := \{\})$$

Algorithm: Recursive Evaluation of Structural Recursion
Input: $f = rec(e)$ and an input graph $d$
Output: $f(d)$
Method: /* assume $e$ returns a graph with input and output markers $\&z_1, \ldots, \&z_p$ */
/* assume $d = (\&x_1 := d_1, \ldots, \&x_m := d_m)$ */
$visited := \{\}$
for $i = 1$ to $m$ do let $(\&z_1 := r_{i1}, \ldots, \&z_p := r_{ip}) := r(d_i)$
return $(\&x_1 \cdot \&z_1 := r_{11}, \&x_1 \cdot \&z_2 := r_{12}, \ldots, \&x_m \cdot \&z_p := r_{mp})$

```
fun r(u) =
  case u of
    {} : return (&z1 := {}, ..., &zp := {})
    &y : return (&z1 := &y · &z1, ..., &zp := &y · &zp)
    {a1 : v1, ..., an : vn} : /* n > 0 */
       if exists (u, s1) ∈ visited then return s1
       else s1 := (&z1 := {}, ..., &zp := {})
            visited := visited ∪{(u, s1)}
            for i = 1 to n do
               if ai = ε then s1 := s1 ∪ r(vi)
               else s1 := s1 ∪ e(ai, dvi) /* dvi is graph d with new input vi */
            return s1
```

**Fig. 12** The recursive semantics of structural recursion

returns a graph with $p$ nodes (labeled with input markers $\&z1, \ldots, \&zp$) and no edges. The assignment:

$$s1 := s1 \cup e(a_i, d_{v_i})$$

proceeds as follows. Notice that $e(a_i, d_{v_i})$ returns a graph with $p$ input nodes. Then, the effect of the assignment is to add $p$ $\varepsilon$ edges from the $p$ inputs of $s1$ to the $p$ inputs of that graph. Similarly for the statement:

$$s1 := s1 \cup r(v_i)$$

Notice that the recursive algorithm constructs an accessible graph $f_2(d)$. The claim that $d' = f_2(d)$ follows now immediately, since the nodes $s1$ constructed by $f_2(d)$ correspond isomorphically to the $S1$ nodes in $f_1(d')$, while the nodes in the graphs $e(a_i, d_{v_i})$ in $f_2(d)$ correspond isomorphically to the $S2$ nodes in $f_1(d)$. □

*5.1.4 Structural Recursion and Constructors* When we introduced the graph constructors in Sec. 4.2 we showed that any data graph $d$ can be defined by an expression using only these constructors. The third alternative for the semantics of structural recursion is to define it inductively, on the structure of this expression. We will prove this below, referring to the graph constructors in Fig. 7, by showing that structural recursion "commutes" with each constructor. However, for the @ and *cycle* constructors the corresponding equations hold only if the expression $e(l, t)$ in $rec(e)$ does not depend on the tree variable $t$. Because of this limitation, we do not adopt this as an alternative definition of structural recursion. Nevertheless, these equations can be useful to an optimizer.

**Proposition 3** *The following data value equalities hold for structural recursion:*

$$\begin{aligned} rec(e)(\{\}) &\equiv \{\} \\ rec(e)(\{l : d\}) &\equiv e(l, d) @ rec(e)(d) & (2) \\ rec(e)(d_1 \cup d_2) &\equiv rec(e)(d_1) \cup rec(e)(d_2) \\ rec(e)(\&x := d) &\equiv \&x \cdot (rec(e)(d)) & (3) \end{aligned}$$

$$
\begin{aligned}
rec(e)(\&y) &\equiv (\&z_1 := \&y \cdot \&z_1, \ldots, \&y \cdot \&z_p) \\
rec(e)() &\equiv () \\
rec(e)(d_1 \oplus d_2) &\equiv rec(e)(d_1) \oplus rec(e)(d_2) \\
rec(e)(d_1 @ d_2) &\equiv rec(e)(d_1) @ rec(e)(d_2) \quad (4)\\
rec(e)(cycle(d)) &\equiv cycle(rec(e)(d)) \quad (5)
\end{aligned}
$$

*In Eq.(3),* $\&x \cdot (\&z_1 := g_1, \ldots, \&z_p := g_p)$ *denotes* $(\&x \cdot \&z_1 := g_1, \ldots, \&x \cdot \&z_p := g_p)$.[8] *In Equations (4) and (5) we make the additional assumption that* $e(l,t)$ *does not depend on the tree variable* $t$.

*Proof* We sketch here the proof for (4) and (5). The graph $d_1 @ d_2$ is obtained by taking the disjoint union of the graphs $d_1$ and $d_2$, then adding certain connecting $\varepsilon$ edges (see the definition in Fig. 7). Since the bulk semantics for the graph $d' = rec(e)(d_1 @ d_2)$ is obtained by applying $e$ on each edge in $d_1 @ d_2$ independently, we can obtain $d'$ by applying $e$ on the edges in $d_1$ separately, then on those in $d_2$, and finally on the connecting $\varepsilon$ edges. The first set of graphs is precisely $rec(e)(d_1)$; the second is $rec(e)(d_2)$; and finally, the third forms precisely the $\varepsilon$ edges connecting $rec(e)(d_1)$ to $rec(e)(d_2)$ in $rec(e)(d_1) @ rec(e)(d_2)$. The case $rec(e)(cycle(d))$ is treated similarly: here $cycle(d)$ is obtained by adding $\varepsilon$ edges to $d$, hence the equation follows easily. All other equalities are easily proved, in a similar fashion, using the bulk semantics for structural recursion. Note that in both (4) and (5) we obtain actual graph isomorphism, in the bulk semantics. In some of the other cases we have only a weaker value equality. For example in (2) we have two $\varepsilon$ edges between the output of $e(l,d)$ and the input of $rec(e)(d)$ on the left hand side, but there are three such edges on the right hand side (the @ operator adds the third one). Hence the two sides are value equivalent, but not isomorphic. □

Note that if $e(l,t)$ depends on $t$, then, when $e$ is applied on some edge in $d_1 @ d_2$ which belongs to $e_1$ it can, through $t$, access and inspect the graph $d_2$, hence we cannot obtain the same effect by computing $rec(e)(d_1)$ and $rec(e)(d_2)$ independently. The following example illustrates a case when Eq. (4) fails. Consider the recursive function

```
sfun f({L:T}) = if not(isempty(h(T))) then {L} U f(T) else f(T)
```

returning all edges which have an `a` below them. Here the function `h` is defined as

```
sfun h({a:T}) = {a}
   | h({L:T}) = h(T)
```

These are not mutually recursive, but `h` is defined earlier than `f` (otherwise `f` would not be allowed to test `isempty(h(T))`, see condition 2 in Sec. 2.2). That is, in UnCAL the function `f(d)` becomes:

$$rec(\lambda(l,t).\ \mathsf{if}\ \mathsf{not}(isempty(rec(\lambda(l',t').\ldots)(t)))\ \mathsf{then}\ \{L\} \cup \&\ \mathsf{else}\ \&)(d)$$

where $\ldots$ stands for the body of `h` which is: $\mathsf{if}\ l' = a\ \mathsf{then}\ \{a\}\ \mathsf{else}\ \&$. Consider now a linear graph $\mathtt{d} = \{\mathtt{b} : \{\mathtt{a} : \{\mathtt{c}\}\}\}$ on which we can compute `f` directly, using the recursive definition, and obtain `f(d)` $= \{\mathtt{b}\}$ (because there is an `a` edge below `b`). However if we express `d` as $\mathtt{d_1} @ \mathtt{d_2}$, with $\mathtt{d_1} = \{\mathtt{b} : \&\mathtt{y}\}$ and $\mathtt{d_2} = (\&\mathtt{y} := \{\mathtt{a} : \{\mathtt{c}\}\})$, then both $\mathtt{f(d_1)}$ and $\mathtt{f(d_2)}$ return the empty graph.

### *5.2 UnCAL Queries*

UnCAL expressions are built from variables and label constants using the operators depicted in Fig. 9. The $rec(\lambda(L,T).e)(e')$ construction is the only one which introduces new variables, $L$ and $T$, whose scope is the expression $e$.

An UnCAL **query** is any expression with a single free variable $db$.

We illustrate with the following example, taken from Sec. 2:

```
let sfun h1({ country: C }) =
     let sfun h2({ name: N }) = { result: N }
     in h2(C)
in h1(db)
```

[8] $\&x \cdot d$ can be expressed using the other constructors, as $(\&x \cdot \&z_1 := \&z_1 @ d, \ldots, \&x \cdot \&z_p := \&z_p @ d)$.

The corresponding UnCAL query is:

$$
\begin{aligned}
&rec(\lambda\ (L1, C).\\
&\qquad \mathsf{if}\ L1 = \text{"}country\text{"}\\
&\qquad\qquad \mathsf{then}\ rec(\lambda(L2, N).\ \mathsf{if}\ L2 = \text{"}name\text{"}\ \mathsf{then}\ \{\text{"}result\text{"} : N\}\ \mathsf{else}\ \{\})(C)\ \mathsf{else}\ \{\})(db)
\end{aligned}
$$

Here we used & everywhere as the recursion marker to simplify the *rec* expressions. Even so, UnCAL is hard to read: it is intended only as an internal algebra for UnQL, and not as user syntax.

The discussion on polymorphism carries over from constructors to all UnCAL expressions: such expressions are not polymorphic, but are assumed to have the expected input and output markers explicitly stated (however we drop the marker indexes whenever they are clear from the context).

### *5.3 UnCAL Queries Are Bisimulation Generic*

**Proposition 4** *If $e$ is bisimulation-generic, then the structural recursion function $rec(e)$, is also bisimulation generic.*

*Proof* We have to show that, for any two data graphs $d, d'$, if $d \equiv d'$, then $rec(e)(d) \equiv rec(e)(d')$. We use the bulk semantics for structural recursion, hence both $g \stackrel{\text{def}}{=} rec(e)(d)$ and $g' \stackrel{\text{def}}{=} rec(e)(d')$ are given by the definition in Sec. 5.1.1. Recall our notation $d.V, d.E, d.I$, and $d.O$ for the vertices, edges, inputs, and outputs of some data graph $d$.

More precisely, given an extended bisimulation $R$ from $d$ to $d'$ (Def. 3), we will construct an extended bisimulation $Q$ from $g$ to $g'$. A first observation is that, for any two vertices $v \in d.V, v' \in d'.V$ s.t. $(v, v') \in R$, the graphs "at" $v$ and $v'$ are value equivalent too: $d_v \equiv d'_{v'}$ (recall that $d_v$ is the graph $d$ with $v$ considered the unique input node), because $R$ is also an extended bisimulation from $d_v$ to $d_{v'}$. Then, it follows that for every label $a$, we have $e(a, d_v) \equiv e(a, d'_{v'})$ (because $e$ is bisimulation generic), and we denote with $Q_{v,a,v'}$ the extended bisimulation from $e(a, d_v)$ to $e(a, d'_{v'})$. We will construct the bisimulation from $g$ to $g'$ by essentially taking the disjoint union of $R$ and all these extended bisimulations $Q_{v,a,v'}$:

$$
\begin{aligned}
Q \stackrel{\text{def}}{=}\ & \{(S1(u, \&z), S1(u', \&z)) \mid (u, u') \in R, \&z \in \mathcal{Z}\} && (6)\\
\cup\ & \{(S2(u, a, v, w), S2(u', a, v', w')) \mid (u, u') \in R, (v, v') \in R,\\
& \quad (u, a, v) \in d.E, (u', a, v') \in d'.E, (w, w') \in Q_{v,a,v'}\} && (7)
\end{aligned}
$$

We now prove that all four conditions in Def. 3 are satisfied, hence $Q$ is an extended bisimulation. We show conditions 2, and 4 first, because they are easy to check. For 2, if $(p, p') \in Q$ and $(\&x \cdot \&z, p) \in g.I$, for $\&x \in \mathcal{X}, \&z \in \mathcal{Z}$, then, according to the definition in Sec. 5.1.1 we have $p = S1(u, \&z)$ and $(\&x, u) \in d.I$; hence $p'$ is also of the form $p' = S1(u', \&z)$, and $(u, u') \in R$. Using condition 2 of the extended bisimulation for $R$, we conclude that $(\&x, u') \in d'.I$, hence $(\&x \cdot \&z, p') \in g'.I$. For condition 4, let $\&x \in \mathcal{X}, \&z \in \mathcal{Z}$. Denoting $u = d.I(\&x)$, $u' = d.I'(\&x)$ we have $(u, u') \in R$, hence $(S1(u, \&z), S1(u', \&z)) \in Q$. It suffices now to observe that $g.I(\&x \cdot \&z) = S1(u, \&z)$ and $g'.I(\&x \cdot \&z) = S1(u', \&z)$.

We prove next that 1 holds: the proof for condition 3 is almost identical and is omitted.

For the first condition, we have to show that for any two nodes $p \in g.V$ and $p' \in g'.V$ s.t. $(p, p') \in Q$, and for any path $(p, \varepsilon^*.l, q) \in g.E$, $l \neq \varepsilon$, there exists a path $(p', \varepsilon^*.l, q') \in g'.E$ s.t. $(q, q') \in Q$. For $p, p'$ there are two cases, when both are $S1$ nodes, and when both are $S2$ nodes. We illustrate the second case only, the other being proved similarly, hence we have $p = S2(u, a, v, w)$, $p' = S2(u', a, v', w')$, and the conditions in Eq. (7) hold. For $q$ we also have two cases, and we only illustrate the case when $q$ is an $S2$ node, $q = S2(x, b, y, z)$, with $b \neq \varepsilon$. Following the path $\varepsilon^*.l$ from $p$ to $q$ in $g$, we highlight all the $S1$ nodes:

$$
p = S2(u, a, v, w) \stackrel{\varepsilon^*}{\to} S1(v_1, \&z_1) \stackrel{\varepsilon^*}{\to} S1(v_2, \&z_2) \stackrel{\varepsilon^*}{\to} \ldots \stackrel{\varepsilon^*}{\to} S1(v_n, \&z_n) \stackrel{\varepsilon^*.l}{\to} S2(x, b, y, z) = q
$$

Here $n \geq 0$. From the definitions in Sec. 5.1.1 we notice that $v = v_1$, $v_n = x$, and for each $i = 1, \ldots, n$ there exists an edge $(v_i, a_i, v_{i+1}) \in d.E$ (with $v_{n+1} \stackrel{\text{def}}{=} y$ and $a_n \stackrel{\text{def}}{=} b$). Thus, the path from $p$ to $q$ in $g$ is split into $n+1$ segments. Our proof relies on finding a similar path in $g'$, consisting of a number of segments, all labeled $\varepsilon^*$ except the last one which is labeled $\varepsilon^*.l$, and whose intermediate nodes are bisimilar to those in $g$. We construct the path in $g'$ inductively, from left to right. The segments in $g$ and $g'$ will not correspond one-to-one, but for a group of segments in $g$ we will construct a group of segments in $g'$. We consider the first segment separately, then do the induction step.

- Consider the first segment in $g$: it corresponds to a $\varepsilon^*$ path in $e(a, d_v)$ from $w$ to some output node labeled $\&z_1$. Since $e(a, d_v) \equiv e(a, d'_{v'})$ and $(w, w') \in Q_{v,a,v'}$, there exists a $\varepsilon^*$ path in $e(a, d'_{v'})$ from $w'$ to some output node also labeled $\&z_1$ (condition 3 in Def. 3): from there we have in $g'$ an $\varepsilon$-edge to $S1(v', \&z_1)$ (an $S2$-to-$S1$ edge in the definition in Sec. 5.1.1). Writing $v'_1 \stackrel{\text{def}}{=} v'$, we have the first segment in $g'$:

$$p' = S2(u', a, v', w') \xrightarrow{\varepsilon^*} S1(v'_1, \&z_1)$$

  Obviously $(S1(v_1, \&z_1), S1(v'_1, \&z_1)) \in Q$ (because of $(p, p') \in Q$ and of Eq. (7)).
- We prove now the induction step on $i$. Assume we have found $v'_i$ such that $(S1(v_i, \&z_i),\ S1(v'_i, \&z_i)) \in Q$. Let $j \in \{i, i+1, \ldots, n\}$ be the smallest number for which $a_j \neq \varepsilon$ (recall that $a_n = b \neq \varepsilon$): that is, we have $a_i = a_{i+1} = \ldots = a_{j-1} = \varepsilon$, and $a_j \neq \varepsilon$. We consider the entire group of segments:

$$S1(v_i, \&z_i) \xrightarrow{\varepsilon} S1(v_{i+1}, \&z_{i+1}) \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} S1(v_j, \&z_j)$$

  Each segment consists of a single $\varepsilon$-edge, which is an $S1$-to-$S1$ edge (see the definition in Sec. 5.1.1), hence each edge preserves the marker and we have $\&z_i = \&z_{i+1} = \ldots = \&z_j$.

  When $j < n$ we include the next segment $S1(v_j, \&z_j) \xrightarrow{\varepsilon^*} S1(v_{j+1}, \&z_{j+1})$ in our group of segments: it traverses the graph $e(a_j, v_{j+1})$. We have a $\varepsilon^*.a_j$ path in $d$, $(v_i, \varepsilon^*.a_j, v_{j+1})$ which (using the extended bisimulation condition for $R$) gives us a path $(v'_i, \varepsilon^*.a_j, v'_{j+1})$ in $d'$ with $(v_{j+1}, v'_{j+1}) \in R$. This gives us the next node on our path in $g'$, namely $S1(v'_{j+1}, \&z_{j+1})$, but we have to prove that there exists an $\varepsilon^*$ path from $S1(v'_i, \&z_i)$ to $S1(v'_{j+1}, \&z_{j+1})$. We split the path $(v'_i, \varepsilon^*.a_j, v'_{j+1})$ into a path $(v'_i, \varepsilon^*, s')$ and the last edge $(s', a_j, v'_{j+1})$. Since each $\varepsilon$-edge in $d'$ determines an $S1$-to-$S1$ edge in $g'$, we have the following path in $g'$:

$$S1(v'_i, \&z_i) \xrightarrow{\varepsilon} S1(-, \&z_i) \xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} S1(-, \&z_i) \xrightarrow{\varepsilon} S1(s', \&z_i)$$

  and from the last node we have a $\varepsilon$-edge into the input $\&z_i$ of $e(a_j, v'_{j+1})$. Now we use the fact that $e(a_j, v_{j+1})$ and $e(a_j, v'_{j+1})$ are bisimilar. The last segment in $g$, $S1(v_j, \&z_j) \xrightarrow{\varepsilon^*} S1(v_{j+1}, \&z_{j+1})$, corresponds to a $\varepsilon^*$ path in $e(a_j, v_{j+1})$ from the input node $\&z_i$ to some output node labeled $\&z_{j+1}$. Using conditions 4 then 3 in Def. 3, we find a similar $\varepsilon^*$ path in $e(a_j, v'_{j+1})$ from the input $\&z_i$ to the output $\&z_{j+1}$, and from there we have another $\varepsilon$-edge to $S1(v'_{j+1}, \&z_{j+1})$. Put together, this gives us a segment:

$$S1(v'_i, \&z_i) \xrightarrow{\varepsilon^*} S1(v'_{j+1}, \&z_{j+1})$$

  and $(S1(v_{j+1}, \&z_{j+1}), S(v'_{j+1}, \&z_{j+1})) \in Q$.

  We now consider the case $j = n$, and here we include the last segment $S1(v_n, \&z_n) \xrightarrow{\varepsilon^*.l} S2(x, b, y, z) = q$ into our group of segments. Recall (see definition 2) that $x = v_n$ and $b = a_n$, and in this case we have $\&z_i = \ldots = \&z_n$. Since $(v_i, \varepsilon^*.b, y) \in d.E$, the bisimulation gives us a path $(v'_i, \varepsilon^*.b, y')$ in $d'$, s.t. $(y, y') \in R$. As in the previous argument, we split it into $(v'_i, \varepsilon^*, s')$ and $(s', b, y')$, and find an $\varepsilon^*$ path from $S1(v'_i, \&z_i)$ to $S1(s', \&z_i)$, and from here an $\varepsilon$-edge into the $\&z_i$ input of $e(b, d'_{y'})$. Then we use the fact that $e(b, d_y)$ is bisimilar to $e(b, d'_{y'})$: the segment $\varepsilon^*.l$ in $g$ corresponds to a path in $e(b, d_y)$ from its input $\&z_i (= \&z_n)$ to the node $z$, and using conditions 4 and 1 in Def. 3 we find a path $\varepsilon^*.l$ in $e(b, d'_{y'})$ from its input $\&z_i$ to some node $z'$. This gives us the last segment in $g'$:

$$S1(v'_i, \&z_i) \xrightarrow{\varepsilon^*.l} S2(x', b, y', z') = q'$$

  This completes the proof. ☐

## 6 Expressive Power of UnCAL

We establish here three results describing UnCAL's expressive power (and, hence, UnQL's expressive power as well). First, we compare UnCAL's expressive power with a certain extension of First Order Logic (FO). When doing so we assimilate a datagraph $d = (V, E, I, O)$ with a first order structure consisting of four relations, $V, E, I, O$, of arities $1, 3, 2, 2$ respectively. We call this the *standard first order structure* of the data graph $d$.

*FO+TC* Immerman [Imm87] describes the language FO+TC: first order logic extended with transitive closure. It extends first order logic with expressions of the form $TC(\lambda\bar{x}, \bar{x}'.\varphi(\bar{x}, \bar{x}'))$, where $\varphi(\bar{x}, \bar{x}')$ is any formula in FO+TC denoting a binary relation on $k$ tuples (we assume both $\bar{x}$ and $\bar{x}'$ are $k$-tuples). Then $TC(\lambda\bar{x}, \bar{x}'.\varphi(\bar{x}, \bar{x}'))$ denotes the transitive closure of $\varphi$. Immerman showed that FO+TC can express over ordered structures precisely the queries in NLOGSPACE, the class of boolean functions computable by a nondeterministic Turing machine with $O(\log n)$ space. We establish first that UnCAL can be expressed in FO+TC.

**Theorem 2** *All UnCAL queries can be expressed in FO+TC.*

*Proof* More precisely we prove that every UnCAL query $f$ can be expressed by a formula FO+TC over the standard first order structure of its input graph. Assume $f$ expects an input data graph in $DB_{\mathcal{Y}}^{\mathcal{X}}$, i.e. with input markers $\mathcal{X} = \{\&x_1, \ldots, \&x_m\}$, and output markers $\mathcal{Y} = \{\&y_1, \ldots, \&y_n\}$. We have to define four FO+TC formulas $\varphi_V(v), \varphi_E(u, l, v), \varphi_I(x, v), \varphi_O(v, y)$ constructing the relations $V', E', I', O'$ of the output graph. The formulas are easy to construct in FO+TC extended with Skolem functions. Indeed, for structural recursion $rec(e)$ the formulas are given in Sec. 5.1. For the other constructors (depicted in Fig. 7) the corresponding formulas are straightforwardly expressible in FO, without TC. The only complication arises with the conditional $\mathsf{if} - \mathsf{then} - \mathsf{else}$ when the boolean condition is $isempty(T)$. Here we have to compute a transitive closure to determine whether any edge labeled with something different from $\varepsilon$ is reachable from $T$'s root.

Once we have the three formulas $\varphi_E, \varphi_I, \varphi_O$ we have to eliminate the Skolem functions. This is done with a standard technique, described for example in [AHV95]. The idea is to replace a Skolem term, say $S2(u, a, v, w)$ with a 4-tuple $(u, a, v, w)$. For example the triple:

$$(S1(u, \&z), \varepsilon, S2(u, a, v, w))$$

becomes now a seven-tuple:

$$(u, \&z, \varepsilon, u, a, v, w)$$

There are two problems however: we create non-homogeneous relations with tuples of unequal arity, and we need to differentiate between tuples from different Skolem terms with the same arity but different function names (for example a 4-tuple $(x, y, z, u)$ could come either from $S2(x, y, z, u)$, or from $S1(S1(S1(x, y), z), u)$). Both problems are solved by padding all the tuples in a collection to a common length, with two distinct constants $\alpha \neq \beta$: a tuple $(x_1, \ldots, x_k)$ will be padded to $(x_1, \ldots, x_k, \alpha, \ldots, \alpha, \beta, \ldots, \beta)$. The position where $\alpha$ changes to $\beta$ encodes the names of the Skolem functions used in the tuple. Details can be found in [AHV95]. Note that the Skolem terms used in the TC operators do not create additional problems: by contrast Skolem terms in the head of datalog rules construct new nodes, and can lead to non terminating computations. □

Using Immerman's result [Imm87] we obtain immediately:

**Corollary 1** *All UnCAL queries are computable in NLOGSPACE (and, hence, in PTIME).*

*UnQL on Relational Data* As explained in Sec. 2, relational data can be encoded as trees, like:

```
{ student: { id: "123", name: "L. Simpson", age: "19" },
  student: { id: "345", name: "T. Quail",   age: "22" },
  student: { id: "789", name: "E. Vader",   age: "32" },
  course:  { cid: "294", title: "An Introduction to Java" },
  course:  { cid: "552", title: "Advances in Databases" },
  enrolls: { id: "345", cid: "294" },
  enrolls: { id: "789", cid: "294" } }
```

This is a tree of depth two, encoding an instance `db` of the relational schema `student(id, name, age)`, `course(cid, title)`, `enrolls(id, cid)`. In general, for every relational schema $S$, each instance $db$ of $S$ can be encoded as a tree datagraph $t$: we call $d$ the *tree encoding of db*. Note that in $t$ both the relation names and the attribute names in $S$ occur as labels. Recall that in UnCAL we move values from leaves to edges (see Fig. 4), hence tree encodings of relational databases have depth three, rather than two.

Hence UnQL can be used to query relational data. It is easy to see that it can express all relational algebra queries. For example the following query expresses a join:

```
select {class T}
where {student: { id: ID, name: "T. Quail"}} in db,
      {enrolls: { id: ID', cid: CID'}} in db,
      {course:  { cid: CID, title: T}} in db,
      ID = ID', CID = CID'
```

The query computes all courses taken by `T. Quail`. Union is already a primitive in UnQL, while for difference one uses the `isEmpty` predicate, as in the following query returning all student names taking course `294` without taking course `552` (notice that we use the variable `ID` in several places, to express joins more concisely):

```
select {result: N}
where {student: { id: ID, name N}, enrolls: { id: ID, cid: "294"} } in db,
      isEmpty( select { some } where { enrolls: { id: ID, cid: "552" }} in db)
```

A question arises: do we get more expressive power than the relational algebra ? An immediate answer is yes, since we can express queries returning non-flat results, like grouping students by age:

```
select {result:{age:A, students:(select { name: N}
                                 where {student: {name: N, age: A}} in db)}}
where  {student: { age: A}} in db
```

Another source of additional power comes from UnQL's ability to express polymorphic queries, with respect to the relational database schema. For example the following query returns all attributes of the student `"Smith"`, except its `id`:

```
select {L : V}
where   {student: {name: "Smith", L: V}} in db, L != "id"
```

This is polymorphic since it works regardless of the input relational schema. For example if we evaluate this query on the tree representation of a table `student(id, name, age, office, phone, email)`, then we retrieve the attributes `name, age, office, phone, email`.

More interestingly, we can write UnQL queries that ignore the schema. For example the following query returns all strings in the database containing the substring `"Java"`:

```
select {result: S}
where {_*: S} in db, match("Java", S)
```

Apart from its ability to bypass the schema, the question still remains whether UnQL can express more queries than the relational algebra, in a setting in which both the input and the output relational schemas are fixed. The answer is no, and we state this in the more general framework of UnCAL queries. More precisely, let $S$ be an input relational schema, and $S'$ be an output relational schema with a single relation. Recall [AHV95] that in relational databases a query $Q$ is always defined in connection with an input schema $S$ and an output schema $S'$: $Q$ maps $S$-instances to $S'$-instances. Consider now an UnCAL query $f$, and suppose it enjoys the following semantic property:

– For every tree encoding $t$ of an $S$-instance, $f(t)$ is a tree encoding of some $S'$-instance.

Then, we can show that there exists a FO query, $Q$, from $S$-instances to $S'$-instances, which is "equivalent" to $f$ in the following sense: for every $S$-instance $db$, $f$ maps the tree encoding of $db$ into the tree encoding of $Q(db)$. For a trivial illustration, consider the UnQL query $f$ above, retrieving all strings in the database containing

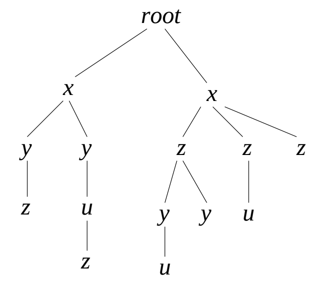


**Fig. 13** Illustration for the proof of Lemma 1.

the substring `"Java"`, and assume the relational schema $S$ to be `student(id, name, age), course(cid, title), enrolls(id, cid)`, and the relational output schema $S'$ to consists of a single relation, with a single attribute. Then $f$ can be expressed as the following union of seven formulas (one for each attribute, in each relation):

$$\begin{aligned} Q = \; & \{x \mid student(x, y, z), match(\text{"Java"}, x)\} \cup \\ & \{y \mid student(x, y, z), match(\text{"Java"}, y)\} \cup \\ & \ldots \\ & \{y \mid enrolls(x, y), match(\text{"Java"}, y)\} \end{aligned}$$

It is important to note that, in general, $Q$ depends not just on $f$, but also on both the input schema $S$ and the output schema $S'$. In the remainder of this section we will state and prove this result formally.

We start with a graph theoretic discussion. Given a graph $G$, we denote a path $p$ as $x_0.x_1.x_2 \ldots x_k$, where $x_0, \ldots, x_k$ are $p$'s nodes. The length of $p$ is $k$ (i.e. the length is the number of edges). The path is *simple*, if all nodes $x_0, x_1, \ldots, x_k$ are distinct. For $m \geq 1$, we say that $p$ has at most $m$ repetitions, if each node $x_i$ occurs at most $m$ times in the path, for $i = 0, \ldots, k$. In particular a path is simple if and only if it has at most 1 repetition. We call a graph $G$ *$k$-short* if all its simple paths have length $\leq k$. For example a tree of depth $k$ is $k$-short, hence the tree encodings of relational instances are 3-short. As another example, the graph consisting of a single cycle with $k$ nodes and $k$ edges is $(k-1)$-short. Observe that in a $k$-short graph one can compute its transitive closure by inspecting all paths of length $\leq k$. We establish the following graph theoretic lemma:

**Lemma 1** *Let $G$ be $k$-short, and let*[9] $\psi(m, k) \stackrel{\text{def}}{=} \frac{m^{k+2}-1}{m-1} - 2$. *Then, for any $m \geq 1$, every path with at most $m$ repetitions in $G$ has length $\leq \psi(m, k)$.*

*Proof* Let $p = x_0.x_1 \ldots x_n$ be a path with at most $m$ repetitions. We will refer in the sequel to $x_i$ as an "occurrence" in $p$: that is, for $i \neq j$, $x_i, x_j$ may denote the same node, but we will treat them as different occurrences. We will arrange the $n+1$ occurrences $x_0, x_1, \ldots, x_n$ in a tree $T$, with $n+2$ nodes, such that each node, except for the root, is labeled with exactly one occurrence $x_i$ (the root is unlabeled). We describe the tree top-down. Once a node is constructed, its subtrees are obtained by processing a subsequence of $p$. Initially we construct the root, and derive its children from the entire sequence $p$, as follows. Consider the first node in $p$, $x_0$. There are (at most) $m$ occurrences in $p$ denoting the same node as $x_0$: $x_0 = x_{i_1}, x_{i_2}, x_{i_3}, \ldots, x_{i_{m'}}$, for $m' \leq m$. They split $p$ into $m'$ subsequences, best visualized as:

$$x_0(x_1 \ldots x_{i_1-1}), x_{i_1}(x_{i_1+1} \ldots x_{i_2-1}), x_{i_2}(x_{i_2+1} \ldots x_{i_3-1}), \ldots x_{i_{m'}}(x_{i_{m'}+1} \ldots x_n)$$

We then create $m'$ children for the root node, as follows. For $j = 1, \ldots, m'$, child $j$ will be labeled with occurrence $x_{i_j}$, and its subtrees will be constructed from the the subsequence to the right of $x_{i_j}$, i.e. $x_{i_j+1}.x_{i_j+2} \ldots x_{i_{j+1}-1}$, in a recursive manner.

[9] As usual, for $m = 1$, $\psi$ is defined to be $\psi(1, k) \stackrel{\text{def}}{=} (k+2) - 2 = k$.

This completes the description of $T$. For a simple illustration of this construction, consider the sequence $x.y.z.y.u.z.x.z.y.u.y.z.u.z$. The corresponding tree is

$$\mathit{root}(x(y(z), y(u(z))), x(z(y(u), y), z(u), z))$$

and is also illustrated in Fig. 13. Let us inspect some properties of $T$. Obviously, each node in $T$ has at most $m$ children. Further, all children with the same parent are labeled with the same node in $G$ (corresponding to different occurrences in $p$). In our example, the root's children are labeled $x, x$; the first $x$'s children are labeled $y, y$; the second $x$'s children are labeled $z, z, z$. Next, each edge in $T$ corresponds to an edge in $G$: to illustrate, in our example we have a node labeled $x$ having children labeled $z$ and, indeed, there exists an edge from $x$ to $z$ in the graph (because $x.z$ occur adjacent in the path). Finally, if a node in $T$ is labeled with occurrence $x_i$, then the node $x_i$ does not occur in any descendant of that node. In our example, a node labeled $x$ does not have any descendant also labeled $x$. This all implies that each path in $T$ from the root to some leaf corresponds to a simple path in $G$. Hence, the depth of $T$ is at most $k+1$ (we had to add 1 for the root), and $T$ has at most

$$1 + m + m^2 + \ldots + m^{k+1} = \frac{m^{k+2} - 1}{m - 1}$$

nodes. Recall that this is $n + 2$. Hence, the length of $p$ is $n = \frac{m^{k+2}-1}{m-1} - 2$. □

Now we can prove a remarkable property enjoyed by all UnCAL queries.

**Proposition 5** *For any UnCAL query $f$ there exists a function $\varphi(k)$ s.t. if the input data graph $d$ is $k$-short, then the output data graph $f(d)$ is $\varphi(k)$-short.*

*Proof* (Sketch) Most tree constructors only add a constant to the lengths of simple paths: for example $d \cup d'$ and $\{l : d\}$ increase it by 1. For $\mathit{cycle}_{\mathcal{Z}}(d)$, if $d$ is $k$-short and $\mathcal{Z}$ has $m$ markers, then $\mathit{cycle}_{\mathcal{Z}}(d)$ is $m \times (k+1)$ short. To see that, observe that there are exactly $m$ input nodes $x_1, \ldots, x_m$ in $d$. Given a simple path $p$ in $\mathit{cycle}_{\mathcal{Z}}(d)$, each $x_i$ occurs at most once in $p$, hence the $m$ input nodes split $p$ into (at most) $m+1$ simple paths fragments of length $\leq k+1$ each (the $+1$ represents the $\varepsilon$ edge connecting an output node to an input node in $\mathit{cycle}_{\mathcal{Z}}(d)$, which is at the end of each path fragment), except for the last fragment whose length is $\leq k$. It follows that $p$ has length at most $k + m(k+1)$.

The most interesting case is $\mathit{rec}_{\mathcal{Z}}(e)(d)$. Assume that $e$ increases the length by a function $\varphi$. Consider a simple path $p$ in $\mathit{rec}_{\mathcal{Z}}(e)(d)$ (Fig. 10 might help visualize such a path).. Let

$$S1(u_0, \&x_0), S1(u_1, \&x_1), \ldots, S1(u_n, \&x_n)$$

be all nodes of type $S1$ occurring in $p$. These nodes are distinct, hence $p' = u_0.u_1.\ldots.u_n$ is a path in $d$ with at most $m$ repetitions (because there are at most $m$ distinct markers among $\&x_1, \ldots, \&x_n$). It follows that the length of $p'$ is $n \leq \psi(m, k)$. The path $p$ can be longer: between any two $S1$ nodes we can have a path of length $\leq \varphi(k) + 2$ (the $+2$ accounts for the two gluing $\varepsilon$ edges), while before the first $S1$ node, and after the last $S1$ node there can be another path, each of length $\leq \varphi(k) + 1$. It follows that $p$'s length is at most $2(\varphi(k) + 1) + \psi(m, k)(\varphi(k) + 2)$. □

**Theorem 3** *Let $f$ be an UnCAL expression. Then, for any $k \geq 1$, there exists a FO formula $\varphi_k$ which computes $f$ on all $k$-short graphs.*

*Proof* We first apply Theorem 2 and obtain a FO+TC formula for $f$. Since all occurrences of TC are applied to a $k'$-short graph, for some $k'$, we can unfold the TC operator $k'$ times and express it in FO. □

From this it is straightforward to derive:

**Corollary 2 (Conservativity)** *Let $S$ be an input relational database schema, and $S'$ an output relational schema. If an UnCAL query $f$ maps tree encodings of $S$-instances into tree encodings of $S'$-instances, then $f$ can be expressed in the relational calculus.*

*Proof* We will construct a query $Q$ in the relational calculus "simulating" $f$. $Q$ operates in three steps. In Step 1, given an $S$-instance $d$, it constructs a first order structure representing the tree $t$ encoding $d$: obviously, this graph is 3-short. This query is expressed in the relational calculus extended with Skolem functions. Note that this step uses the schema $S$. In Step 2 $Q$ applies the FO formula $\varphi_3$ from Theorem 3, to obtain the first order structure for $f(t)$. In Step 3, it constructs from $f(t)$ the relational output, knowing that $f(t)$ is a tree encoding of some $S'$ instance; notice that this step uses the schema $S'$. The three formulas are then composed, and finally we eliminate Skolem functions, as outlined in the proof of Theorem 2. Only steps 1 and 3 are new. We illustrate them by example only, the general case should be obvious. For step 1, assume the relational schema to be `course(cid, title)`, `enrolls(id, cid)`. Then the following two queries construct the relations $E$ and $I$ of the tree $t$ ($O = \emptyset$, and we omit $V$ which simply consists of all nodes mentioned in $E$, and is straightforward):

$$\begin{aligned}
E = \; & \{(\mathit{Root}(), \text{“}\mathit{course}\text{”}, F1(x,y)) \mid \mathit{course}(x,y)\} \cup \\
& \{(F1(x,y), \text{“}\mathit{cid}\text{”}, F2(x,y)) \mid \mathit{course}(x,y)\} \cup \\
& \{(F2(x,y), x, F3(x,y)) \mid \mathit{course}(x,y)\} \cup \\
& \{(F1(x,y), \text{“}\mathit{title}\text{”}, F4(x,y)) \mid \mathit{course}(x,y)\} \cup \\
& \{(F4(x,y), y, F5(x,y)) \mid \mathit{course}(x,y)\} \cup \\
& \{(\mathit{Root}(), \text{“}\mathit{enrolls}\text{”}, G1(x,y)) \mid \mathit{enrolls}(x,y)\} \cup \\
& \{(G1(x,y), \text{“}\mathit{id}\text{”}, G2(x,y)) \mid \mathit{enrolls}(x,y)\} \cup \\
& \{(G2(x,y), x, G3(x,y)) \mid \mathit{enrolls}(x,y)\} \cup \\
& \{(G1(x,y), \text{“}\mathit{cid}\text{”}, G4(x,y)) \mid \mathit{enrolls}(x,y)\} \cup \\
& \{(G4(x,y), y, G5(x,y)) \mid \mathit{enrolls}(x,y)\} \\
I = \; & \{(\&, \mathit{Root}())\}
\end{aligned}$$

Both queries are in the relational calculus extended with the Skolem functions $\mathit{Root}, F1, F2, \ldots, G5$. Note that the values are placed on the last edges, as illustrated in Fig. 4. Step 3 is the reversed direction, and is easier. Assume the output schema $S'$ to be `result(A, B)`; hence, Step 2 returns a first order structure $(V', E', I', O')$ representing a tree encoding an $S'$ instance. Then we can construct the binary relation `result(A,B)` by:

$$\{(x,y) \mid I'(\&, r), E'(r, \text{“}\mathit{result}''', u), E'(u, \text{“}A'', v), E'(v, x, \_), E'(u, \text{“}B'', w), E'(w, y, \_)\}$$

This completes the proof. $\square$

Based on our earlier discussion, the converse also holds: all relational calculus queries can be expressed in UnCAL.

## 7 Optimizations

We have shown in Sec.5.1.4 some simple equations which can be used by an optimizer based on rewriting rules. Here we show a more powerful equation, which can be used to optimize mediator systems. Technically this involves rewriting the composition of two structural recursion functions, i.e. $rec(e_2) \circ rec(e_1)$. Recall that both recursion bodies $e_1$ and $e_2$ are functions expecting a label and a tree argument: $e_1(l,t)$, $e_2(l,t)$.

**Theorem 4** *(1) Assume that $e_2(l,t)$ does not depend on the argument $t$. Then:*

$$rec(e_2) \circ rec(e_1) = rec(rec(e_2) \circ e_1)$$

*(2) For any $e_1, e_2$ we have:*

$$rec(e_2) \circ rec(e_1) = rec(\lambda(l,t).\, rec(e_2)(e_1(l,t) @ rec(e_1)(t)))$$

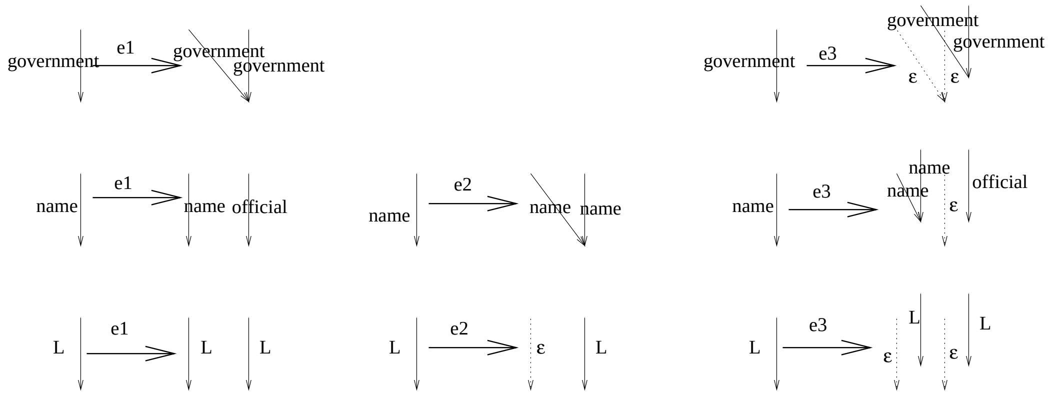


**Fig. 14** Illustration for optimizing structural recursion.

Both equations offer powerful optimizations. The left hand side describes two structural recursions computed in sequence: evaluating $rec(e_2) \circ rec(e_1)(t)$ directly requires us to compute the intermediate graph $t' = rec(e_1)(t)$. As we argue below, this intermediate graph is often as large as the input graph, and expensive to compute. Moreover, the second structural recursion, $rec(e_2)(t')$ often returns only a small result, making the cost for computing $t'$ even more of a waste. By contrast, the right hand side of both equations consists of a single structural recursion, albeit with a more complex body: a direct evaluation of this function makes it unnecessary to compute the intermediate result.

The intuition in (1) is rather simple. The first recursion, $rec(e_1)$, replaces every edge $l$ with a new graph[10] $e_1(l)$. Thus, the intermediate graph consists of many "fragments" $e_1(l)$. Next $rec(e_2)$ traverses the resulting graph and replaces each edge $l'$ with the graph $e_2(l')$. The two steps can be achieved in a single step, by replacing each edge $l$ with a new graph $e_3(l)$: the latter is $e_1(l)$ in which every edge $l'$ is replaced with $e_2(l')$, i.e. $e_3(l) = rec(e_2)(e_1(l))$. While this is intuitive, the details are tricky: $rec(e_2)(e_1(l))$ technically is a recursion on an unfinished graph, since $e_1(l)$ is only a fragment. What makes things work is the correct interaction between recursion and markers – since the markers in $e_1(l)$ tell how this fragment is connected to the others.

The explanation of (2) is the following. In (1) we needed $e_2(l, t)$ not to depend on $t$, since we apply it on a fragment $e_1(l)$. If $e_2(l, t)$ does depend on $t$, then we need to replace the fragment $e_1(l)$ with the entire graph – at least following edges forward. This is done by substituting $e_1(l) @ rec(e_1)(t)$ for $e_1(l)$. After that, the intuition in (2) is the same as in (1).

Before proving the theorem, we illustrate with two examples.

*Example 1* Assuming our database `d` in Fig 2 we construct a view in which every `name` attribute under `government` is relabeled `official`. The view is computed by `d'` = `f1(d)` where:

```
sfun f1({government:T}) = {government:g1(T)}       sfun g1({name:T}) = {official:g1(T)}
   | f1({L:T})          = {L:f1(T)}                   | g1({L:T})     = {L:g1(T)}
```

Next, the following query on the view asks for all `names` in the view:

```
sfun f2({name:T}) = {name:g2(T)}        sfun g2({L:T}) = {L:g2(T)}
   | f2({L:T})    = f2(T)
```

(We could have defined `f2` simpler by replacing the first clause with `sfun f2({name : T}) = {name : T}`, but then its body would use `T` directly, violating the condition on $e_2$ in Theorem 4. Hence we modified `f2` to copy `T` with the aid of the function `g2`.) The answer to our query is obtained as `f2(d')`. Obviously that answer can be also obtained directly from the database, by returning all `names`, except those under a `government` label. Theorem 4 allows us to derive the optimized query automatically. First we express both as `rec(e1)`, `rec(e2)` (See also Fig. 14):

[10] Assume, for illustration purposes, that $e_1(l, t)$ does not depend on $t$.

```
e1(L) = case L of
        government : (&z11 := {government: &z12},  &z12 := {government: &z12})
        name :       (&z11 := {name: &z11},        &z12 := {official: &z12})
        _ :          (&z11 := {L: &z11},           &z12 := {L: &z12})

e2(L) = case L of
        name : (&z21 := {name:&z22},  &z22 := {name: &z22})
        _ :    (&z21 := &z22,         &z22 := {L: &z22})
```

The view $d'$ is defined as the first component of $rec(e1)(d)$, while the query answer is the first component of $rec(e2)(d')$, which is the first component of $rec(e2)(rec(e1)(d))$. The optimization in Theorem 4 allows us to compute this as the first component of $rec(rec(e2) \circ e1)(d)$. Let us look at the body of this recursion, $e3 \stackrel{\text{def}}{=} rec(e2) \circ e1$ (See also Fig. 14):

```
e3(L) =
  case L of
    government: (&z11.&z21 := &z12.&z21,                 &z12.&z21 := &z12.&z21,
                 &z11.&z22 := {government: &z12.&z22},  &z12.&z22 := {government: &z12.&z22})
    name:       (&z11.&z21 := {name: &z11.&z22},        &z12.&z21 := &z12.&z21,
                 &z11.&z22 := {name: &z11.&z22},        &z12.&z22 := {official: &z12.&z22})
    L:          (&z11.&z21 := &z11.&z21,                 &z12.&z21 := &z12.&z21,
                 &z11.&z22 := {L: &z11.&z22},           &z12.&z22 := {L: &z12.&z22})
```

Note that $e3$ returns a tree with four markers. Hence, we can express $rec(e)$ as four mutually recursive functions, named `f3, g3, h3, k3`:

```
sfun f3({government:T}) = g3(T)               sfun g3({L: T}) = g3(T)
   | f3({name:T})       = {name: h3(T)}
   | f3({L:T})          = f3(T)

sfun h3({government: T})= {government: k3(T)}  sfun k3({name: T}) = {official: k3(T)}
   | h3({L: T})         = {L: T}                   | k3({L: T})    = {L: T}
```

Finally we observe that `g3(t) = {}`, for any tree `t`: this is because it only calls itself recursively without ever constructing any result. Hence we can farther simplify the functions to:

```
sfun f3({government:T}) = {}
   | f3({name:T})       = {name: h3(T)}
   | f3({L:T})          = f3(T)

sfun h3({government: T})= {government: k3(T)}  sfun k3({name: T}) = {official: k3(T)}
   | h3({L: T})         = {L: T}                   | k3({L: T})    = {L: T}
```

It is interesting to compare `f3(d)` to our informal description earlier on how to compute the query directly from the database: retrieve all `names` except those under `government`. Namely `f3(d)` does the same, except that it replaces the value `v` under `name` with the view `f1(v)`. The reader may check that this is, indeed, the correct answer for our query.

*Example 2* We consider now the example in Sec. 2: the functions `f5, g5` define a view in where all areas are converted to $mi^2$, while the query `Q` returns the land area of France. Here the task is to simplify a structural recursion function `f5` followed by a `select-where` query:

```
query Q :=
  select {area: A} where {country: {name: "France", area.land: A}} in f5(db)
```

First we write `Q` as a structural recursive function, using the rewriting in Sec. 2. We only apply one step of the rewriting:

```
sfun h5({country: C}) =  bh5(C)

fun bh5(C) =  /* body of h5 */
       select {result: A}  where {name: "France", area.land: A} in C
```

That is, the query `Q` is obtained as the result of `h5(f5(d))`.

We now rewrite `f5, g5` as $rec(e1)$, and `h5` as $rec(e2)$, where:

```
e1(L, T) = case L of
            area:     (&z1 := {area: &z2}, &z2 := {area: &z2})
            isInt(L): (&z1 := {L: &z1},    &z2 := {(0.3861*L): &z2})
            _ :       (&z1 := {L: &z1},    &z2 := {L: &z2})
e2(L, C) = case L of
            country: bh5(C)
            _ :      {}
```

Our goal is to compute $rec(e2) \circ rec(e1)$: by Theorem 4, this is equivalent to

$$rec(\lambda(l,t).(rec(e2)(e1(l,t) @ rec(e1)(t)))) = rec(\lambda(l,t).\, rec(e2) \circ e1')$$

where $e1'(l,t) \stackrel{\text{def}}{=} e1(l,t) @ rec(e1)(t)$. Let us first examine $e1'$. It is obtained by unfolding the recursion in $rec(e1)$ once, i.e. replacing the markers $\&z1, \&z2$ with the recursive function calls $f5(T), g5(T)$:

```
e1'(L, T) = case L of
            area:     (&z1 := {area: g5(T)},  &z2 := {area: g5(T)})
            isInt(L): (&z1 := {L: f5(T)},     &z2 := {(0.3861*L): g5(T)})
            _ :       (&z1 := {L: f5(T)},     &z2 := {L: g5(T)})
```

Next we compute $e3(l,t) \stackrel{\text{def}}{=} rec(e2) \circ e1'(l,t)$

```
e3(L, T) = case L of
            country: (&z1 := bh5(f5(T)),  &z2 := bh5(g5(T)))
            _ :      (&z1 := {},          &z2 := {})
```

$rec(e3)(db)$ translates into two structural recursion functions `k, k'`, of which we only need the first one (the second one is unnecessary since they are not recursive). Thus we have rewritten the query `Q` into `Q1`:

```
query Q1 :=
  let sfun k({country:C}) :=
                select {area: A} where {name: "France", area.land: A} in f5(C)
  in k(db)
```

We now repeat the same process again on the inner `select-where` query. After several such steps we obtain the simplified query `Q'` in Sec. 3.

The rest of this section is dedicated to the proof of the theorem.

Let us start by looking more carefully at the types in Theorem 4 (1):

$$\begin{aligned} e_1 &: Label \times DB_{\mathcal{Y}} \to DB^{\mathcal{Z}_1}_{\mathcal{Z}_1} \\ e_2 &: Label \times DB_{\mathcal{Y}\cdot\mathcal{Z}_1} \to DB^{\mathcal{Z}_2}_{\mathcal{Z}_2} \end{aligned}$$

$$\begin{aligned} rec(e_1) &: DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X}\cdot\mathcal{Z}_1}_{\mathcal{Y}\cdot\mathcal{Z}_1} \\ rec(e_2) &: DB^{\mathcal{X}\cdot\mathcal{Z}_1}_{\mathcal{Y}\cdot\mathcal{Z}_1} \to DB^{\mathcal{X}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2}_{\mathcal{Y}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2} \\ rec(e_2) \circ rec(e_1) &: DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2}_{\mathcal{Y}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2} \end{aligned}$$

$$\begin{aligned} rec(e_2) &: DB^{\mathcal{Z}_1}_{\mathcal{Z}_1} \to DB^{\mathcal{Z}_1\cdot\mathcal{Z}_2}_{\mathcal{Z}_1\cdot\mathcal{Z}_2} \\ rec(e_2) \circ e_1 &: Label \times DB_{\mathcal{Y}} \to DB^{\mathcal{Z}_1\cdot\mathcal{Z}_2}_{\mathcal{Z}_1\cdot\mathcal{Z}_2} \\ rec(rec(e_2) \circ e_1) &: DB^{\mathcal{X}}_{\mathcal{Y}} \to DB^{\mathcal{X}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2}_{\mathcal{Y}\cdot\mathcal{Z}_1\cdot\mathcal{Z}_2} \end{aligned}$$

Recall that $\cdot$ is associative, i.e. $(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z)$. This shows that both expressions $rec(e_2) \circ rec(e_1)$ and $rec(rec(e_2) \circ e_1)$ have the same type. As we stated repeatedly, expressions in UnCAL are not polymorphic, but have to state their expected makers explicitly: the two occurrences of $rec(e_2)$ in the equation have different types. Written formally, the equation is:

$$rec_{\mathcal{X}\cdot\mathcal{Z}_1,\mathcal{Y}\cdot\mathcal{Z}_1,\mathcal{Z}_2}(e_2) \circ rec_{\mathcal{X},\mathcal{Y},\mathcal{Z}_1}(e_1) = rec_{\mathcal{X},\mathcal{Y},\mathcal{Z}_1\cdot\mathcal{Z}_2}(rec_{\mathcal{Z}_1,\mathcal{Z}_1,\mathcal{Z}_2}(e_2) \circ e_1)$$

Now we prove Theorem 4.

*Proof* It is possible to prove both statements directly, by examining the graphs on both sides of the equation and proving them to be value equivalent. Such a proof would be complete, but it becomes quite technical and nonintuitive. Instead, we prefer an alternative proof method: we show that both equations hold on finite trees, by induction on the tree structure, then use Fact 1 to argue that they also hold for graphs. We caution the reader that this proof is somewhat restrictive, since in the Appendix we only substantiate Fact 1 for a subclass of UnCAL queries (for positive ones): still, we prefer it over the complete, but highly technical direct proof.

To be precise, we need to prove that the equalities hold on any finite forest $t$. It actually suffices to check that for finite trees $t'$ only: this is because the value of $rec(e)(t)$ on a forest $t$ is fully determined by its value on each component tree (recall $rec(e)(t \oplus t') = rec(e)(t) \oplus rec(e)(t')$, Prop. 3). Thus, we will prove the following equations, by induction on the finite tree $t'$:

$$rec(e_2)(rec(e_1)(t')) = rec(rec(e_2) \circ e_1)(t') \tag{8}$$
$$rec(e_2)(rec(e_1)(t')) = rec(\lambda(l,t).\, rec(e_2)(e_1(l,t) @ rec(e_1)(t)))(t') \tag{9}$$

We start with (8).

- When $t' = \{\}$ then both sides are $\{\}$.
- When $t' = \{l : t\}$, we have:

$$\begin{aligned} rec(e_2)(rec(e_1))(\{l : t\}) &= rec(e_2)(e_1(l,t) @ rec(e_1)(t)) \\ &= rec(e_2)(e_1(l,t)) @ rec(e_2)(rec(e_1)(t)) \\ &= (rec(e_2) \circ e_1)(l,t) @ rec(rec(e_2) \circ e_1)(t) \\ &= rec(rec(e_2) \circ e_1)(\{l : t\}) \end{aligned}$$

We used here Eq.(4) in Prop. 3, and the fact that Eq.(8) holds on $t$, by induction hypothesis.
- When $t' = t_1 \cup t_2$, we have:

$$\begin{aligned} rec(e_2)(rec(e_1))(t_1 \cup t_2) &= rec(e_2)(rec(e_1)(t_1) \cup rec(e_1)(t_2)) \\ &= rec(e_2)(rec(e_1)(t_1)) \cup rec(e_2)(rec(e_1)(t_2)) \\ &= rec(rec(e_2) \circ e_1)(t_1) \cup rec(rec(e_2) \circ e_1)(t_2) \\ &= rec(rec(e_2) \circ e_1)(t_1 \cup t_2) \end{aligned}$$

Here we used the induction hypothesis for $t_1$ and $t_2$.
- When $t' = \&y$ (an output marker), then let the two sets of markers be $\mathcal{Z}_1 = \{\&z_{11}, \ldots, \&z_{1p}\}$, and $\mathcal{Z}_2 = \{\&z_{21}, \ldots, \&z_{2q}\}$. Then the following results directly from the marker types:

$$\begin{aligned} rec(e_2)(rec(e_1)(\&y)) &= rec(e_2)(\&z_{11} := \&y \cdot \&z_{11}, \ldots, \&z_{1p} \cdot \&z_{1p}) \\ &= (\&z_{11} \cdot \&z_{21} := \&y \cdot \&z_{11} \cdot \&z_{21}, \&z_{11} \cdot \&z_{22} := \&y \cdot \&z_{11} \cdot \&z_{22}, \\ &\qquad \ldots, \&z_{1p} \cdot \&z_{2q} := \&y \cdot \&z_{1p} \cdot \&z_{2q}) \\ &= rec(rec(e_2) \circ e_1)(\&y) \end{aligned}$$

We prove now (9). The cases when $t'$ is $\{\}, t_1 \cup t_2$ or $\&y$ are handled similarly as in the previous proof, and are omitted. The interesting case is when $t' = \{l : t\}$. Here the key observation is that the right hand side does a single iteration, without a recursive call on $t$. That is, denoting $e_3 = \lambda(l,t).\, rec(e_2)(e_1(l,t) @ rec(e_1)(t))$, the right hand side is $rec(e_3)(\{l : t\}) = e_3(l,t) @ rec(e_3)(t) = e_3(l,t)$. This is because in the right hand side the recursion on the remaining tree is hidden in the expression $e_3$. Technically, this follows from the fact that there are no $\mathcal{Z}_1 \cdot \mathcal{Z}_2$ output markers occurring in $e_3$ (this is because the $\mathcal{Z}_1$ output markers in $e_1(l,t) @ rec(e_1)(t)$ where "consumed" by the append operation). To see the technical details, we illustrate the types for the case when $\mathcal{X} = \{\&\}$ and $\mathcal{Y} = \{\}$, i.e. when the entire recursion is applied to trees. Then:

$$\begin{aligned} e_1 &: Label \times DB \to DB^{\mathcal{Z}_1}_{\mathcal{Z}_1} \\ e_2 &: Label \times DB \to DB^{\mathcal{Z}_2}_{\mathcal{Z}_2} \end{aligned}$$

$$\begin{aligned} rec(e_1) &: DB \to DB^{\mathcal{Z}_1} \\ \lambda(l,t).e_1(l,t) @ rec(e_1) &: Label \times DB \to DB^{\mathcal{Z}_1} \subset DB^{\mathcal{Z}_1}_{\mathcal{Z}_1} \\ e_3 = \lambda(l,t).\, rec(e_1(l,t) @ rec(e_1)) &: Label \times DB \to DB^{\mathcal{Z}_1 \cdot \mathcal{Z}_2} \subset DB^{\mathcal{Z}_1 \cdot \mathcal{Z}_2}_{\mathcal{Z}_1 \cdot \mathcal{Z}_2} \\ rec(\lambda(l,t).\, rec(e_2)(e_1(l,t) @ rec(e_1)(t))) &: DB \to DB^{\mathcal{Z}_1 \cdot \mathcal{Z}_2} \end{aligned}$$

The types illustrate that $e_3$ doesn't have any occurrences of the output markers $\mathcal{Z}_1 \cdot \mathcal{Z}_2$.

The case $t' = \{l : t\}$ follows now immediately:

$$rec(e_2)(rec(e_1)(\{l : t\})) = rec(e_2)(e_1(l,t) @ rec(e_1)(t))$$

$$\begin{aligned} rec(e_3)(\{l : t\}) &= e_3(l,t) \\ &= rec(e_2)(e_1(l,t) @ rec(e_1)(t)) \end{aligned}$$

This concludes the proof. □

## 8 Getting Practical: How to Evaluate UnQL

UnQL can be evaluated in two ways: top-down and bulk. We describe here these two strategies. Both assume that the UnQL query has been translated into UnCAL, then essentially use the two semantics for structural recursion in Sec. 5.1 (recursive and bulk). We will illustrate them both on the query `Q7` (a variation on query `Q1` from Sec. 2):

```
query Q7 :=
   select {result: E}
   where  {country: {name: "France",  *.ethnicGroup: E}} in db
```

which, translated into structural recursion becomes:

```
query Q7' :=
  let sfun f7({ country : C }) =
       let sfun g7({ name: "France" }) =
            let sfun h7({ ethnicGroup: E}) = {result: E} U h7(E)
                   | h7({ L: E} = h7(E)
            in h7(C)
       in g7(C)
  in f7(db)
```

Notice that we go back here to the original data model in Sec. 2, where data values are stored on leaves rather than edges.

*Top-down Evaluation* Given an UnCAL query $Q$, this strategy evaluates its operators in the order in which they appear in the query. All constructors are evaluated directly. Structural recursion is evaluated using the recursive algorithm in Fig. 12. For example, our query `Q7'` is evaluated on some input graph `db` as follows. First apply the recursive evaluation algorithm to compute `f7(db)`. This algorithm needs to traverse *db* recursively (top-down) and evaluate the recursion body on each edge in the graph. The recursion body is the function `g6`, which is another structural recursion: hence this function is evaluated recursively on the graph starting at the current node, etc. Each recursive function is computed using the algorithm in Fig. 12.

The advantage of this method is its simplicity. On tree data, this method proceeds like any interpretor of a functional programming language. On cyclic data, the algorithm in Fig. 12 adds some more cleverness in that it avoids chasing infinite loops. The disadvantage of this method is that no further optimizations are possible: all operators are evaluated precisely in the order in which they occur in the UnCAL query. However, for small or medium data sets, like XML documents, this strategy results in acceptable performance.

*Bulk Evaluation* This consists in the construction of a query plan, which can be further optimized before evaluation. The plan starts from a ternary input relation $E(x, l, y)$, and constructs a ternary output relation denoting the result graph. The result graph may (and usually does) have $\varepsilon$ edges: they are removed in a final step with a transitive closure operation, see Sec. 4.1. Alternatively, $\varepsilon$ edges may also be removed in intermediate subexpressions. The query plan's operators are the selections, generalized projections, and dependent joins. The generalized projections allow us to apply a Skolem function to some columns in the input relation and construct new nodes in some column of the output relation. Dependent joins (see e.g. [FLMS99]) are non-commutative: they bind variables to values in the left operand, then evaluate the right operand for each such value. Fig. 15 illustrates a query plan for our example, omitting the $\epsilon$-edge removal operation. It is obtained by applying the bulk semantics of Sec. 5.1.1 for every recursive function, and by doing several simplifications.

In general, every query can be expressed first in the relational calculus using the expressions in Sec. 5.1.1, then translated into a query plan. This, however, results in an unnecessarily complex plan, that needs to be simplified in order to be practical. In Fig 15 we did the simplifications by hand: a general framework for such simplifications is beyond the scope of the paper. We explain now the plan in Fig. 15.

Recall the terminology we used in the bulk semantics Sec. 5.1.1: a structural recursion function $f$ returns a graph consisting of two types of nodes: $S1$ nodes and $S2$ nodes. In the query plan, there is a union operator ($\cup$) for each such function: the left operand computes the $S1$-to-$S2$ and $S2$-to-$S1$ edges (all are $\varepsilon$ edges), while the right operand computes the actual body (the $S2$-to-$S2$ edges). The topmost union operator in our example corresponds to `f7`: since its body is nonrecursive, there are no output markers, and, hence, no edges of type $S2$-to-$S1$ in the left operand. Moreover, `f7` will only be applied to the root node, hence we further optimized by selecting $u = root$. The function `f7` constructs nodes with Skolem functions $R1$ and $R2$: $R1(root)$ will be the root of `f7(db)`, while $R2(C)$ will be the connection node to the right operand. The right operand repeats the selection $\sigma_{u=root, l=\text{"}country\text{"}}$ (a potential for further optimization), but only retains the bindings of the $C$ variable, which it passes to the right operand of a dependent join. Here, the top-most union operator corresponds to `g7(C)`. The relation $E$ is now scanned with the first column bound to $C$, and searched for label name `"name"` and value `"France"`: for each such edge a new $\varepsilon$ edge is created from $R2(C)$ to $S1(C)$ (with $S1$ a new Skolem function). The right operand here corresponds to `h7(C)`: since no variables are bound by `g7`, there is no dependent join here. The plan for `h7(C)` traverses the entire graph, since `h7` is recursive: hence, no simplifications are further possible to exploit the fact that it is called on the node $C$. But the plan for `h7` contains a different simplification: $S2$ nodes have been eliminated. Instead, each label $L$ contributes either to a labeled $S1$-to-$S1$ edge (when $L$ is `"ethnicGroup"`), or to an $\varepsilon$ such edge. This optimization was mentioned in Sec. 5.1.1 as a possible way to reduce the number of $\varepsilon$-edges in the result.

Note that the original query was recursive, but the execution plan is non-recursive. This is because the plan constructs a graph with possible many $\varepsilon$-edges, corresponding to the recursion. In fact the plan for `h7` does just that: it replaces every label different from `ethnicGroup` with an $\varepsilon$-edge. These are eliminated in a final step (not shown in Fig. 15), as described at the end of Sec. 4.1.

The advantage of bulk evaluation is that we can perform separate optimizations. This is very powerful, because we can apply known optimization techniques for non-recursive queries to optimize structural recursion. For example optimizations for dependent joins have been considered in the context of object-oriented

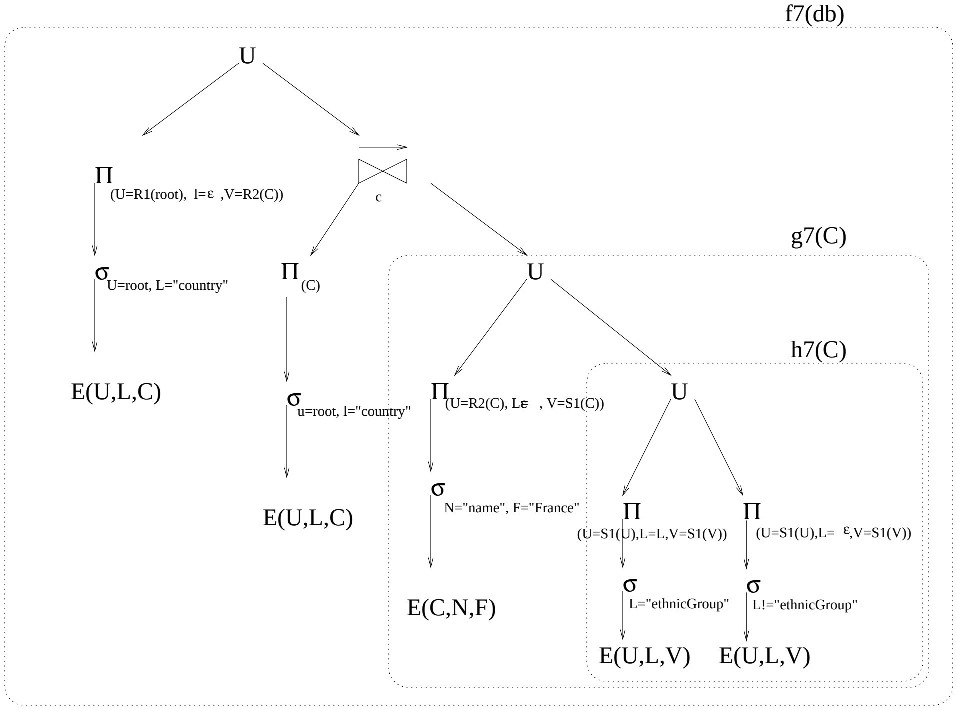


**Fig. 15** A query plan for bulk evaluation.

databases [BCD89, CD92] and of semistructured data [FLMS99]. The only restriction comes when the original query has the boolean predicate *isempty*: this introduces some transitive closure operators in the query plan, where optimizations are harder. However, most practical applications require only the positive fragment of UnQL (i.e. without *isempty*) and, hence, can benefit from well-known relational optimizations.

## 9 An Implementation of UnQL

We have implemented UnQL in Standard ML [AM87, ATT]. UnQL queries are translated into UnCAL, for which we implemented the top-down evaluation strategy. Input graphs are loaded into main memory and the query interpreter works directly on that graph representation. Inaccessible memory is automatically garbage collected in ML, therefore no explicit memory management is necessary. This is adequate for small input graphs, e.g., graphs with at most 1000 nodes and 10,000 edges, but it does not scale to large graphs with 10,000 or more nodes. For large scale inputs, a secondary storage manager for accessing graph elements directly from disk would be necessary. The entire implementation of UnQL and UnCAL is about 10,000 lines of commented ML code.

Our implementation performs an additional optimization to the naive recursive evaluation strategy. Namely, it keeps track of which markers are needed so that only those parts of a recursive function that are actually needed are evaluated. For example, assume we have two mutually recursive functions `f` and `g`, and a call `f(d)` for some data graph `d`. This will be translated into the UnCAL expression $\&z1 @ rec(e)(d)$, where $e$ is the recursion body, returning a graph with inputs and outputs $\&z1, \&z2$. The evaluation algorithm in Sec. 8 would compute $d' \stackrel{\text{def}}{=} rec(e)(d)$ first, then compute $\&z1 @ d'$. Recall that $d'$ is precisely $(\&z1 := f(d), \&z2 := g(d))$. In short, we have evaluated unnecessarily $g(d)$. To avoid that, our implementation uses a smarter evaluation strategy. For the top-most operator append (@), the recursive evaluation computes its left operand first, resulting in $d'' \stackrel{\text{def}}{=} \&z1$. Then the interpreter computes the set of all output markers in $d''$: in this case, it is $\{\&z1\}$, but in more complex cases, this may require a traversal[11] of $d'$. Next,

[11] In fact our implementation avoids this traversal altogether, by computing the set of output markers together with the result of the evaluation.

the interpreter evaluates the right operand of @, $rec(e)(d)$, but now it "knows" that only the input marker &$z1$ is needed. Hence it proceeds as in the Algorithm in Fig. 12, but at the root node computes only the &$z1$ component of $s1$. As it proceeds recursively, it only computes the required component of $s1$, at each node. Of course, cycles in the graph and the logic of a given query may force it to compute eventually the &$z2$ component for the root too, but that is computed only if it belongs to the accessible part of the result.

## 10 Conclusions and Acknowledgements

We believe that the principles of UnQL will be useful in describing the foundations of query languages for XML and semistructured data. We have illustrated query constructs (structural recursion), shown how they related to practical query languages (XSL), and proved powerful optimization techniques. We believe these foundations to be of use both in query language design and their implementation.

However, there are some important and interesting areas of research that may well bear fruit. In connection with XML, we have shown how the principles of UnQL will work on an ordered tree model, however it is not clear how they can be extended to an ordered graph model (the graph model discussed in section 4 was an unordered graph.) Some query primitives for dealing with order are provided by XML-QL but we still lack a complete picture of this topic.

Another area is the connection with Graphlog (see Sec. 1), i.e. datalog on graphs. This is another elegant formalism for querying semistructured data, but its connection with structural recursion and the connection with optimization techniques for recursive datalog queries [BR86] remain unexplored.

Optimizations of recursive queries as found for example in XSL is an exciting research area. The techniques illustrated here only provide the equational tools: future work is needed to combine these with a cost model and an optimization algorithm.

**Acknowledgements**. We are deeply indebted to Susan Davidson for her collaboration on this topic and for getting us interested in semistructured data in the first place. We would also like to thank Gerd Hillebrand for his contributions, and the anonymous reviewer who suggested to us the example in Sec. 5.1.1 and made numerous constructive criticisms.

## References

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and Xml.* Morgan Kaufmann, 1999.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

[AK89] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conference*, pages 159–73, Portland, OR, May 1989.

[AM87] Andrew W. Appel and David B. MacQueen. A standard ml compiler. *Functional Programming Languages and Computer Architecture*, 1987.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[ATT] AT&T Bell Laboratories, Murray Hill, NJ 07974. *Standard ML of New Jersey User's Guide*, February 1993.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 97–105, 1987.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the $O_2$ object-oriented database system. In *Proceedings of 2nd International Workshop on Database Programming Languages*, pages 122–138. Morgan Kaufmann, 1989.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[BDS95] Peter Buneman, Susan Davidson, and Dan Suciu. Programming constructs for unstructured data. In *Proceedings of the Workshop on Database Programming Languages*, Gubbio, Italy, September 1995.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Conference*, pages 16–52, Washington, DC, USA, May 1986.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Conf. on Database Programming Languages, DBPL*, 1991.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[CD92] Sophie Cluet and Claude Delobel. A general framework for the optimization of object oriented queries. In M. Stonebraker, editor, *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 383–392, San Diego, California, June 1992.

[Cla99a] James Clark. Xml path language (xpath), 1999. `http://www.w3.org/TR/xpath`.

[Cla99b] James Clark. Xsl transformations (xslt) specification, 1999. `http://www.w3.org/TR/WD-xslt`.

[CM90] M. P. Consens and A. O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, Nashville, TN, April 1990.

[Con98] World Wide Web Consortium. Extensible Markup Language (XML) 1.0, 1998. `http://www.w3.org/TR/REC-xml`.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 5, pages 193–242. Elsevier, Amsterdam, 1990.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, Toronto, 1999.

[DGM98] D.Calvanese, G.Giacomo, and M.Lenzerini. What can knowledge representation do for semi-structured data ? In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.

[FFK+98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[FFLS97] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[FLMS99] D. Florescu, L. Levy, I Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Philadelphia, June 1999.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness*. W. H. Freeman, San Francisco, 1979.

[GPVdBVG90] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. In *ACM Symposium on Principles of Database Systems*, pages 417–424, 1990.

[GPVdBVG94] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of Very Large Data Bases*, pages 436–445, September 1997.

[HHK95] Monika Henzinger, Thomas Henzinger, and Peter Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of 20th Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.

[KW93] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77–120, 1993.

[Mai86] D. Maier. A logic for objects. In *Proceedings of Workshop on Deductive Database and Logic Programming*, Washington, D.C., August 1986.

[Mil89] Robin Milner. *Communication and concurrency*. Prentice Hall, 1989.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277–295, 1999.

[MW99] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of Very Large Data Bases*, pages 413–424, September 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[PT87] Robert Paige and Robert Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16:973–988, 1987.

[QL98] Query for XML: position papers. http://www.w3.org/TandS/QL/QL98/pp.html.

[Rob99] Jonathan Robie. The design of xql, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[RS97] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. Springer Verlag, 1997.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.


## A Lifting Equations from Finite Trees to Graphs

In this appendix we establish formally Fact 1 enabling us to lift equations from finite trees to graphs. We are concerned here with infinite graphs, with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$. The sets $\mathcal{X}, \mathcal{Y}$ are finite, which implies that every graph has only a finite set of input nodes. Recall that a finite graph is referred to as a data graph. Throughout this appendix we are only concerned with graph isomorphism, not bisimulation: thus, the notation $g = g'$ means that $g$ and $g'$ are isomorphic.

An infinite graph is called *locally finite* if for each node the set of its outgoing edges is finite. We only consider locally finite graphs: this is sufficient for our purposes because we only need to consider infinite graphs of the form *unfold*$(d)$, for some data graph $d$.

A *forest* is a graph in which each input node is the root of a (finite or infinite) tree. In other words, a forest is a finite set of trees, each with a distinct input marker (because $\mathcal{X}$ is a finite set). A tree is a special case of a forest with a single input node.

*Compactness* Given two forests $t, t'$ with the same inputs and outputs we say that $t$ *occurs* in $t'$, in notation $t \subseteq t'$, if there exists an isomorphism between $t$ and a subgraph of $t'$: we require the isomorphism to preserve the marker labeling, for both input markers and output markers. Obviously $\subseteq$ is reflexive and transitive, and is also antisymmetric (this follows from the fact that the forests are locally finite). The following is straightforward:

**Fact 5** *Given two (possible infinite) forests $T, T'$ with the same inputs and outputs, we have $T = T'$ iff for any finite forest $t$, $t \subseteq T \iff t \subseteq T'$.*

Note that this property only holds when $T, T'$ are locally finite, otherwise it is easy to find counterexamples. For example take both $T, T'$ to be flat trees, i.e. consisting of a root an a single level of children, both have infinitely many children, and all edges have the same label, $a$. The only difference is that $T$ has countably many children, while $T'$ has uncountably many. Then all finite trees $t$ occurring in either $T$ or $T'$ have the form $\{a, a, \ldots, a\}$, but the two trees are not isomorphic.

The rest of this Appendix deals with functions on labeled graphs. More precisely, writing $G^{\mathcal{X}}_{\mathcal{Y}}$ for the set of labeled graphs with inputs $\mathcal{X}$ and outputs $\mathcal{Y}$, we consider functions of type $f : G^{\mathcal{X}}_{\mathcal{Y}} \to G^{\mathcal{X}'}_{\mathcal{Y}'}$ that map isomorphic inputs to isomorphic outputs, and map forests to forests. The sets $\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}'$ determine the function's *type*. Two functions have the same type if they have the same $\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}'$.

Often one needs to prove equalities of the form $f(d) = g(d)$, for all data graphs $d$, where $f, g$ are two functions on labeled graphs of the same type. It is much easier to prove instead $f(t) = g(t)$, for all finite forests $t$. In this appendix we state two sufficient conditions which allow us to lift equations from finite forests to graphs.

First we show how to lift equations from finite forests to infinite forests. The key property here is:

**Definition 5** *Let $f$ be a function from forests to forests. We say that $f$ is* compact *if, for any $t'$ the following property holds: $t' \subseteq f(T)$ iff $\exists t_0 \subseteq T$ s.t. $\forall t_1$, $t_0 \subseteq t_1 \subseteq T \Longrightarrow t' \subseteq f(t_1)$. Here $t', t_0, t_1$ range over finite forests.*

Note that a compact function is not necessarily monotone (i.e. $T \subseteq T'$ does not imply $f(T) \subseteq f(T')$). In UnCAL the *isempty* operator allows us to define non-monotone functions. For example consider the function `f(T)` below:

```
sfun h({a:T}) = {a}
   | h({L:T}) = h(T)

sfun f({a:T}) = {}
   | f({L:T}) = if isempty(h(T)) then {L} U f(T) else f(T)
```

which takes a tree `T` and returns the set of labels `L` in `T` which have no descendants labeled `a` (this example is the negation of that in Sec. 5.1.4). For some tree `T` a label `b` may be in the result `f(T)`, but if we extend `T` to `T'` (thus $\mathtt{T} \subseteq \mathtt{T}'$) by introducing an `a` label under `b`, then `b` is not in `f(T')`, hence `f` is not monotone. Still, we show that `f` is compact. Suppose $\mathtt{t}' \subseteq \mathtt{f}(\mathtt{T})$. We choose $\mathtt{t_0} \subseteq \mathtt{T}$ to consists of that subtree of `T` which includes all edges of `T` whose label `b` is in `t'`: for each such edge we also include in $\mathtt{t_0}$ all the ancestor edges too. Then, every extension $\mathtt{t_1}$ of $\mathtt{t_0}$ (i.e. $\mathtt{t_0} \subseteq \mathtt{t_1} \subseteq \mathtt{T}$) will include all those edges `b` too, and, moreover, none of these edges will have an `a` descendant in $\mathtt{t_1}$, because they didn't have one in `T`. Hence, $\mathtt{t}' \subseteq \mathtt{f}(\mathtt{t_1})$.

**Proposition 6** *Let $f, g$ be two compact functions. If $f(t) = g(t)$ for every finite forest $t$, then $f(T) = g(T)$ for every forest $T$.*

*Proof* Given some $T$, let $T_1' = f(T), T_2' = g(T)$. We plan to use Fact 5, so let $t' \subseteq T_1'$. If we can prove $t' \subseteq T_2'$ then we are done, since the reverse is proved similarly. To do this, we use the compactness property for $g$. First we apply the compactness property to $f$ and $t' \subseteq T_1'$. We obtain that there exists $t_0 \subseteq T$ s.t. whenever $t_0 \subseteq t_1 \subseteq T$, we have $t' \subseteq f(t_1)$. Since $f(t_1) = g(t_1)$, we also haver $t' \subseteq g(t_1)$. Hence we can use the compactness property for $g$ and conclude that $t' \subseteq T_2'$. □

Next we show how to lift equations from infinite forests to graphs.

**Definition 6** *Let $f$ be a function from labeled graphs to labeled graphs. We say that it is unfolding if $unfold(f(d)) = f(unfold(d))$, for any data graph $d$.*

**Proposition 7** *Let $f, g$ be unfolding. Then $f(T) = g(T)$ for every forest $T$ implies $f(d) = g(d)$ for any data graph $d$.*

*Proof* Follows directly from $f(d) = f(unfold(d)) = g(unfold(d)) = g(d)$. □

The following theorem then substantiates Fact 1:

**Theorem 6** *If $f, g$ are compact and unfolding, then $f(t) = g(t)$ for every finite forest $t$, implies $f(d) = g(d)$ for any data graph $d$.*

*Proof* We simply combine the two propositions above. □

We end this appendix showing how Theorem 6 can be applied to UnCAL queries. It turns out that not all UnCAL queries are compact. For a trivial example, consider a query `k` that returns $\{\mathtt{c}\}$ if there exists a subtree of the form $\{\mathtt{a} : \{\}\}$, and returns $\{\mathtt{d}\}$ otherwise. Consider an infinite chain $\mathtt{T} = \{\mathtt{a} : \{\mathtt{a} : \{\ldots\}\}\}$. Then $\mathtt{k}(\mathtt{T}) = \{\mathtt{d}\}$, but for any finite subtree $\mathtt{t_1} \subseteq \mathtt{T}$, $\mathtt{k}(\mathtt{t_1}) = \{\mathtt{c}\}$. However, we prove the compactness property for an important class of UnCAL queries.

Call an UnCAL query *positive* if every occurrence of the *isempty* operator has the form if $isempty(e_1)$ then $\{\}$ else $e_2$. Note that positive queries may still contain unrestricted conditionals testing label equality, i.e. of the form if $l_1 = l_2$ then $e_2$ else $e_3$. We will prove below that all positive UnCAL queries are compact. Positive UnCAL queries are monotone, of course: the function `f` earlier in this Appendix is an example of a non-monotone UnCAL query which is compact.

First, we need to discuss how UnCAL's semantics carries over from finite graphs to infinite graphs. For the constructors the same definitions in Fig 7 apply, while for structural recursion we adopt the bulk semantics. In particular each UnCAL query returns a locally finite graph: the constructors only introduce finitely many outgoing edges (however both $d_1 @ d_2$ and $cycle(d)$ may introduce infinitely many incoming edges to some nodes), while for structural recursion $rec(e)(t)$ one can check in the bulk definition that the outdegree of an $S1$ node is the same as the outdegree of the corresponding node in $t$, while the outdegree of an $S2$ node is the same as the outdegree of some node in $e(l, d_v)$ (refer to the notations in Sec. 5.1.1), hence $rec(e)(t)$ is locally finite. However, an UnCAL query does not map forests to forests. For example consider $t@t'$. If both $t, t'$ are trees, then in $t@t'$ there are several $\varepsilon$-edges leading to the root of $t'$ (one from each output node in $t$). Similarly $cycle(d)$ and $rec(e)(t)$ fail to return a forest in general. To get around that, given an UnCAL query $f(t)$, we apply Theorem 6 to the function $f'(t) = unfold(f(t))$. More precisely we prove the following.

**Proposition 8** *Let $f(t)$ be a positive UnCAL query. Then $f'(t) \stackrel{\text{def}}{=} unfold(f(t))$ is both compact and unfolding.*

*Proof* The proof is done by induction on an UnCAL query, referring to UnCAL's definition in Fig. 9. Recall that an UnCAL query is an expression with a unique free variable. Subexpressions however may have multiple free variables, hence we can assimilate such an expression with a function $f(t_1, \ldots, t_n)$. The definitions for both compactness and invariance under unfolding can be extended naturally to such functions. For illustration purposes however we show the proofs only for the case of a single free variable.

We consider a few cases for $f$: the others are proved similarly.

- The case when $f(t) = t$. Then $f'(t) = unfold(t)$ is trivially unfolding. Compactness is also trivial to prove: we show the details next. Notice that for any forest $T$, $f'(T) = T$, so let $t' \subseteq T$: we pick $t_0 \stackrel{\text{def}}{=} t'$, and notice that for every $t_1$ s.t. $t_0 \subseteq t_1 \subseteq T$ we also have trivially $t' \subseteq f'(t_1) = t_1$. This proves the "only if" direction. For the "if" direction, given some $t_0 \subseteq T$, choose $t_1 \stackrel{\text{def}}{=} t_0$. The conditions $t_0 \subseteq t_1 \subseteq T$ are satisfied, hence we have $t' \subseteq f(t_1) = t_1$, which also gives us $t' \subseteq T$.
- The case when $f(t) = f_1(t) \cup f_2(t)$. Technically $f(t) = \{\varepsilon : f_1(t), \varepsilon : f_2(t)\}$, and we have similarly $f'(t) = \{\varepsilon : f_1'(t), \varepsilon : f_2'(t)\}$. We prove invariance under unfolding first: $unfold(f'(d)) = unfold(\{\varepsilon : f_1'(d), \varepsilon : f_2'(d)\}) = \{\varepsilon : unfold(f_1'(d)), \varepsilon : unfold(f_2'(d))\}$. Here we apply induction hypothesis on $f_1', f_2'$, and the claim follows immediately. Now we prove compactness, showing the "only if" direction (the "if" direction is similar and omitted). Assume $t' \subseteq \{\varepsilon : f_1'(T), \varepsilon : f_2'(T)\}$. There are three cases: when $t'$ has a single node, when it contains only one of the two $\varepsilon$ edges, or when it contains both. We illustrate the last case only, the others are similar (actually simpler) and omitted. Hence $t' = \{\varepsilon : t_1', \varepsilon : t_2'\}$ and $t_1' \subseteq f_1'(T)$, $t_2' \subseteq f_2'(T)$. Now we apply the induction hypothesis which says that $f_1', f_2'$ are compact, and this gives us a $t_{01} \subseteq T$ for $f_1'$ and a $t_{02} \subseteq T$ for $f_2'$ s.t. $\forall t_1, t_{01} \subseteq t_1 \subseteq T$ implies $t' \subseteq f_1'(t_1)$, and similarly for $f_2'$. We choose then $t_0$ to be the "union" of $t_{01}$ and $t_{02}$. Recall that containment $t_{01} \subseteq T$, $t_{02} \subseteq T$ means that $t_{01}, t_{02}$ are isomorphic to finite subforests of $T$. The union refers to those two subforests in $T$: hence $t_0$ is a finite subforest of $T$, $t_0 \subseteq T$, and we have both $t_{01} \subseteq t_0$ and $t_{02} \subseteq t_0$. Now consider tome $t_1$ s.t. $t_0 \subseteq t_1 \subseteq T$. It follows that $t_{01} \subseteq t_1 \subseteq T$, hence, by compactness for $f_1'$, we have $t_1' \subseteq f_1'(t_1)$. Similarly $t_{02} \subseteq t_1 \subseteq T$, hence, by compactness for $f_2'$, we have $t_2' \subseteq f_2'(t_1)$. It follows that $t' = \{\varepsilon : t_1', \varepsilon : t_2'\} \subseteq \{\varepsilon : f_1'(t_1), \varepsilon : f_2'(t_1)\}$.
- The case when $f(t) = rec(e)(g(t))$ where $e$ is $\lambda(l', t').e(l', t')$. In its most general form $e$ may also depend on $t$, but for illustration purposes we assume that it only depends on $l'$ and $t'$. We sketch the proof for invariance under unfoldings first. We start by noticing that $unfold(f(d)) = unfold(rec(e')(g'(d)))$, where $e' = unfold(e)$, and $g'(d) = unfold(g(d))$: in other words, doing some unfoldings early doesn't hurt if we want to unfold the final result anyway. We omit the lengthly details of this simple fact, but illustrate the intuition behind it with the graph in Fig. 10(c). If we unfold the graph on the right of the figure, we get an infinite forest. The same can be obtained by unfolding the graph on the left first (the reader may help visualize it by looking at Fig. 6, which contains a similarly shaped graph and shows its unfolding), then applying structural recursion, then unfolding again (since structural recursion creates a directed acyclic graph). Invariance under unfoldings follows then immediately: $unfold(f'(d)) = f'(d) = unfold(rec(e')(g'(d))) = unfold(rec(e')(g'(unfold(d))))$ (we applied induction hypothesis on $g'$). Now we sketch the proof for compactness, illustrating the "only if" direction only. Here we start with $t' \subseteq f'(T) = unfold(rec(e')(g'(T)))$. Hence $t'$ will consists of a finite set of $S1$ nodes and a finite set of $S2$ nodes. The first set gives us a tree $t_g' \subseteq T' = g'(T)$, while the second set gives us, for each edge $(u, a, v)$ in $t_g'$, a tree $t_{e,u,a,v}' \subseteq e'(a, T_v')$ (notations as defined in Sec. 5.1.1): assume that there are $n$ such trees $t_{e,u,a,v}'$. We apply the compactness property for $g$ and $t_g'$, and this gives us some tree $t_{g0}$. Next we

apply the compactness property $n$ times for $e$, once for each of the $n$ trees $t'_{e,u,a,v}$: this gives us $n$ trees $t_{10}, \ldots, t_{n0}$. We then take $t_0$ to be the "union" (as discussed above) of the $n+1$ trees $t_{g0}, t_{10}, \ldots, t_{n0}$. Let now $t_1$ be such that $t_0 \subseteq t_1 \subseteq T$. From $t_{g0} \subseteq t_1$ and the fact that $g$ is compact, it follows that $t'_g \subseteq g(t_1)$, i.e. $g(t_1)$ contains all nodes for the $S1$ nodes we need to include in $t'$. Furthermore $t'_{10} \subseteq t_0, \ldots, t'_{n0} \subseteq t_0$, hence, by the compactness argument for $e'$, the combined graphs $e'(a, T'_v)$ contain all the $S2$ nodes in $t'$. It follows that $rec(e')(g'(t_1))$ contains both the $S1$ nodes and the $S2$ nodes in $t'$, in other words $t' \subseteq rec(e')(g'(t_1)) = f'(t_1)$.

– We finally illustrate the case when $f(t) =$if $isempty(f_1(t))$ then $\{\}$ else $f_2(t)$. For invariance under unfolding we notice that $isempty(f_1(t)) = isempty(unfold(f_1(t))) = isempty(f'_1(t))$, hence $f'(t) =$if $isempty(f'_1(t))$ then $\{\}$ else $f'_2(t)$. We have $unfold(f'(t)) = f'(t) =$if $isempty(f'_1(t))$ then $\{\}$ else $f'_2(t)$ and we apply the fact that $f'_1, f'_2$ are unfolding. For compactness we show the "only if" direction. Let $t' \subseteq f'(T)$. There are two cases: when $isempty(f_1(T)) = true$ and when it is $false$. In the first case we have $f'(T) = \{\}$, hence $t'$ can only be $\{\}$, so $t' \subseteq f'(t_1)$ for any $t_1$ (one can also notice that $f'(t_1)$ is, in fact, $\{\}$, whenever $t_1 \subseteq T$, but this fact is not needed). In the second case we have some non-$\varepsilon$ edge in $f_1(T)$: let $s' \subseteq f_1(T)$ be some finite tree containing at least one such edge. By compactness for $f_1$, we find some $s_0$ s.t. for all $t_1$, $s_0 \subseteq t_1 \subseteq T$ we have $s' \subseteq f_1(t_1)$, hence $isempty(f_1(t_1)) = false$. To prove compactness for $f'(T)$, let $t' \subseteq f'(T) = f'_2(T)$. We now use compactness for $f'_2$ and find some $t_{20}$ s.t. for all $t_1$ containing $t_{20}$, $t' \subseteq f'_2(t_1)$. We choose then $t_0$ to be the "union" (as above) of $s_0$ and $t_{20}$. Let $t_1$ be such that $t_0 \subseteq t_1 \subseteq T$. We still have $isempty(f_1(t_1)) = false$, hence it is still the case that $f'(t_1) = f'_2(t_1)$, and we also have $t' \subseteq f'_2(t_1)$.

The other cases are similar and omitted. □