Towards a Language for the Fully Generic Queries

Catriel Beeri[1], Tova Milo[2] and Paula Ta-Shma[1]

[1] Institute of Computer Science, Hebrew University
Jerusalem, 91904, Israel
{beeri,paula}@cs.huji.ac.il
[2] Department of Computer Science, Tel-Aviv University
Tel-Aviv, Israel
milo@math.tau.ac.il

1 Introduction

Database queries differ in one fundamental aspect from regular functions — they are generic, that is invariant under database isomorphisms. This property was first discussed in [AU79]. At about the same time, [CH80] posed the problem whether there is a complete query language, that is a language that expresses precisely the computable generic database functions, and presented an affirmative answer.

Over the years, the notion of genericity has been refined. Clearly, if a constant is used in a query, the query cannot be expected to be invariant under renamings that do not preserve it. Consequently, Hull and Yap [HY84] introduced the idea of invariance under isomorphisms that preserve some constants. Their notion of C-generic, where C is a finite set of constants, is essential to capture functions such as $\sigma_{A=3}(R)$ as queries. But now, if we consider the query $\sigma_{prime(A)}(R)$, we see that the same argument holds: If a renaming does not preserve the primality of numbers, then the query will not be invariant under it. Thus, one must in the general case consider mappings that preserve some set of domain constants, functions and predicates (often referred to as built-in functions/predicates.) Typically, if a query does not use a built-in function/predicate, it should be invariant under mappings that do not preserve it.

Generalizing further, genericity is invariance under some class of mappings [BMT96]. The larger this class of mappings, the smaller resulting class of queries. The class can be defined by a set of constants and functions that need to be preserved, or by some other means. For example, in recent work on spatial query languages [PVV94, PSV96], genericity is considered w.r.t. translations, similarities and homeomorphisms of the plane, and each defines a different class of queries. In [BMT96], we defined the class of *fully generic* queries as those queries that are invariant under the largest class of mappings. Being the smallest possible class of queries, this class is contained in all other genericity classes, and is in this sense fundamental. This class can be characterized as those queries that are invariant also under mappings that do not preserve equality. Indeed, equality is also a built-in predicate of a domain. When a query uses equality, it is invariant only under mappings that preserve it, i.e. one-to-one mappings; but if it does not use it, it should also be invariant under mappings that are not one-to-one. Thus, the fully generic queries, embodying the "pure" notion of genericity, are invariant under arbitrary mappings on the domain, many-to-many in the general case. Following the view that genericity is a fundamental property of database query languages, one can say that the class of fully-generic queries is the class of quintessential database functions.

In this paper we investigate the existence of a complete language for the fully generic queries:

Open Problem 1.1 *Does there exists a query language $\mathcal{L}$ such that for all queries Q, Q is fully generic iff Q is expressible in $\mathcal{L}$?*

We settle this question in the affirmative for relational databases. We also study queries over complex values and settle the question in the affirmative at certain types, and in particular for all flat or boolean output queries over nested input types, and for queries whose input and output are doubly nested sets. The novel proof technique we use is interesting in itself, and is constructive, in that we construct all possible fully generic queries at these types.

Our investigation continues a long line of research of genericity in database theory. In particular, a solution of our problem may shed some light on the nature of complete languages for other genericity classes, although certainly the nature of the specific class of mappings used to define a genericity class plays an important role as well. Our investigation is also potentially relevant to practical issues. For example, in heterogeneous databases, and data on the Web, equality is often not available. For example, in the databases and logic programming bibliography server [Ley], there are currently separate entries for authors "Ross Paterson" and "Ross A. Paterson". Are these really two distinct authors ? While heuristics can be used to guess the answer[3], there is no computationally feasible way to provide a certain answer. It follows that, for example, if we have lists of accepted papers for DBPL and for VLDB, "Ross Paterson" is in the first list, and "Ross A. Paterson" in the second, then we cannot answer the query which asks for authors that have papers in both conferences; intersection uses equality hence is not fully generic. On the other hand, the $\cup$, $\times$ and π of the relational algebra are fully generic. Our study could therefore delineate theoretical bounds on queries on the Web.

The fully generic queries are also interesting because they are, in a sense, the most optimizable. The invariance of a query under a mapping can be restated as an algebraic equivalence. Invariance under more mappings directly implies more applicable equivalences. For example, projection is a (non-injective) mapping, therefore, fully generic queries, which commute with all mappings, commute with projection. Such equivalences are typically used for query optimization, and in this sense the fully generic queries are the most amenable to algebraic optimization.

[3] this practically important problem is called *data scrubbing*

We feel that genericity can be profitably used as a tool for language design, and for the study of language properties in general. The fully generic queries can be viewed as the pure restructuring queries, which do not use any domain specific information whatsoever. Our candidate for a complete fully generic language presented in Section 2 is closely related to the elegant core nested relational algebra presented in [BBW92]. As shown in [BBW92], adding equality to this core language gives precisely the nested relational algebra. We feel that, in [BBW92], a key behind the elegant language design is the separation of the fully generic restructuring queries from the domain specific operations (such as equality, orderings etc.) which are added as needed. Although language design is a matter of taste, we point out that the language of [BBW92] was successfully used as a vehicle for proving numerous properties of the nested relational algebra, such as conservative extension [Won93], which were significant improvements on previous results. This is despite the fact that other presentations of the nested relational algebra had been studied for some time.

Finally, we point out that our conjecture and the results we have obtained are of independent interest. For example, if our conjecture about the complete language is true [4], then all fully generic queries are computable. This contrasts with all other genericity classes studied to date, which contain non-computable queries. More importantly, it may be possible, using our methods, to attain a *complete understanding* of the fully generic queries. For example, equivalence of fully generic queries may be decidable, which would again be in contrast with most generic database languages such as the relational algebra.

Section 2 presents the definitions of full genericity and queries, and the candidate language. In Section 3, we show that complex values of a given type can be partitioned into finitely many disjoint classes, such that the behavior of a fully generic query on inputs of this type is determined independently for each class. This allows us to prove the main result for each class separately. Full genericity implies that inputs and outputs of queries are related by patterns. We define these in Section 4, and present closure properties for the set of patterns of a query. In Section 5 we prove our conjecture for the (simple) case of queries with relational output. Section 6 deals with the much more difficult case of input and output with double set nesting. Here we make full use of the closure properties.

2 The Problem and Our Approach

2.1 Complex values

We adopt the complex value model here. A complex value is a generalized nested relation, where set and tuple constructors can be arbitrarily interleaved [AHV95]. Atomic values come from an infinite domain, denoted D.

Definition 2.1 (complex value type) *Complex value types are defined recursively as* $t := d \mid [] \mid t \times \ldots \times t \mid \{t\}$, *where* d *denotes the domain* D, *and* $[]$ *is the type of 0-ary tuples (there is only one such tuple, which is also denoted* $[]$*).*

[4] As indicated above, we have proofs for some but not for all complex value types

Each complex value type τ is associated with a domain of complex values, $dom(\tau)$, as follows.

$$dom(d) = D \qquad dom([]) = \{[]\} \qquad dom(\{t\}) = \mathcal{P}^{fin}(dom(t))$$
$$dom(t_1 \times \ldots t_n) = dom(t_1) \times \ldots \times dom(t_n)$$

$\mathcal{P}^{fin}$ denotes finite powerset. Following [CH80, BNTW95], we denote the type $\{[]\}$ as `bool`, and use $\{\}$ for *false* and $\{[]\}$ for *true*. Complex values can also be viewed as trees, where internal nodes are labeled with the type constructors, and the leaves are labeled with atomic values, $[]$ and $\{\}$.

2.2 Mappings and genericity

We now define mappings on complex values, and then define genericity as invariance under such mappings. A mapping on complex values is obtained by taking some mapping on atoms and extending it. For example, classical genericity considers one-to-one mappings on atoms, which are uniquely extended to one-to-one mappings on complex values. A mapping which maps 1 to a and 2 to b, when extended to complex values, maps $\{[1,2],[2,2]\}$ to $\{[a,b],[b,b]\}$. Our mappings, however, are many-to-many in the general case. [5] For such mappings there is a unique natural extension to tuples, component-wise. However there is more than one candidate extension to sets, hence to complex values, each giving rise to a different notion of genericity [BMT96]. We use the *rel* extension mode of [BMT96], since it is the one that does not preserve equality in any way. [6]

Mappings on complex value types are defined inductively, starting from a mapping on atoms. We associate the two type constructors $\times, \{\}$ with mapping constructors, also denoted $\times, \{\}$:

Definition 2.2 (mappings) *A mapping H on type τ is some subset of $dom(\tau) \times dom(\tau)$. As a special case, a mapping on atoms is some subset of $D \times D$. We write $H(x,y)$ if the pair (x,y) belongs to this subset. Let $H, H_1, \ldots, H_n$ be given mappings on types $\tau, \tau_1, \ldots, \tau_n$.*

tuples: *$H_1 \times \ldots \times H_n$ is the mapping on $\tau_1 \times \ldots \times \tau_n$ s.t.:*
$$(H_1 \times \ldots \times H_n)([x_1, \ldots, x_n],[y_1, \ldots, y_n]) \iff H_i(x_i, y_i), (1 \leq i \leq n).$$
sets: *$\{H\}$ is the mapping on $\{\tau\}$ s.t.:*
$$\{H\}(S_1, S_2) \iff \begin{array}{l} \forall x_1 \in S_1\ \exists x_2 \in S_2 \text{ such that } H(x_1, x_2), \\ \text{and } \forall x_2 \in S_2\ \exists x_1 \in S_1 \text{ such that } H(x_1, x_2). \end{array}$$

Now, given a mapping H on atoms, it can be extended inductively to each complex value type. We denote by H_τ the mapping obtained by lifting H to type τ. In most of the paper we abuse the notation and simply write $H(t,t')$, for two values of type τ.

[5] Many-to-many mappings are commonly referred to as *relations*, although this term is overloaded in our context.

[6] This extension mode has also been used to lift partial orders (as opposed to general relations) to sets, in work on querying incomplete information. There it is known as the Plotkin ordering [LW93].

Example 2.3 Firstly, we extend the mapping $H_d = \{(1,a),(1,b),(2,b)\}$, which is not functional in either direction, to sets, to obtain $H_{\{d\}} =$
$\{(\{1\},\{a\}),(\{1\},\{b\}),(\{1\},\{a,b\}),(\{2\},\{b\}),(\{1,2\},\{a,b\}),(\{1,2\},\{b\})\}$.
As a second example, consider the mapping $H_d = \{(e,a)(i,a)(f,b)(j,b)(g,c)\}$ and the two binary relations $r_1 = \{[e,f],[i,f],[e,j],[i,j],[f,g],[j,g]\}$, $r_2 = \{[a,b],[b,c]\}$. H_d can be extended (as explained above) to a mapping over 2-ary tuples, and then further extended to a mapping over sets of 2-ary tuples. It is easy to verify that in this extended mapping, $H_{\{d\times d\}}(r_1, r_2)$ holds. $\square$

We note that for any type τ, $H_\tau(\{\}, S)$ holds for S of type τ if and only if $S = \{\}$. (We overload notation and use $\{\}$ for empty sets of all types). Moreover, extensions of many-to-many mappings do not preserve cardinality of sets (except for cardinality 0, as just noted).

Since databases can be viewed as tuples of complex values, queries are functions from complex values to complex values. Such a function is fully generic if it is invariant under *all* mappings:

Definition 2.4 *A total function $Q : \tau_1 \to \tau_2$ is* **fully generic** *if for all mappings H on atomic domains, and for all $S, S' \in dom(\tau_1)$, whenever $H(S, S')$ holds, so does $H(Q(S), Q(S'))$.* [7] *A* **query** *$Q : \tau_1 \to \tau_2$ is a total fully-generic function from $dom(\tau_1)$ to $dom(\tau_2)$.*

Note that we require queries to be typed, total and fully generic, but not necessarily to be computable. In [BMT96] we also considered restricting the mappings H above to be *functional*, and showed that the resulting notion of full genericity coincides with this one.

2.3 A fully generic language

We search for a query language that expresses exactly all fully generic queries. Our candidate language $\mathcal{L}$ is the fully generic component of the complex value algebra, and a new operator we call *one-each*, see Figure 1.. $\mathcal{L}$ is (essentially) the same language as $\mathcal{M}_\cup(cond)$ from [BNTW95] plus *one-each*. *one-each* can be used in $\mathcal{L}$ to express *powerset* (hence the latter is not included in $\mathcal{L}$), but we do not know how to express *one-each* using the remaining operators of $\mathcal{L}$ and *powerset*. This is why we include it. We note that as soon as equality is added to $\mathcal{L}$ we get the full nested relational algebra plus *powerset* [BNTW95]. [8]

It is not hard to show (proof omitted) that

Proposition 2.5 *All queries expressible in $\mathcal{L}$ are fully generic.*

2.4 The problem and our approach

The rest of this paper deals with the question whether all fully generic queries can be expressed in $\mathcal{L}$. As we mentioned in the introduction, an affirmative

[7] This corresponds to rel-fully generic from [BMT96]

[8] Note that *powerset* plus equality can express *one-each*.

Query	Meaning
id	the identity function : $id(S) = S$
$\circ$	function composition
$[f_1, \ldots, f_n]$	takes n functions and creates a new function. On input v, $[f_1, \ldots, f_n](v) = [f_1(v), \ldots, f_n(v)]$.
$\$i$	selects the i^{th} column of an n-tuple
$map(f)$	applies f to each element of a set
$single$	creates a singleton set
flatten	flattens a set of sets into a set
$\times$	cartesian product
$\cup$	set union
$powerset$	(finitary) powerset
$one\text{-}each$	accepts a set of sets $S = \{s_1, \ldots, s_n\}$, and returns the set of all sets (whose elements are taken from S) having at least one element from each one of the s_i. For example $one\text{-}each(\{\{1\}, \{2, 3\}, \{3, 5\}\}) = \{\{1, 2, 3\}, \{1, 2, 5\}, \{1, 2, 3, 5\}, \{1, 3\}, \{1, 3, 5\}\}$.
$\{\}_\sigma$	always returns the empty set (of type $\{\sigma\}$)
$[\,]_\sigma$	maps all elements of type σ to the empty tuple, the single element of type $unit$
$cond$	a conditional (if then else) : it returns its second argument if its first is non-empty, and otherwise it returns its third argument.

Fig. 1. Candidate language $\mathcal{L}$

answer would imply, for example, that all fully generic queries are computable. In contrast, queries which are generic in the classical sense are not necessarily so. For example, any query from flat sets to `bool` that depends only on the input size must be generic w.r.t. all injective mappings, and it is easy to show that there are non-computable such queries. Thus, non-computable generic queries exist. The reason this might be different for fully generic queries is that they must act uniformly across inputs of different sizes, because a many-to-many mapping can relate *any* non-empty flat set to any other.

It is easy to verify that there are four fully generic queries from flat sets to `bool`, because such a query must choose either *true* or *false* for all non-empty inputs, and similarly for empty inputs. For non boolean output types, there are many more fully generic queries, since there are many ways to permute and combine the elements in the input to produce the output. Allowing flat inputs and outputs, we get, for example, $\cup$, π and $\times$, three out of five of the relational algebra operators. Moving to nested inputs and outputs we have *single*, *powerset*, *flatten* etc. Indeed the class of fully generic queries is by no means trivial—adding equality to $\mathcal{L}$ gives the entire nested relational algebra, *powerset* included [BNTW95]. Our experience is that as the inputs and outputs become more deeply nested, solving the problem ew posed becomes considerably more difficult. Intuitively, this is because deeply nested input can contain many patterns which can be exploited by fully generic queries—if the output is also deeply nested, the query can restructure these patterns in many ways. As the nesting gets deeper, there are just so many more possibilities.

Here is a brief outline of our approach to the solution of the problem. We observe that the inputs and outputs to fully generic queries are related by patterns; full genericity implies there are many such patterns. Descriptors are introduced to formally describe a query's patterns. A query is completely characterized by the set of its descriptors, which typically is infinite. For our purpose, we need a finite representation of this set, which is obtained as follows. We point out that a descriptor d_1 may dominate another descriptor d_2, meaning that whenever d_1 describes a fully generic query, then so does d_2. Therefore, for each fully generic query we search for a finite number of dominating descriptors (this is the most difficult part). Once that is achieved, we find a query in $\mathcal{L}$ corresponding to each dominating descriptor, and take the (finite) union, thus obtaining a query in $\mathcal{L}$.

In this paper we motivate and describe the central constructs and ideas behind the proof. A complete account can be found in [Ta-97].

3 Canonical Forms

Since full genericity deals with mappings between values, we need to understand when it is possible to map one complex value to another. For that we introduce the notion of *canonical form*.

Definition 3.1 *A* **canonical form** *(for type τ) is a complex value (of this type) containing at most one atomic value z.*

For example $\{\}$, $\{\{\}\}$, $\{\{[z,z]\}\}$, and $\{\{\},\{[z,z]\}\}$ are canonical forms.

Definition 3.2 *We say that a complex value S has canonical form c, denoted $cf(S)$, if c is obtained from S by replacing all atomic values in S (if any) by z.*

For instance, both values $\{\{\},\{[a,b]\}\}$ and $\{\{\},\{[a,b]\},\{[c,d],[e,f]\}\}$ have the canonical form $\{\{\}\{[z,z]\}\}$. Think of this canonical form as representing all depth two sets that contain the empty set, and some nonempty sets of pairs.

Note that the number of canonical forms for each type τ is finite, and they can all be constructed inductively from the type definition. (Proof omitted). For example, the type $\{\{b \times b\}\}$ has exactly the 4 canonical forms listed above $\{\}$, $\{\{\}\}$, $\{\{[z,z]\}\}$, and $\{\{\},\{[z,z]\}\}$.

An important observation, that is used extensively in the proofs in this paper, is that the language $\mathcal{L}$ can distinguish between inputs of different canonical forms.

Claim 3.3 *For each type τ and each canonical form c of type τ, the function $c(x) : dom(\tau) \rightarrow$ `bool` that returns true iff $cf(x) = c$ is expressible in $\mathcal{L}$.*

The significance of canonical forms for the study of genericity is in the fact they determine whether values can be mapped to each other :

Claim 3.4 *For any two values S_1, S_2 of the same type, there exists some H such that $H(S_1, S_2)$ if and only if $cf(S_1) = cf(S_2)$.*

Basically this means the following: Full genericity means uniformity of behavior, but this uniformity applies only to values of the same canonical form. There is no relationship between the outputs of a query for values with distinct canonical forms. This, with the fact that the language $\mathcal{L}$ can distinguish between canonical forms, implies that to prove our conjecture it suffices to consider a query's behavior separately on values of each canonical form. Another implication concerns partial queries. If for such queries full genericity means that a query is defined on a value if and only if it is defined on each value related to it by a mapping, then a query is fully defined or fully undefined on classes of values defined by canonical forms. Since canonical forms can be computed, each computable partial fully generic query can be 'completed' to a total computable fully generic query. This justifies our choice to study only total queries.

An important observation, that is used in the sequel, is that just 'having the same canonical form' may not suffice to understand the interaction between queries and mappings. Consider for example the nested set $\{\{\{\},\{a\}\},\{\{b\}\}\}$. The elements a and b in it have the same canonical form, and so do the sets $\{a\}$ and $\{b\}$. Still they can be distinguished from each other – the first appears in a nested set containing the empty set, while the other does not, that is, their parents in the input have different canonical forms. Indeed, there is a fully generic query that returns a value where a occurs but b does not.[9] To fully characterize a sub-value of the input, one must consider both its canonical form, and those of its ancestors in the input tree.

4 Query Descriptors

We expect fully generic queries to behave *uniformly* across inputs of the same canonical form. Informally, this means that for a query there are some fixed patterns that relate inputs (having a certain canonical form) to the outputs. *Query descriptors* are intended to capture such patterns.

We first show that a fully generic query is completely characterized by its behavior on what we will call completely disjoint values. We then define the notion of descriptor, which describes the essential patterns which hold between a query output and a completely disjoint query input.

4.1 Completely disjoint values

To study fully generic queries we need to understand their behavior on various inputs. We now justify studying this behavior only w.r.t to a subset of all possible inputs—the *completely disjoint* values. These are the values where no atomic value occurs more than once as a leaf in the value tree.

Claim 4.1 *If fully rel-generic queries Q_1, Q_2 agree on all completely disjoint inputs, then they agree on all inputs.*

Hence, in most of this paper we deal only with such input values.

[9] For example, the query $Q(s) = \{s'' \mid s'' \in s', s' \in s, \emptyset \notin s\}$.

4.2 Descriptors

Let S and T be values. We want to describe the pattern that relates S as an input to a query, and T as a potential sub-value [10] of the output. (We do not assume at that there actually is such a query.) We include the following components: First, T itself (in its tree form). Second, for every atom that appears in T (as a leaf), we record its origin in S. This is well-defined, assuming S is completely disjoint. This component can be visualized as pointers going from T's leaves to S's leaves. Since we want to record only the essential patterns between T and S, and since these should be independent of the actual atomic values involved, we now omit the actual values of the atoms. Moreover, if we can leave out more information (while still retaining the essential patterns), these patterns would become more widely applicable. We note that only a part of S is directly relevant to T: the atomic leaves that occur in T, and their positions in S's structure. We can expect T to be similarly related to other values that have these components of S, but differ from S in other parts. Therefore, we omit all parts of S, except for the leaves of S that have incoming pointers from T, and their ancestors up to the root.

This pruning of S's structure is, however, a bit too drastic. There are two pieces of information that are needed for the patterns to faithfully describe the input-output relations in queries, that cannot be derived without considering all of S. Therefore, these are included as node labels. First, as we noted before, we need to know the canonical forms of all nodes and their ancestors. Therefore we include the canonical form of each node. Second, consider a set node n, and assume it has in S a set of children of canonical form c, and some of them contribute to T. It turns out to be important to know whether all these children contribute to T, hence all are left in the descriptor, or some do not contribute to T, hence are removed. In the first case, we add to n's label the pair $(c, =)$, meaning that the set of such children in S is the same as those represented. In the second case , we add to n's label the pair $(c, <)$, meaning that the set represented is a subset of the set in S. Since n may have children of several canonical forms, its label will contain, in addition to its own canonical form, several such pairs.

Definition 4.2 *Let S be a completely disjoint complex value, and T any value, such that $adom(T) \subseteq adom(S)$. The* **descriptor** *of T w.r.t. S is the triple $d = \langle in(d), out(d), map(d) \rangle$, defined as follows:*

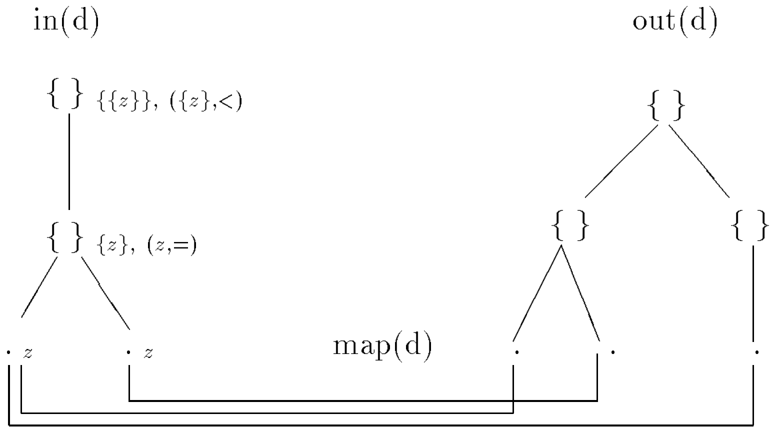
- *$in(d)$ is the minimal subtree of S that contains its root, and each leaf that occurs in T, with the atomic values that label the leaves removed, and with each node labeled with its canonical form.*
- *Additionally, internal set constructor nodes are labeled as follows: if a node has in S children of canonical form c, and all of them are in $in(d)$, then its*

[10] A sub-value is the value corresponding to a full subtree of a given complex value. For example $\{1, 2, 3\}$ is a sub-value of $\{\{1, 2, 3\}, \{4\}\}$ but $\{1, 2\}$ and $\{\{1, 2, 3\}\}$ are not.

label contains $(c, =)$*; if some of them but not all are in* $in(d)$ *then the label contains* $(c, <)$*.*

- *The children of tuple constructor nodes are labeled with their component number.*
- $out(d)$ *is* T*, with the atomic values at the leaves omitted.*
- $map(d)$ *is a mapping from the leaves of* $out(d)$ *to the leaves of* $in(d)$*, that maps each node to its unique source in* $in(d)$*.*[11]

Example 4.3 *Let* $S = \{\{b, c\}, \{d\}\}$ *and* $T = \{\{b, c\}, \{b\}\}$*. The descriptor of* T *w.r.t* S *is described graphically below. The leftmost leaf in* $in(d)$ *corresponds to the value* b*, the rightmost to* c*.* S *contains two sets of the same canonical form, namely two nonempty sets. Since only one of them contributes values to* T*, only one is represented. Therefore the root of* $in(d)$ *is labeled, in addition to* S*'s canonical form, with* $(\{z\}, <)$*, where* $\{z\}$ *is* $cf(\{b, c\}) = cf(\{d\})$*. On the next level down, since both members of* $\{b, c\}$ *occur in* T*, the label contains* $(\{z\}, =)$*.*



Note that if T is one of $\{\}, [\,], \{[\,]\}$, or any tree without atomic leaves, then its descriptor w.r.t a set S is $(\langle cf(S)\rangle, T, \emptyset)$. Also note that the descriptor is independent of specific atomic values appearing in the input and output. Only the relationship between them is being recorded.

The following claim shows that a descriptor successfully captures the patterns between input and output.

Claim 4.4 *Let* Q *be a fully generic query, and assume that* d *describes* T_1 *w.r.t.* S_1 *and* T_2 *w.r.t.* S_2*. Then* T_1 *is a sub-value of* $Q(S_1)$ *if and only if* T_2 *is a sub-value of* $Q(S_2)$*.*

This claim would not hold if we did not record the $<, =$ labels of a descriptor. For example, let $Q = id$, $S_1 = T = \{1, 2\}$, $S_2 = \{1, 2, 3\}$. Then a descriptor d which

[11] $map(d)$ is defined with respect to the trees before the atomic values are removed.

describes T w.r.t. S_1 has one set node with label $(\{z\}, =)$, and 2 children. The descriptor d' which describes it w.r.t. S_2 has exactly the same structure, except that the $=$ is replaced by $<$. If this information was not present, then since T is a sub-value of $id(S_1) = S_1$ the claim would imply that T is a sub-value of $id(S_2) = S_2$, which is not true (note that subsets are not necessarily sub-values).

For nested outputs, we can consider sub-values T at various levels. In the rest of this paper we are mainly interested in queries whose output is a set, and the sub-values that are considered are the members of these output sets. This together with the above claim justifies the following definition.

Definition 4.5 *d* **describes a query** *Q (of output type $\{\tau\}$) if there exist S, T such that $T \in Q(S)$ and d describes T w.r.t. S. The* **set of descriptors** *of Q, denoted D_Q is $\{d \mid d \text{ describes } Q\}$.*

Descriptors describe *members* of query output w.r.t. their input. Since Q above has output type $\{\tau\}$, T has type τ. This is also the type of $out(d)$.

A fully generic query is completely determined by its set of descriptors:

Claim 4.6 *If Q, Q' are fully generic and $D_Q = D_{Q'}$ then $Q = Q'$.*

4.3 Dominance

Our technique for showing that all fully generic queries (of certain types) are expressible in $\mathcal{L}$ works as follows. If D_Q is finite, then we consider each descriptor separately, and find a query in the language for it, then take their union. However, for some queries, (in particular queries with non-relational output type), the set of descriptors is infinite. For this case, think of the space of descriptors as a geometric space, and of D_Q as a subspace. We show that this subspace has a nice geometry, that allows us to infer a division into a *finite* number of subspaces. Then we present a query for each subspace, and take their union.

The observation that we use in this analysis is that some query descriptors are related in the sense that whenever a query has some descriptor, it also has some other descriptors.

Definition 4.7 *We say that d* **dominates** *d', denoted $d \rightarrow d'$, if for all fully generic queries Q, if d describes Q then also d' describes Q. For a descriptor d, $closure(d) = \{d' \mid d \rightarrow d'\}$.*

Note that the dominance relation is reflexive and transitive, and $d \rightarrow d' \iff closure(d') \subseteq closure(d)$.

The definition of the dominance relation is semantic. Some results concerning it used in our proofs, that use the form of descriptors, are presented below. Additional results that are also used in the proofs are omitted, for lack of space. For details, see [Ta-97].

Claim 4.8 ($<$-to-$=$ Property) *Assume d' is obtained from d by replacing some $(c, <)$ by $(c, =)$ in the label of a set constructor node. Then $d \rightarrow d'$.*

For example, the descriptor described in Example 4.3 dominates the descriptor that has the same components, except that the label of $in(d)$'s root now contains $=$ instead $<$.

For descriptors whose *out* component has no set constructors, i.e., those that describe atomic values or tuples of such values, the converse holds as well.

Claim 4.9 (=-to-< property) *Assume $out(d)$ has no set constructors, and d' is obtained from d by replacing some $(c, =)$ by $(c, <)$ in the label of a set constructor node. Then $d \rightarrow d'$.*

Combining the two results, we have that in descriptors whose *out* component has no set constructors, the components (c, p) in labels can be omitted.

We use other dominance properties which involve a descriptor's structure and not just its labels. Most importantly, the Merge Lemma shows how a descriptor dominates the result of merging two siblings in its *in* component. We present special cases of the Merge Lemma in section 6.

5 Queries from nested types to flat output

We are now ready to tackle the question whether all fully generic queries can be expressed in the language $\mathcal{L}$. We first consider queries from arbitrary nested input to *flat output*, that is, to sets of tuples, or sets of atoms. The proof technique we use here is specific and does not work for nested outputs, hence the separation.

Theorem 5.1 *All fully generic queries from arbitrary nested types to flat output are expressible in $\mathcal{L}$ (without one-each).*

In particular, all fully generic queries with relational input/output are expressible, and so are all fully generic boolean queries from arbitrary nested types (since the output is a 0-ary relation).

Proof. (sketch) We present below the major ideas behind the proof and the queries that result, for the case that Q is a query with relational output. (The case for output being a set of atoms is similar).

We first argue that for queries Q with fixed input type and relational output, the set D_Q of descriptors, is **finite**, because

1. Each descriptor d describes a tuple in the output of Q. The number of leaves of such a tuple is the relation arity. Thus the number of leaves in $in(d)$ is bounded by the relation arity.
2. The number of canonical forms for any given type is finite.

Now, assume $D_Q = \{d_1, ..., d_k\}$, for some finite k. We construct a query q_i (of the same type as Q) for each d_i, s.t. $D_Q = D_{q_1} \cup \ldots \cup D_{q_k}$, and therefore, $Q = q_1 \cup \ldots \cup q_k$. Each q_i is expressed in $\mathcal{L}$, (without *one-each*), and the Theorem follows. A brief description of the construction of the q_i's appears in Appendix A. The dominance results of Section 4.3 are used to show that indeed $D_Q = D_{q_1} \cup \ldots \cup D_{q_k}$.

Corollary 5.2 *All fully generic queries with relational output are computable.*

6 Queries to sets of sets

For queries with nested output, the sizes of the sets in the output is not bounded, hence there is no bound on the sizes of descriptors. It follows that the number of descriptors for a given query may be infinite, hence the case here is considerably more difficult than that for flat output. We did manage to handle certain cases. Specifically, we consider here fully generic queries whose output is a set of sets, and whose input is a set or a set of sets. This section is dedicated to proving the following theorem.

Theorem 6.1 *All fully generic queries from sets (resp. sets of sets) to sets of sets are expressible in $\mathcal{L}$. When the input is a flat set, this can be done without using one-each.*

Corollary 6.2 *All fully generic queries from sets or sets of sets to sets of sets are computable.*

6.1 Descriptors and meta-descriptors

For each canonical form, it is easy to find an expression in $\mathcal{L}$ for a query that gives an empty output ($\{\}$ or $\{\{\}\}$) on inputs of that form. Hence, from now we consider non-empty output. Two ideas facilitate our proof: a *simplification* of the representation of descriptors for the cases under consideration, and a generalization to *meta-descriptors*, that can be used to represent possibly infinite sets of descriptors.

Simplification of descriptors: Since the output is a set of flat sets, each descriptor in D_Q describes a flat set w.r.t. some input. That is, the $out(d)$ component of each descriptor is a flat set. $map(d)$ maps the members of this flat set *onto* the leaves of $in(d)$, the tree representing the input. Moreover, by the semantics of sets, no two members of $out(d)$ could be mapped to the same node of $in(d)$, that is, $map(d)$ is one-to-one. Clearly, given $in(d)$, it is possible to reconstruct $out(d)$ and $map(d)$, hence they are superfluous, and are omitted from now.

Assume below that T is a flat non-empty set, S is a set or a set of sets, $T \in Q(S)$, and d ($\equiv in(d)$) describes T w.r.t. S.

First, consider the case that S is a set. For a non-empty T, d is a tree consisting of a root labeled with S's canonical form and a pair (z, p) (where p is $<$ or $=$) and with children that are leaves, all labeled with z. Thus, the only factors that distinguish such a descriptor from others are p and the number k of leaves. Each descriptor can therefore be represented by (p, k).

If S is a set of sets, and T is non-empty, then d is a tree whose root is labeled with S's canonical form c and a pair $(\{z\}, p)$, and with children whose structure is that of a descriptor of a flat set, as considered above. Here also, the component $\{z\}$ in the root's label is redundant. Since the order of the children of the root is immaterial, and repetitions may occur, a descriptor can be represented as $\langle c; p; b\rangle$, where b is a bag of descriptors, each of the previous kind, namely pairs (p', k).

Example 6.3 If $S = \{a, b\}$, and $T = \{a\}$, then the representation of the descriptor of T w.r.t S is $(<, 1)$, meaning that T has one element from S, and S contains additional elements. If $S = \{\{a, b\}, \{c, d, e\}, \{\}\}$, and $T = \{a, b, c\}$, then the representation of the descriptor of T w.r.t S is $\langle c, =, \{\!\{(=, 2), (<, 1)\}\!\}\rangle$, where $c = cf(S) = \{\{z\}, \{\}\}$. Here, the first $=$ indicates that all non empty member sets of S participate in T. The pair $(=, 2)$ in the bag indicates that T contains all elements of a member set of S, that has two elements. The other pair indicates that T contains one element of another set in S, that contains additional elements. □

Dominance for simplified descriptors: Aside from the $<$-to-$=$ Property, there are other dominance properties that we use in our proofs. We now present a couple, as they apply to simplified descriptors.

downwards closed: *A descriptor dominates all descriptors obtained by replacing (p, k) by (p, k'), $1 \leq k' \leq k$.*

sum property: *A descriptor of the form $\langle c; p; b\rangle$ dominates all descriptors $\langle c; p; b'\rangle$, where b' is obtained from b by replacing elements $(<, k_1)$ and $(<, k_2)$ by $(<, k_1 + k_2)$.*

Note that downwards closedness applies to simplified descriptors of depths 1 and 2, whereas the sum property only applies to those of depth 2.

Generalizing descriptors to meta-descriptors: The observation here is that sets of descriptors describing a query often contain repetitions of similar components, which can be described compactly. By downwards closedness, (p, k) dominates (p, k') if $1 \leq k' < k$. Thus, we take (p, k) to represent the set $\{(p, k') \mid k' \leq k\}$; similarly, we take (p, ∞) to represent $\{(p, k) \mid 1 \leq k\}$. Finally, we take $(p, u)^+$ to represent an arbitrary (but positive) number of occurrences of (p, u) in a bag.

Definition 6.4 *A* **meta-descriptor** *of depth 1 is a pair (p, u) where p is $=$ or $<$, and u is either some finite number k, or ∞.*
A **meta-descriptor** *of depth 2 is $\langle c; p; b\rangle$ where c is a canonical form, p is either $<$ or $=$, and b is a bag whose elements are of the form l or l^+, where l is a meta-descriptor of depth 1.*

Note that, in particular, each regular descriptor is also a meta-descriptor. We use $d, d_1, \ldots$ for descriptors,and $\hat{d}, \hat{d}_1, \ldots$ for meta-descriptors. Capital letters are used for sets.

Definition 6.5 *We say that a meta-descriptor* $\hat{d}$ **denotes** *the set of descriptors that can be obtained from it by performing the following, in any order: (1) Replace each* l^+ *by a positive number of occurrences of* l*. (2) Replace each* ∞ *by some positive finite number. We write* $denotes(\hat{d})$ *for the set of descriptors denoted by* $\hat{d}$*. For a set of meta-descriptors* $\hat{D}$*,* $denotes(\hat{D}) = \bigcup\{denotes(\hat{d}) \mid \hat{d} \in \hat{D}\}$*.*

Note that the replacement order can be, w.l.o.g. from the top down. As an example of the above definition, $denotes((<, \infty)) = \{(<, 1), (<, 2), (<, 3), \ldots\}$, an infinite set of descriptors.

Definition 6.6 *The* **closure** *of a set of meta-descriptors* $\hat{D}$ *is* $closure(\hat{D}) = closure(denotes(\hat{D}))$*. We say* $\hat{D}$ **describes** *a query* Q *if* $closure(\hat{D}) \subseteq D_Q$*, and that it* **defines** *a query* Q *if* $closure(\hat{D}) = D_Q$*. If* $\hat{D} = \{\hat{d}\}$*, we also say that* $\hat{d}$ *describes, respectively defines,* Q*.*

Claim 6.7 $closure(\bigcup \hat{D}_i) = \bigcup closure(\hat{D}_i)$*, hence* $closure(\hat{D}) = \bigcup_{\hat{d} \in \hat{D}} closure(\hat{d})$*. It follows that If* $\hat{D}_1$ *defines* Q_1 *and* $\hat{D}_2$ *defines* Q_2*, then* $\hat{D}_1 \cup \hat{D}_2$ *defines* $Q_1 \cup Q_2$*.*

In the proofs below, we consider a query Q, and try to show that D_Q can be described by a finite union of (not necessarily disjoint) sets of descriptors, say $D_1, \ldots, D_n$, and each of these can be associated with a meta-descriptor. Specifically, for each D_i, we show the existence of some meta descriptor $\hat{d}_i$ such that $denotes(\hat{d}_i) \subseteq D_i$ and $D_i \subseteq closure(\hat{d}_i)$. It follows that $denotes(\bigcup\{\hat{d}_i\}) \subseteq D_Q$ and $closure(D_Q) \subseteq closure(\bigcup\{\hat{d}_i\})$. But for any fully generic query Q, D_Q is closed, hence it contains the closure of any subset. Thus, $closure(D_Q) = closure(\bigcup\{\hat{d}_i\})$. Clearly if we find queries $Q_1, \ldots, Q_n$ for the meta-descriptors, we are done. Thus, to prove theorem 6.1 it suffices to show:

Claim 6.8 *Let* Q *be a fully generic query whose input is a set (set of sets) and output is sets of sets. Then there exists (a) a finite set of meta descriptors* $\{\hat{d}_1, \ldots, \hat{d}_k\}$ *which defines* Q*, and (b) fully generic queries* $Q_1, \ldots, Q_k$ *expressible in* $\mathcal{L}$ *such that* $\hat{d}_i$ *defines* Q_i*.*

It is important to note that many meta-descriptors do not **define** any (fully generic) query, even if they do describe certain queries. For example, this is the case for all descriptors of the form $(=, k)$, k finite (proof omitted). In our search for a finite set of meta-descriptors, since we give each a corresponding query, we must avoid those that do not define queries.

The rest of this section is dedicated to the proof of the above claim. We consider the cases for the two input types separately. In each case the proof has two parts: (a) finding the finite set of descriptors, and (b) presenting a query for each of the descriptors found in step (a). We sketch the (a) parts; the (b) parts are described in appendix B.1, B.2, respectively.

6.2 Set inputs

(a): Finding a finite set of meta-descriptors.

We express D_Q (without empty descriptors) as a union $C_< \cup C_=$, where $C_<$ contains all descriptors of the form $(<,k)$ in D_Q, and $C_=$ contains those of the form $(=,k)$. If $\{k \mid (<,k) \in C_<\}$ is bounded by say k_0, let $\hat{d}_< = (<,k_0)$. Otherwise, let $\hat{d}_< = (<,\infty)$. In either case, $denotes(\hat{d}_<) = C_< \subseteq closure(\hat{d}_<)$, where the latter containment follows from our downward closedness results. The same analysis applies to $C_=$, giving two possibilities for a $\hat{d}_=$ for it. Now $(=,k)$ does not define a query, for any finite k. However, we show that if $\hat{d}_=$ is $(=,k)$, then $C_= \subseteq closure(C_<)$, hence we do not need to find a meta-descriptor for $C_=$, since we have one for $C_<$. In summary, we need to find queries for at most 2 meta-descriptors.

6.3 Set of set inputs

(a): Finding a finite set of meta-descriptors.

A descriptor now is $\langle c; p; b\rangle$. The proof here is considerably more difficult than that above, although the main ideas used there still apply. Intuitively, we confront two problems that didn't previously exist. Firstly, the size of descriptors (i.e. the size of bags b) can be arbitrarily large. We need to show that a finite number of meta-descriptors suffice to capture this infinite collection. Secondly, even if the size of bags is bounded, there could still conceivably be infinitely many "independent" descriptors, where no one dominates the other.

Recall that the canonical form c is fixed. Now, we also fix p, and until further notice, c, p are fixed, so we identify a descriptor with its bag b. The descriptors can be further divided into those in which b contains only inequalities, called *pure inequality* descriptors, denoted $D_<$, and those that have at least one equality denoted $D_=$. For each of these, we ask whether the number of inequalities is bounded or not, giving rise to four cases to consider. Let us call the number of inequalities in a bag its *inequality size*.

(1) Pure inequalities, unbounded inequality size: Using the sum property, we show that if the inequality size of bags in $D_<$ is unbounded, then the *values* in the pairs in them are also unbounded. That is, $D_<$ contains **all** bags with any positive inequality size and any values in the pairs. Since inequalities dominate equalities, by the <-to-= Property, we get that $D_<$ contains **all** possible bags, with both equalities and inequalities. Also, it is denoted by the meta-descriptor $\{\!\{(<,\infty)^+\}\!\}$.

(2) With equalities, unbounded inequality size: By a similar analysis we obtain that if the inequality size in the subclass of $D_=$ of bags with *precisely one* equality is unbounded, then it contains **all** bags with one equality and any positive inequality size, with any values in the pairs, and therefore the meta-descriptor $\{\!\{(=,\infty),(<,\infty)^+\}\!\}$ denotes it. Furthermore, using the <-to-= Property, its closure contains all of $D_=$. Finally (using the sum property and more) we show that if the inequality size of $D_=$ is unbounded, then it is unbounded for this subclass as well. Thus, in summary, for this case it suffices to find a query for the single meta-descriptor above.

(3) Pure inequalities, bounded inequality size: If the inequality size of $D_<$ is bounded, treat each size separately. For a size m, sort the elements of each bag in non-decreasing order. Now, if the numbers in the first positions of all bags are unbounded, then so are the numbers in all remaining positions. The meta-descriptor is $(<,\infty)^m$, where l^m is shorthand for l repeated m times. If the numbers in the first position are bounded by k, we have one pair for the meta-descriptor: $(<,k)$. Now split the bags into a different class for each value in $[1..k]$, and continue inductively to the remaining positions. This gives us a finite number of meta-descriptors, each of the form $\{\!\{(<,k_1),\ldots,(<,k_j),(<,\infty)^l\}\!\}$, where $m = j + l$.

(4) With equalities, bounded inequality size: The last case, by far the most complex, is when the inequality size of $D_=$ is bounded. Note that the number of *equalities* may still be unbounded. The proof can be summarized as follows (for the full proof see [Ta-97]): Divide $D_=$ into subclasses $D^i_=$ according to the number i of equalities. We show that for each i some descriptors are contained in $closure(D^{i-1}_=)$, hence can be removed. The remaining subclass, say $\bar{D}^i_=$, has the property that that it is bag cross product of a set of inequality bags W_i and a set of equality bags of size i. The latter turns out to be the set of **all** equality bags of size i, and can therefore be represented by $(=,\infty)^i$. It remains to find meta-descriptors for W_i. Since W_i has bounded inequality size, the results above imply that there is a finite set of meta-descriptors $\hat{D}_i$ for it. However, this does not suffice, since we have an infinite sequence $W_1, W_2, W_3, \ldots$ corresponding to $\bar{D}^1_=, \bar{D}^2_=, \bar{D}^3_=, \ldots$. We first point out that $W_1 \supseteq W_2 \supseteq \ldots$. The descriptors in these sets are all of bounded size. We then define a relation $\succeq$ on the corresponding finite sets of meta-descriptors, and we show that $\hat{D}_1 \succeq \hat{D}_2 \succeq \ldots$. The crux of the proof is to show that $\succeq$ is well-founded, using König's Lemma.[12] Hence in the sequence there is only a finite number of *distinct* sets, and therefore the same holds for the sequence of the W_i's. It follows that we can construct a finite set of meta-descriptors for the class $D_=$ as a whole. The meta descriptors obtained take the form $\{\!\{(<,k_1),\ldots,(<,k_j),(<,\infty)^l,(=,\infty)^m\}\!\}$, where l is finite, and m can be finite or $+$. (The $+$ is needed in the representation of the tail of the sequence, where the size of the equality bags for the smallest W_i is between i and ∞.)

This covers the analysis for all cases for fixed c, p. Note that unlike the case for flat input, even though we do guarantee a finite number of meta-descriptors for every query of a given type, there is no bound on their number. Similarly to the case for flat input, if p is $=$, some of the meta-descriptors obtained do not define queries, and as we did there, we show that the corresponding classes of descriptors are included in the closures of classes where p is $<$ and therefore are already covered.

[12] A finitely branching tree with no infinite branch is finite.

7 Conclusions

We have shown that when queries are invariant under all (many-to-many) mappings, there is a strong relationship between their inputs and outputs, that can be captured by patterns. The set of patterns of a query enjoys many elegant structural closure properties, some of which we presented (one in a simplified form). We are working towards a full understanding of the structure of the sets of patterns of fully generic queries. For the special cases treated here, the patterns can be represented very succinctly; our success in proving our conjecture for these cases is partially due to this representation.

For the types we considered, all fully generic queries at these types are computable. the same will hold for all types if our conjecture turns out to hold for them, in contrast to the case for all other known cases of genericity.

Finally, we note that for the case of set of set inputs, we seem to need the operation *one-each* (see Appendix B.2), which to our knowledge has not been considered in the literature on complex object languages so far. Whether this is the only operation needed to obtain a complete language for all types still remains open. Also, while we have not been able to express it using the other operations of $\mathcal{L}$, we have no proof of its independence.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, pages 110 – 120, 1979.

[BBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In Biskup and Hull, editors, *International Conference on Database Theory*, Berlin, Germany, October 1992. Springer-Verlag. LNCS 646.

[BMT96] C. Beeri, T. Milo, and P. Ta-Shma. On genericity and parametricity. In *ACM Symposium on Principles of Database Systems*, Montreal, Canada, June 1996.

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[CH80] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156 – 178, 1980.

[HY84] R. Hull and C. K. Yap. The format model : a theory of database organization. *Journal of the ACM*, 31(3):518 – 537, 1984.

[Ley] Michael Ley. Databases and logic programming, a bibliography server:. `http://www.informatik.uni-trier.de/~ley/db/index.html`.

[LW93] Leonid Libkin and Limsoon Wong. Semantic representations and query languages for or-sets. In *ACM Symposium on Principles of Database Systems*, Washington DC, 1993.

[PSV96] C.H. Papadimitriou, D. Suciu, and V. Vianu. Topological queries in spatial databases. In *ACM Symposium on Principles of Database Systems*, pages 81 – 92, Montreal, Canada, 1996.

[PVV94] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *ACM Symposium on Principles of Database Systems*, pages 279 – 288, 1994.

[Ta-97] Paula Ta-Shma. *Genericity in Database Query Languages.* PhD thesis, The Hebrew University of Jerusalem, 1997.

[Won93] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *ACM Symposium on Principles of Database Systems*, pages 26 – 36, Washington, DC, 1993.

A Construction of queries from nested input to flat output

Let $D_Q = \{d_1, ..., d_k\}$ be the set of descriptors for Q, where the descriptor $d_i = \langle in(d_i), out(d_i), map(d_i)\rangle$. We construct a query q_i (of the same type as Q) for each d_i, s.t. $D_Q = D_{q_1} \cup \ldots \cup D_{q_k}$, and therefore, $Q = q_1 \cup \ldots \cup q_k$.

Given d_i, $q_i = q_i^{out} \circ q_i^{in}$, where q_i^{in}, q_i^{out} deal with $in(d)$ and $out(d)$ respectively. q_i^{in} is constructed inductively on the structure of $in(d)$ as follows, and returns tuples whose arity is the number of leaves of $in(d)$. Let q^n be the query associated with node n.

Node type	Children	Query
Leaf m		$q^m(x) = single([x])$
Tuple m	$m_1, \ldots, m_j$ (in that order)	$q^m(x) = q^{m_1}(\$1(x)) \times \ldots \times q^{m_j}(\$j(x))$
Set m	$m_1, \ldots, m_j$ of canonical form $c_1, \ldots, c_j$	$q^m(x) =$ $flatten(map(cond(c_1, q^{m_1}, \{\}))(x)) \times \ldots$ $\times flatten(map(cond(c_j, q^{m_j}, \{\}))(x))$

$cond(c, q, \{\})$ here is a shorthand for $\lambda y.cond(cf(y) = c, q(y), \{\})$. [13]

q_i^{out} has the form $map(f_i)$, where f_i is a function on tuples which possibly changes column order and duplicates certain columns (also expressible in $\mathcal{L}$), according to the relationship between $in(d), out(d)$, as expressed in $map(d)$.

Example A.1 Suppose d_i describes the tuple $t = [a, b, a]$ w.r.t. a set $S = \{\{\{a\}, \{b\}\}, \{\{c\}\}, \{\{\}, \{d\}\}\}$. The *in* part of d_i is shown in figure 2.
Let c_1, c_2, c_3 denote the canonical forms $\{\{\{z\}\}, \{\{\}, \{z\}\}\}$, $\{\{z\}\}$, $\{z\}$ resp. Using the above construction,

$q_i^{in} = flatten(map(cond(c_1, q_{temp} \times q_{temp}, \{\})))$,

where

$q_{temp} = flatten(map(cond(c_2, flatten(map(cond(c_3, single([x]), \{\}))), \{\})))$.

For example $q_i^{in}(S) = \{[a, b], [b, a], [b, b], [a, a], [c, c]\}$.

The *out* part of d_i is a 3-tuple where first and third columns point to the left branch, and second column points to the right branch. Therefore, we need

[13] $\mathcal{L}$ is variable free, although we use variables for readability.

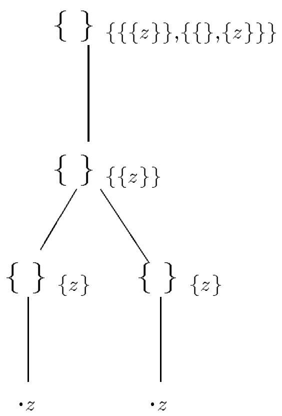


Fig. 2. The *in* component of descriptor d_i

to duplicate the first column of the above query into the third column : $q_i^{out} = map([\$1, \$2, \$1])(x)$.

For example $q_i(S) = q_i^{out}(q_i^{in}(S)) = \{[a, b, a], [b, a, b], [b, b, b], [a, a, a], [c, c, c]\}$.

Note that q_i produces the desired tuple $[a, b, a]$ but additional ones as well. We show in the full paper that the descriptors for all other tuples produced by a query q_i are dominated by d_i, thus belong to D_Q anyhow. So no new descriptors are added in the construction. □

B Queries for meta-descriptors

For each meta-descriptor $\hat{d}_i$ which arose in section 6, we now present a query Q_i such that $closure(\hat{d}_i) = D_{Q_i}$. For each case (output being a set or set of sets) the resulting query has the form $Q_1 \cup \ldots \cup Q_k$.

B.1 Set inputs

The appropriate queries appear in the following table.

Meta-descriptor	Query
$(<, \infty)$	$powerset \setminus \emptyset$
$(<, k)$	$\mathcal{P}^k \setminus \emptyset$
$(=, \infty)$	$single$
c	if $cf(x) = c$ then $\{\{\}\}$ else $\{\}$

$\mathcal{P}^k(S)$ returns all subsets of S of size at most k.

Claim B.1 *Each of the above queries can be expressed in $\mathcal{L}$, without one-each.*

B.2 Set of set inputs

First, we show how to compose queries for more complicated meta-descriptors from those for simpler ones. Given queries defined by two meta-descriptors $\langle c; <; b_1\rangle$ and $\langle c; <; b_2\rangle$, we compose the query defined by $\langle c; <; b_1 \uplus b_2\rangle$, where $\uplus$ denotes bag union.

Claim B.2 *If* $closure(\langle c; <; b_1\rangle) = D_{Q_1}$ *and* $closure(\langle c; <; b_2\rangle) = D_{Q_2}$ *then* $closure(\langle c; <; b_1 \uplus b_2\rangle) = D_Q$*, where* $Q(S) = map(\$1 \cup \$2)(Q_1(S) \times Q_2(S))$.

Therefore, meta-descriptors with root $<$ of finite size can be constructed from meta-descriptors of size 1, and we only present these. For root $<$ we also present the possible meta-descriptors of infinite size (i.e. with $+$). In the second part of the table, we present queries for all meta-descriptors which arise for root $=$.

In the queries below, for readability, we allow queries to use variables, even though formally $\mathcal{L}$ is variable free. It should be clear that this is only syntactic sugar. We assume variable S denotes the query input.

Meta-descriptor	Query		
$\langle c\rangle$	if cf(S) = c then	$\{\{\}\}$	else $\emptyset$
$\langle c, <, \{\!\{(<, k)\}\!\}\rangle$	"	$\mathit{flatten} \circ map(\mathcal{P}^k \setminus \emptyset)$	"
$\langle c, <, \{\!\{(<, \infty)\}\!\}\rangle$	"	$\mathit{flatten} \circ map(powerset \setminus \emptyset)$	"
$\langle c, <, \{\!\{(=, \infty)\}\!\}\rangle$	"	id	"
$\langle c, <, \{\!\{(<, \infty)^+\}\!\}\rangle$	"	$(powerset \setminus \emptyset) \circ \mathit{flatten}$	"
$\langle c, <, \{\!\{(=, \infty)^+\}\!\}\rangle$	"	$map(\mathit{flatten}) \circ (powerset \setminus \emptyset)$	"
$\langle c, =, \{\!\{(<, \infty)^+\}\!\}\rangle$	"	$\mathit{one\text{-}each}$	"
$\langle c, =, \{\!\{(=, \infty)^+\}\!\}\rangle$	"	$single \circ \mathit{flatten}$	"
$\langle c, =, \{\!\{(=, \infty), (<, \infty)^+\}\!\}\rangle$	"	$map(\$1 \cup \$2)(\mathit{one\text{-}each}(S) \times S)$	"

Claim B.3 *Each of the above queries can be expressed in* $\mathcal{L}$.

Unlike for the flat input case, *one-each* is necessary here.

This article was processed using the LaTeX macro package with LLNCS style