On Genericity and Parametricity*

Extended Abstract

Catriel Beeri
Hebrew University [†]

Tova Milo
Tel-Aviv University [‡]

Paula Ta-Shma
Hebrew University [§]

1 Introduction

Genericity is a property of queries expressed in declarative query languages. The idea, first expressed in [2, 7], is that data values are uninterpreted, hence a query should be invariant under isomorphisms (i.e. element renaming) of databases. This has been generalized, following [10], to account for preservation of constants. This concept of genericity is intimately related to classical relational database research, that considered databases as constructed over an abstract domain of uninterpreted elements. However, the scope of database models has vastly expanded in the last decade. Multiple domains, use of domain functions and predicates, user-defined types, and bulk types — these are essential components of recent data models. Genericity is central to many results in the theory of databases, yet little attention has been paid to it in the development of the new models. This paper tries to amend this situation.

We show in the paper that there is a rich structure of genericity concepts, and argue that genericity provides insight about the relationships between the properties of data structures used in a data model, the data types used in instances of a data model, and queries. Several recent papers have indicated the role of having different notions of genericity [9, 13]. On a more pragmatic level, genericity can be used as a tool for proving inexpressibility results: If one shows that all queries in a language are of a certain genericity class, then queries not in the class are not expressible. We follow Chandra [6] in presenting a few such results. Also, we argue that genericity can be related to query optimization, since it deals with possible commutativity of a query and some mappings. By refining the notion of genericity, we open the way for providing more information about queries. Given a query, the interesting question is not whether it is generic but rather what is the tightest genericity class for it.

The following factors are relevant to genericity: the functions and predicates of the base types, the bulk type constructors and their nesting, and the structure of the query, particularly its type. The issue of how to account for the use of interpreted functions and predicates was briefly discussed in [2, 7], with almost no follow-up in the literature. This is treated in the first part of the paper, Sections 2 and 3. To understand the impact of functions and predicates on genericity, we start with the case where no functions or predicates – not even the equality predicate – are used. This leads us to consider mappings that do not necessarily preserve equality of values, that is homomorphisms, or general relations. To our know'edge, there has been no study of invariance unde such mappings in the database area, with the exception of Chandra's paper [6], one of the starting points for our work. The combination of general relational mappings and set (or bag, list) constructors provides *several* interesting notions

*Work suported by grants from the Israel Science Foundation, administered by the Israeli Academy of Sciences.

[†]Department of Computer Science, The Hebrew University, Jerusalem 91904, Israel, beeri@cs.huji.ac.il

[‡]Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel, milo@math.tau.ac.il

[§]Department of Computer Science, The Hebrew University, Jerusalem 91904, Israel, paula@cs.huji.ac.il of genericity. We show how to account for both constants and interpreted functions and predicates; for the special case of equality (or a total order), we arrive back at injective functional mappings. We also analyze and classify the genericity properties of many well known database operations, particularly w.r.t the degree to which they use equality.

The relationship between the type structure of queries, in particular of polymorphic queries, and invariance under mappings between different types, has not been considered in classical genericity, nor in our refinements so far. This is studied in the second part, Section 4. Typed lambda calculi are the framework of choice for studying type structure and polymorphism of functions. The 2nd-order calculus, also known as *System-F* [14, 8], is a polymorphic language, more expressive than all current query languages of interest. The *parametricity theorem* states a property of functions expressible in this language; in a formulation due to Wadler [15], it is that the function commutes with certain mappings determined by its type. We show this is closely related to genericity (where queries are required to commute with mappings), and in fact to be more general in several ways. Unfortunately, the 2nd-order λ calculus can express lists, but not sets. A similar language for sets is not known to us. We present a technique to relate sets and lists, and use it to transfer parametricity from lists to sets, for a restricted set of types, that is, however, sufficiently expressive to cover the languages for complex values in [1, 4, 5], *powerset* included. We give some examples to illustrate the potential relationship between genericity/paramtericity and query optimization.

We conclude with a discussion in Section 5.

2 Definitions, Notions of Genericity

We assume that a database instance may be defined over a signature Σ, namely a collection of base types with interpreted functions and predicates. We assume Σ contains the type bool.

Definition 2.1 A *complex value type* over Σ is a tree whose leaves are base types, $d_1, \ldots, d_n$, and with internal nodes labeled with type constructors $\times$, $\{\}$, $\{\!\!\{\}\!\!\}$ and $<>$, for forming products (i.e. pairs or tuples) sets, bags[1] and lists, resp. □

Given domains for the base types, the domains for all types are defined as usual.[2] We write $x : \alpha$ to denote that x is of type α. Since databases can be viewed as tuples of complex values, we deal with functions from complex values to complex values. In the following, for simplicity, we use d_i to name both a base type and its domain.

2.1 Motivation

The essential idea of genericity is that a query specifies some pattern(s); hence, if two databases contain the "same" patterns, they should generate the "same" answer for the query. Traditionally, 'having the same patterns' has been interpreted as 'related by an **iso**morphism'. But in many cases the same holds for 'related by a **homo**morphism'.

Example 2.2 The query $Q_1 = \pi_{\$1,\$3}(R \bowtie R)$ (i.e., $R \circ R$), when applied on the relation $r_1 = \{(e,f),(i,f),(e,j),(i,j),(f,g),(j,g)\}$, returns $\{(e,g),(i,g)\}$.
The relation $r_2 = \{(a,b),(b,c)\}$ is a homomorphic image of r_1, under the homomorphism $h(e) = h(i) = a$, $h(f) = h(j) = b$, $h(g) = c$, and so is the query result on this relation, $\{(a,c)\}$. That is, $Q_1(h(r_1)) = h(Q_1(r_1))$ holds. □

However, one should exercise care in generalizing genericity to preservation w.r.t homomorphic mappings. If r_3 is the relation obtained by removing from r_1 the tuples $(e,f),(i,f),(j,g)$, then the query returns $\emptyset$. But $h(r_3)$ is still r_2, and thus the result of the query on this relation is not the homomorphic image of $\emptyset$. The problem is that, because h *identifies* elements, it may create patterns in r_2 that did not exist in r_3, and that may influence the query result. Note, however, that this problem does not arise for all queries. For example, the query $Q_2 = R \times R$ is invariant under all mappings, regardless of which domain elements are being identified.

Chandra defines [6]: A function h is a *strong homomorphism* from a relation r_1 to a relation r_2 if $r_1(\bar{x}) \leftrightarrow r_2(h(\bar{x}))$. It can be seen that h is a strong homomorphism from r_1 to r_2, but only a regular homomorphism from r_3 to r_2. The query Q_2 is preserved by regular homomorphisms, but Q_1 is preserved only under strong homomorphisms.

[1] In the full paper we present definitions and results for bags, but in this abstract, for brevity, bags are not mentioned further.

[2] We allow infinite complex values.

We conclude that queries are often preserved under homomorphic mappings, and that some queries are more sensitive to the degree of preservation of equality than others.

2.2 Relational Mappings

Classical genericity of queries is defined as invariance w.r.t. to extensions of injective functions on the base domains. As seen above, it is of interest to consider non-injective mappings. Now, invariance applies symmetrically to both directions of a mapping. But, the inverse of a function, even of a strong homomorphism, is not necessarily a function! So, let us generalize to relations. We thus consider general binary relations on base domains, and how they can be extended to complex values. For example,

$K = \{(e,a),(i,a),(f,b),(j,b),(g,c),(g,d)\}$

is a general mapping, that is not functional in either direction. We also do not require mappings to be total or surjective on the mapped domains. To avoid confusion with the standard use of 'relation' in databases, we refer to these relations as *mappings*. We say that $H(x,x')$ holds if the pair (x,x') is in the mapping H. Note that mappings are *typed*. If mapping H has domain τ and codomain τ', we write $H : \tau \times \tau'$.

To be able to extend mappings from base domains to complex types, we associate a *mapping constructor* with each type constructor. Then if $C(X_1, \ldots X_n)$[3] is a type constructor, and we have mappings $H_i : \tau_i \times \tau'_i$, $i = 1 \ldots n$, we can extend them to a mapping between $C(\tau_1, \ldots, \tau_n)$ and $C(\tau'_1, \ldots, \tau'_n)$, denoted by $C(H_1, \ldots, H_n)$. (Specific notations for the type constructors of def. 2.1 are introduced below.) For complex type expressions, we can then use induction on type structure.

For tuple construction there is only one reasonable choice — the extension is defined component wise; similarly for lists, since order needs to be preserved. More formally,

Definition 2.3 Let $K_i : \tau_i \times \tau'_i, i = 1 \ldots n$ be given mappings. Their extension to a mapping between the tuple types $\tau_1 \times \ldots \times \tau_n$ and $\tau'_1 \times \ldots \times \tau'_n$, denoted $K_1 \times \ldots \times K_n$, is defined as follows: $K_1 \times \ldots \times K_n(\bar{x}, \bar{y})$ holds, for n-tuples $\bar{x} : \times \tau_i$, $\bar{y} : \times \tau'_i$, iff $K_i(x_i, y_i)$ holds for each i. □

[3]The X_i are type variables.

Definition 2.4 Let $K : \tau \times \tau'$ be some mapping. Its extension to a mapping between the list types $\langle \tau \rangle$ and $\langle \tau' \rangle$, denoted $\langle K \rangle$, is defined as follows: For lists $l : \langle \tau \rangle$, $l' : \langle \tau' \rangle$, $\langle K \rangle (l, l')$ holds iff l, l' are of equal lengths, and $K(l_i, l'_i)$ holds for each pair of elements of the lists in corresponding positions. □

However, for the set constructor there are (at least) two possibilities for extension.

Definition 2.5 Let $K : \tau \times \tau'$ be some mapping. Then $\{K\}^{rel}$, $\{K\}^{strong}$ are extensions of K to mappings over the types $\{\tau\}, \{\tau'\}$, s.t. for every $R_1 : \{\tau\}, R_2 : \{\tau'\}$,

1. $\{K\}^{rel}(R_1, R_2)$ holds iff $\forall x_1 \in R_1 \; \exists x_2 \in R_2$ s.t. $K(x_1, x_2)$, and $\forall x_2 \in R_2 \; \exists x_1 \in R_1$ s.t. $K(x_1, x_2)$.
2. $\{K\}^{strong}(R_1, R_2)$ holds iff $\{K\}^{rel}(R_1, R_2)$ and each of R_1, R_2 is the maximal set that stands in the $\{K\}^{rel}$ relation to the other.

We refer to *rel*, *strong* as *extension modes*. □

The notions of *strong*, *rel* generalize Chandra's *strong homomorphism* and unrestricted homomorphisms, respectively to general mappings.

Example 2.6 Consider the relations r_1, r_2, r_3 and the mapping h of Example 2.2. The mapping h can be extended (as explained above) to a mapping $h \times h$ over pairs. This mapping can then be further extended to a mapping $\{h \times h\}^x$, for $x =$ *rel*, *strong*, over sets of pairs. It is easy to verify that $\{h \times h\}^x(r_1, r_2)$ holds, for $x =$ *rel*, *strong*. On the other hand, $\{h \times h\}^{rel}(r_3, r_2)$ holds, but $\{h \times h\}^{strong}(r_3, r_2)$ does not. □

Mappings are not required to be total or surjective. For $\{K\}^x(R_1, R_2)$ to hold, for any x, it must hold that $R_1 \subseteq dom(K) \wedge R_2 \subseteq co\text{-}dom(K)$. Thus, $\{K\}^x(R_1, R_2)$ may be false either because K is "too small", or because it does not map R_1 to R_2.

Definition 2.7 A *type expression* $T(X_1, \ldots, X_n)$ is a tree with type variables (from $X_1, \ldots X_n$) at the leaves, and type constructors at the internal nodes. $T(\tau_1/X_1, \ldots, \tau_n/X_n)$ denotes the result of substituting τ_i for X_i in T. Given lists of base types $d_1, \ldots, d_n, d'_1, \ldots, d'_n$, the types $T_1 = T(d_1/X_1, \ldots, d_n/X_n)$ and $T'_1 = T(d'_1/X_1, \ldots, d'_n/X_n)$ are called *associated types*. □

Now consider how a given a family of mappings $H = \{H_i : d_i \times d'_i\}$, on base domains, can be *extended* to a mapping between T_1, T'_1. There are many ways for doing that. Firstly, if two mappings share a domain and codomain, and the domain type appears in two or more positions in a product in T_1, we can select either one for each position. Hence, we disallow H where two mappings have the same domain and codomain. Secondly, we can use any of the extension modes for each occurrence of a set constructor. If we label each set node of T with an extension mode, then the there is a unique mapping constructor associated with each internal node, and the tree represents an n-ary function on mappings: If the variables are substituted by $H_1, \ldots, H_n$ (subject to the constraints above), resp., then we obtain a mapping $\mathcal{T}(H_1, \ldots, H_n) : T_1 \times T'_1$. In the sequel, we do not consider further 'mixed extensions'. We denote by H^{rel} (H^{strong}) the family of extensions of H to mappings on all (associated pairs of) nested types, where the *rel* (*strong*) mode is used at **all** occurrences of set constructors. We write $H^x(R_1, R_2)$, where x is *rel* or *strong*, when R_1, R_2 stand in the relationship H^x . Essentially, we consider H^x, as a many-sorted mapping, i.e., a (possibly infinite) collection of mappings $\{H_i\}$, where H_i is a mapping between pair of domains of associated types. For a class $\mathcal{H}$ of mappings on base domains, we denote by $\mathcal{H}^x$ the class $\{H^x \mid H \in \mathcal{H}\}$.

We summarize some useful properties.

Proposition 2.8 .
(i) If H is total or surjective, then so is H^{rel}.
(ii) If T, T' are associated types that contain a set constructor, then H^{strong} on them is injective.
(iii) If H_1, H_2, H_3 are mappings s.t. $H_3 = H_1 \circ H_2$, then $H_3^{rel} = H_1^{rel} \circ H_2^{rel}$. If further, co-dom$(H_1)$ = dom(H_2), then $H_3^{strong} = H_1^{strong} \circ H_2^{strong}$.
(iv) For every mapping H, $\{(H^{-1})\}^x = (\{H\}^x)^{-1}$, for $x = rel, strong$. □

2.3 Invariance and Genericity

We are now in position to define genericity.

Definition 2.9 .
(i) A function Q *is invariant*[4] under H^x, if for any two legal inputs to Q, R_1, R_2, if $H^x(R_1, R_2)$ holds, then so does $H^x(Q(R_1), Q(R_2))$.
(ii) A function Q on a class $\mathcal{D}$ of complex types, is *generic* w.r.t. a class $\mathcal{H}$ of mappings on base domains, and extension mode x, denoted $x - Gen_{\mathcal{D}}(\mathcal{H})$, if it is invariant under all the mappings in $\mathcal{H}^x$. The queries in the class are called *x-generic* functions (On $\mathcal{D}$ w.r.t, $\mathcal{H}$). □

[4]This is related to *logical relations* used in the study of typed languages [12].

For example, the query $Q_3 \equiv \pi_{\$1}(R)$ is invariant under both H^{rel} and H^{strong}, for each H, because two tuples are related iff all their attributes are related. Thus Q_3 is *x-generic* w.r.t all possible mappings. The query $Q_4 \equiv \sigma_{\$1=\$2}(R)$ is not invariant under all H^{rel} mappings. For example, the relations $R_1 = \{[a, a]\}$, $R_2 = \{[b, c]\}$ are related by H^{rel}, where $H = \{(a, b), (a, c)\}$, but $Q_4(R_1) = R_1$, and $Q_4(R_2) = \emptyset$ are not. This is because in the *rel* extension for tuples, attributes can be mapped independently of each other, even if they have the same value. This is immaterial for injective mappings, and indeed Q_4 is *rel*-generic w.r.t such mappings.

A well-known property of genericity:

Proposition 2.10 *If Q is in $Gen_{\mathcal{D}}(\mathcal{H})$ and $\mathcal{D}' \subseteq \mathcal{D}, \mathcal{H}' \subseteq \mathcal{H}$, then Q is also in $Gen_{\mathcal{D}'}(\mathcal{H}')$. In particular, for a fixed $\mathcal{D}$, we have $\mathcal{H}' \subseteq \mathcal{H} \rightarrow Gen_{\mathcal{D}}(\mathcal{H}) \subseteq Gen_{\mathcal{D}}(\mathcal{H}')$.* □

That is, a smaller class of mappings induces a larger class of generic queries. For example, Q_4 above is not *rel-* generic w.r.t all mappings, but only w.r.t. injective mappings. For a given class of databases and an extension mode there exists a hierarchy of genericity classes. One natural path in the hierarchy goes from all mappings, to functional mappings (whose extensions are homomorphisms), then to one-to-one functions (whose extensions are isomorphisms).

We saw that relational and injective mappings define different genericity classes, witness Q_4. But, it turns out that under quite general conditions general and functional mappings define the same classes. We present it in a restricted, yet representative form: We say that a query is *defined at all types* if it has type $T(X_1, \ldots, X_n) \rightarrow S(X_1, \ldots, X_n)$ where the X_i are type variables. Cross product, union, projection, are examples.

Proposition 2.11 *A query defined at all types is x-generic w.r.t all functional mappings iff it is x-generic w.r.t all mappings, for x = rel,strong.* □

The proof uses the well known idea that a many-to-many mapping can be decomposed into two many-to-one mappings.

2.4 Generalizing Genericity

The class of functions that are generic w.r.t all mappings (even if only total injective mappings are considered) is too small — it excludes many useful queries. For example, for $Q_5 \equiv \sigma_{\$1=7}$. It is easy to find sets R_1, R_2 and an injective mapping H from **int** onto itself, such that $H^x(R_1, R_2)$, but $H^x(Q_5(R_1), Q_5(R_2))$ does not hold, for both modes. The point is that queries that mention constants and functions of the domains are invariant only under mappings that take these constants and functions into account.

2.4.1 First-Order Constants

Generalizing [10], we say that a mapping H *preserves* a (first-order) constant c if $H(c,c)$ holds; it *strictly preserves* c if additionally whenever $H(x,y)$ holds, $x = c$ iff $y = c$. Equivalently, H preserves c if $H^{rel}(\{c\},\{c\})$ holds, and strictly preserves it if $H^{strong}(\{c\},\{c\})$. Preservation of c allows H to associate c with other values; strict preservation disallows it.

A query is (strictly) x-c-generic, x being *rel*, *strong*, if it is x-generic w.r.t to all mappings that (strictly) preserve c. These notions of preservation and genericity are extended to sets of constants as usual. Combinations that mix strict and regular preservations are possible. Strict preservation implies preservation, but not vice versa, hence it allows more functions to be generic. Q_5 above is *rel*-generic for mappings that strictly preserve 7, but is not for those that only preserve it.

The accepted notion of genericity is w.r.t. a finite set of constants, since any query, being a finite expression, mentions only a finite set of constants. The following was noted by Chandra [6].

Lemma 2.12 *For any finite set C of constants from an infinite domain the query* **even** *is not strictly x-C-generic, for $x = rel, strong$. For regular preservation, C can be arbitrary.* □

2.5 Second and Higher-Order Constants

even is not C-generic; similarly $\sigma_{\$1>\$2}$ is not C-generic for a finite C. We would like both to be considered generic. Intuitively, we need to account for the use of $>$ in the last query, the (implicit) use of $=$ in *even*, and in general for the use of interpreted functions and predicates in queries.

We say that a mapping H^x *preserves* a function f if f is invariant under (i.e., generic w.r.t) H^x; i.e. if $H^x(x,y)$ than $H^x(f(x), f(y))$ also holds. For predicates, there are two possible approaches for defining preservation. A predicate can be viewed as a complex value — a (possibly infinite) set of pairs, or as a boolean-valued function. We present here only the second: Initially, say that a mapping H^x *preserves* a predicate p if p *as a function* is invariant under (i.e., generic w.r.t) H^x. This leaves one detail unclear: how to capture the intended special semantics of the truth values. For that, we strengthen our definition and require the mapping H being considered to be the identity on *bool*. With this, we have, for example:

Proposition 2.13 *Under the functional interpretation, H^x preserves p iff it preserves $\neg p$.* □

In the full paper we compare the various notions. As usual, a query Q is x-C-generic, where C is a set of first and second-order constants, if it is x-generic, w.r.t all the mappings that preserve each element of C.

For example, the query Q_5 is both *rel* and *strong*-$\{=, 7\}$ generic. i.e. invariant under all the mappings that preserve both the equality predicate, and the number 7. (Note that only injective mappings, that induce isomorphisms, preserve equality.) In fact, the query is invariant under a larger class of mappings — those that preserve the unary predicate " $=_7$ ". This , therefore, is a more accurate genericity classification for it.

3 Properties of Genericity

In this section we present some properties of genericity, and also classify many query language operations in terms of their genericity properties. We treat operations on (flat) relational databases. (In the full paper we deal also with nested relations/complex value operations.)

3.1 The Full Genericity Classes

We refer to queries that are generic w.r.t. the full class of mappings as (*rel*/*strong*) *fully generic*. These are the smallest classes of generic queries.

To be able to state closure of genericity classes under operations, we view operators like $\cup$ as function(query) constructors

Proposition 3.1 *All genericity classes of relational queries are closed under composition, $\times$ and $\cup$. If f is in some genericity class, then the class is closed under map(f). Also, $\hat{\emptyset}$ (returning the empty relation), Id (identity) and tuple projection are fully generic for both extension modes* $\square$

Corollary 3.2 *The sub-language of the relational algebra, consisting of the operations $\times, \Pi, \cup$, and of the base queries $\hat{\emptyset}$ and R, where R is a relation name, is fully generic, for both modes.* $\square$

We can give calculus connectives and quantifiers similar interpretation as function constructors, and obtain a corresponding result; in particular:

Proposition 3.3 *The functions expressed in the relational calculus, using only atomic formulas $R(\bar{x})$ with no repeated variables, using $\vee$ on formulas with the same free variables, using $\wedge$ on formulas with disjoint variables sets, and using $\exists$, are fully generic for both modes.* $\square$

These results provide insight about which queries are fully generic, for both modes: those expressed by operations that do not use equality in any way. For operations that use equality, we have:

Proposition 3.4 *The class of rel-fully C-generic queries, for any finite set of first-order constants from an infinite domain, is not closed under the operations $-, \cap$.* $\square$

These two operations are, however, *strong*-fully generic, as seen below. With the following Proposition (also from [6]), we have that the classes of *rel*/*strong*-fully generic queries are incomparable.

Proposition 3.5 *The query $eq_{adom(d)}$ (computing the equality relation over the active domain of a database), is not strong-fully generic, but is rel-fully generic.* $\square$

Note that although $eq_{adom(d)}$ is *rel*-fully generic, since mappings are not required to preserve $=$, operations that require equality (like $-$ and $\cap$) are not *rel*-fully generic.

3.2 Fully *strong*-generic Queries

The query $\sigma_{\$i=\$j}(R) = \{t \in R \mid t.i = t.j\}$ is also not *strong*-generic. (For otherwise, using the results above we would have that $eq_{adom(d)}$ is *strong*-fully generic, in contradiction to Proposition 3.5.) However, Chandra in [6] observed that *strong*-fully generic queries do capture some notion of equality. He defined a variant: $\hat{\sigma}_{\$i=\$j}(R) = \{\pi_{\bar{\$_j}}(t) \mid t \in R,\ t.i = t.j\}$, where $\pi_{\bar{\$_j}}$ denotes the projection *out* of the column j, and proved:

Proposition 3.6 *(*Chandra*) All classes of strong-generic functions are closed under the relational operations $\cup, \cap, \Pi, \times, -, \hat{\sigma}$ (hence also under complement w.r.t the active domain).*

Thus, *strong* genericity captures certain usage of equality in queries, but does not allow to show equality in the output. In particular, this version of selection eliminates one of the two occurrences of the equal values in a tuple, hence we cannot generate the equality relation. $\hat{\sigma}_{\$1=\$2}(R)$ is *strong*-fully generic, but $\sigma_{\$1=\$2}(R)$ is not. (A similar result for the calculus is omitted.) These results distinguish four sub-languages of relational algebra (calculus): One that uses no equality whatsoever, one that allows its use in the query but not in its output, one that allows its use in the output but not in the query (e.g. $x, x \mid r(x)$), and one that allows full usage of equality, and is thus generic only w.r.t 1-1 mappings.

In the full paper we present results about *fixpoint* and *while* operations.

3.3 The Full Domain, and Domain Independence

Proposition 3.6, talks about the active domain. What about full domain semantics ? The query $\{t \mid \neg R(t)\}$, that returns $\overline{R}$, the complement of R, is not fully generic for either mode, since mappings are not required to be total or surjective. A mapping may not be defined on complements of related relations. In fact, one can express the property of being *domain independent* for a query by saying that it is fully generic. If only total and surjective mappings are considered then complement becomes generic.

Proposition 3.7 *Let H be a total and surjective mapping. Then for each two (not necessarily finite) sets of tuples R, R', $H^{strong}(R, R')$ iff $H^{strong}(\overline{R}, \overline{R'})$.* $\square$

Thus the active domain can be replaced, in Proposition 3.6, by full domain if only total and surjective mappings are considered. We also have that

Proposition 3.8 *A query Q whose output is a set of (flat) tuples is strong-generic w.r.t to a class of total and surjective mappings iff $\overline{Q}$ is.* □

We conclude by noting that known results can be explained in terms of genericity-related properties, rather than language-related properties. Let *adom* denote the active domain of a database.

Theorem 3.9 *Let Q be a relational query, that is x-generic w.r.t the total and surjective mappings. If on a given database its result contains a tuple with a component from $\overline{adom}$, then it contains every tuple obtained from it by replacing this component by any other element from $\overline{adom}$.* □

This is a simple instance of the *four Russians' theorem* [3], stated as a genericity-related result.

4 Parametricity

Parametricity is a property of functions, very similar to genericity. We proceed to present the 2nd-order λ calculus, the parametricity theorem for the functions expressible in it, and its extensions and applications to our context.

4.1 The Parametricity Theorem

The 2nd-order λ-calculus is an expressive language with a polymorphic type system. We present a cursory description. The language has both value and type expressions. In the pure language, a *type expression* is either a base type, a type variable, or one of $S \rightarrow T, \forall X.T$, where S, T are type expressions; $\rightarrow, \forall X.$ are the *function* type constructor, and *universal quantification* on type variables. Free and bound variables are defined as usual. A **closed type** is a type expression without free variables. W.l.o.g., we assume that in $\forall X.T$, X (possibly other variables also) is free in T, and write $\forall X.T(X)$. The value language has *lambda*-abstraction, $\lambda x : T.e$ (T is a type expression)[5], application, $e_1 e_2$, and additionally *abstraction on types*, $\Lambda X.e$, and application to a type, $e[\alpha]$. If $f : \forall X.T(X)$, then $f[\alpha] : T(\alpha/X)$.

For example, $I = \Lambda X.\lambda x : X.x$ is the universal identity function, of type $\forall X.X \rightarrow X$, and $I[int]$, obtained by application to type int, is the identity function on int, of type $int \rightarrow int$. I is **polymorphic** – its type has $\forall$; $I[int]$ is **monomorphic**, its type doesn't contain any type variables or $\forall$. *Intuitively*, I, represents a collection of functions, indexed by types; $I[\alpha]$ is the α'th component. For a general discussion, and further references see [12]. We will not refer to the value part of the language further.

Regarding types, both products (tuples) and lists are expressible in the language, hence we add them. Thus, we have $\times, \langle\rangle$ from section 2 as additional type constructors.

Definition 4.1 A *type expression* is a tree with type variables or base types at the leaves, and type constructors, from $\times, \rightarrow, \langle\rangle, \forall X$ at the internal nodes. □

The kind of polymorphism that occurs in the 2nd order λ calculus is called *parametric* polymorphism (hence the name parametricity). For this kind, polymorphic functions must work *uniformly* at all types. For example, consider the function $+\!\!+$, which appends two lists. It can be applied to a pair of lists of *any* element type, and has **polymorphic** type $\forall X.\langle X\rangle \times \langle X\rangle \rightarrow \langle X\rangle$. Since $+\!\!+$ must work uniformly for lists of any element type, it *cannot use any information particular to a specific element type*, not even the equality predicate between elements. In this sense, elements are treated by it as uninterpreted black boxes. That $+\!\!+[\alpha]$ behaves like $+\!\!+[\beta]$ means that if their inputs are related by any mapping $H : \alpha \times \beta$ whatsoever, then their outputs must be similarly related: for any lists $u, v : \langle\alpha\rangle$, $u', v' : \langle\beta\rangle$, if $(\langle H\rangle \times \langle H\rangle)([u,v],[u',v'])$ then $\langle H\rangle(+\!\!+[\alpha]([u,v]) \ , +\!\!+[\beta]([u',v']))$. As seen below, this follows from the parametricity theorem. If we restrict H to range over mappings that are extensions of mappings on base types, this implies that $+\!\!+$ is *rel*-fully generic. But it provides more: H can be a mapping between *any* two types (not necessarily base), e.g. $H : char \times \langle int\rangle$ could be $\{(a,\langle 1\rangle),(b,\langle 7,1\rangle)\}$.

Before presenting the parametricity theorem, we need to generalize the correspondence between type expressions and mappings between types, introduced in Section 2. Recall that, to extend mappings on base types to related complex types, we replaced the type variables in the leaves of the tree

[5]Note: x is a value variable, X is a type variable.

representing the type expression by the given mappings, and then viewed each type constructor in the tree as a mapping constructor. We generalize, firstly, by allowing arbitrary mappings as substitutions, and relaxing the restriction from Section 2 that domains of the mappings are distinct. Thirdly, following our extended set of constructors, we need to define for $\rightarrow$ and $\forall$ corresponding mapping constructors. Finally, we now allow a type expression to have base types as leaves; these correspond to special constant mappings.

Firstly, one generalization has been noted – the mappings substituted for the variables in the leaves are not restricted to have base types as domains and codomains. Further, we allow that different type variables be substituted *independently*, even by different mappings with the same domain and codomain. For example, consider the function *zip* : $\forall X \forall Y. \langle X \rangle \times \langle Y \rangle \rightarrow \langle X \times Y \rangle$, that takes a pair of lists of equal length, and creates a list of pairs. Since it is polymorphic, we expect that if $H_X : \alpha_X \times \beta_X$ and $H_Y : \alpha_Y \times \beta_Y$ are mappings, and we have $\langle H_X \rangle(x_\alpha, x_\beta)$ and $\langle H_Y \rangle(y_\alpha, y_\beta)$, then $zip[\alpha_X][\alpha_Y]([x_\alpha, y_\alpha])$ and $zip[\beta_X][\beta_Y]([x_\beta, y_\beta])$ are related by $\langle H_X \times H_Y \rangle$. This holds also if all four types are the same type, say *int*, and H_X and H_Y are *different* mappings from integers to integers. Contrast these with the restrictions on mappings substituted in a tree of Section 2.

We now define the new mapping constructors.

Definition 4.2 Let $K : \alpha \times \beta$, $K' : \alpha' \times \beta'$, be given mappings. Their extension to a mapping between the function types $\alpha \rightarrow \alpha', \beta \rightarrow \beta'$, denoted $H \rightarrow H'$, is defined as follows:
$(K \rightarrow K')(f, f')$ iff whenever $K(x, x')$ then also $K'(f(x), f'(x'))$. □

That is, functions are related if whenever their inputs are related then so are their outputs. Note that for $K = K'$, and $f = f'$, this states that f is invariant under K, as defined in def. 2.9. We can now restate the previous property of $+\!\!+$ as: for any $H : \alpha \times \beta$, $(\langle H \rangle \times \langle H \rangle \rightarrow \langle H \rangle)(+\!\!+[\alpha], +\!\!+[\beta])$.

So far, each type constructor we considered was a function from types to types, and the corresponding mapping constructor was similarly a function from mappings to mappings. The universal quantifier is different. A type expression $T(X)$ with X free, is essentially a function from types to types. The $\forall X$ constructor thus takes a *function* on types to a type. Similarly, the corresponding mapping constructor takes a function on mappings to a mapping. To facilitate the definition, we introduce *mapping variables* $\mathcal{X}, \mathcal{X}_1 \ldots$. To move from types to mappings, replace each occurrence of X by $\mathcal{X}$. For example, $\langle X \rangle \times \langle X \rangle$ is a type expression, representing a unary type function; $\langle \mathcal{X} \rangle \times \langle \mathcal{X} \rangle$ is the corresponding mapping expression, representing a unary mapping function

Definition 4.3 Assume polymorphic type $\forall X.T(X)$. The corresponding mapping, denoted $\forall \mathcal{X}.\mathcal{T}(\mathcal{X})$, is defined as follows:
$\forall \mathcal{X}.\mathcal{T}(\mathcal{X})(f, f')$ holds iff for every $H : \{\alpha \times \beta\}$ we have that $\mathcal{T}(H)(f[\alpha], f'[\beta])$. □

That is, polymorphic functions are related if their α and β components are related by every mapping between α and β, extended to the appropriate type. Note that if X is the only free variable of $T(X)$, then $\forall \mathcal{X}.\mathcal{T}(\mathcal{X})$ denotes a fixed mapping, which can be viewed as an intersection of mappings. A similar view holds for $\forall X$ [12]. Now, further rewriting the property of the append function $+\!\!+$, we get $(\forall \mathcal{X}. \langle \mathcal{X} \rangle \times \langle \mathcal{X} \rangle \rightarrow \langle \mathcal{X} \rangle)(+\!\!+, +\!\!+)$. Note that now we have a direct statement about *append* itself, and the mapping is derived from its type only, without using any information about the function.

Finally, a base type leaf, b, corresponds to the identity mapping I_b on that type. The reason for this is illustrated by the function *count* : $\forall X. \langle X \rangle \rightarrow int$, that counts the number of elements in a list. Consider two instances of the function $count[\alpha] : \langle \alpha \rangle \rightarrow int$ and $count[\beta] : \langle \beta \rangle \rightarrow int$, where α and β are related by some mapping H. Let H' be some mapping over the integers. For $(\langle H \rangle \rightarrow H')(count[\alpha], count[\beta])$ to hold, the output of the functions should be related by H' whenever the input is related by $\langle H \rangle$. For example, assume given a list $\langle a, b \rangle$; the mapping $H = \{(a, e), (b, f)\}$ takes it to $\langle e, f \rangle$. Both lists have count 2, as one expects, so necessarily $H'(2, 2)$. The same holds for each cardinality, hence H' must be I_{int}; thus, *int* on the right of $\rightarrow$ in the type of *count* is a "constant" type, and should correspond to a constant mapping, namely I_{int}.

The parametricity theorem states a property of calculus expressions that depend *only* on their type.

Theorem 4.4 (Parametricity [14, 15]) *Let T be* *a closed type, and let $\mathcal{T}$ be the corresponding mapping. If l is expressible in the 2nd-order λ calculus, and $l : T$ then $\mathcal{T}(l, l)$.* □

Corollary 4.5 *List functions with polymorphic type are* rel*-fully generic.* □

Functions which use **equality** between elements do not work *uniformly*: the computation of $=$ itself is element-type dependent. For example, consider list difference, where $l - l'$ removes from l all copies of elements appearing in l'. Like $-$ for sets (proposition 3.4), it is not *rel*-fully generic; obviously, the parametricity theorem does not apply to it. Indeed, it is not polymorphic, as defined so far, and it does *not* have type $\forall X.\langle X\rangle \times \langle X\rangle \rightarrow \langle X\rangle$. It can be given the type $\forall X^{=}.\langle X^{=}\rangle \times \langle X^{=}\rangle \rightarrow \langle X^{=}\rangle$, where $X^{=}$ now ranges only over types with $=$. The parametricity theorem can be adapted to deal with such functions, by restricting attention to mappings that preserve $=$, i.e., are injective, using $\mathcal{X}^{=}$ to range over such mappings.

4.2 Parametricity for Sets

The parametricity theorem does not apply to queries over sets because sets cannot be represented in the λ calculus.[6] Our approach is to use the theorem for lists, and transfer the parametricity property from lists to 'similar' sets. We use the *rel* extension mode only. To facilitate the transfer, we use a simple (set-theoretic) typed semantic domain, as follows. Domains for base types, and complex values types are as in Section 2. Domains of higher-order function types are defined in a straightforward manner: the domain for $\alpha \rightarrow \beta$ includes *all* functions from the domain of α to that of β. This construction gives an interpretation to all monomorphic types. Adding collections of functions, collections of collections, and so on, indexed by monomorphic types, gives us an interpretation for polymorphic types, where in $\forall X$ the variable X ranges over monomorphic types only. A further restriction is that all universal quantifiers appear on the outside of a type. This is definitely adequate in the database context. The semantics of a polymorphic function f is a collection of α-components $f[\alpha]$, one for each monomorphic α. We do not introduce a language for sets, but rather consider (pure set) values and their types from the domain above.

[6]The work in [11] studies *list* queries expressible in the λ calculus.

We denote a pure list type expression by T^{list}; if every occurrence of $\langle\rangle$ is replaced by $\{\}$, we obtain a pure set type expression, denoted T^{set}. We call such types *related*. The mappings, $\mathcal{T}^{list}$ and $\mathcal{T}^{set}$ are also called related. $\mathcal{T}^{set}$ is obtained from T^{set} as in the previous section, using the set mapping constructor with the *rel* extension mode.

It is simpler to present the claim and proof for the case that in $\forall\mathcal{X}.\mathcal{T}^{set}(\mathcal{X})$, $\mathcal{X}$ ranges over all mappings between base types only. As this defines a mapping different from $\forall\mathcal{X}.\mathcal{T}^{set}(\mathcal{X})$, we denote it by $\forall\mathcal{X}.\tilde{\mathcal{T}}^{set}(\mathcal{X})$. A similar restriction applies to lists, and we denote the mapping by $\forall\mathcal{X}.\tilde{\mathcal{T}}^{list}(\mathcal{X})$. Later we generalize to all mappings between monomorphic set types.

Let *toset* be the function which converts a list to a set with the same elements. The following lemma relates this function and the *rel* extension mode.

Lemma 4.6 *Let $H : \alpha \times \beta$, lists $l : \langle\alpha\rangle, l' : \langle\beta\rangle$ and sets $s : \{\alpha\}, s' : \{\beta\}$ be given.*

1. *If $\langle H\rangle(l, l')$ and $toset(l) = s$, $toset(l') = s'$ then $\{H\}^{rel}(s, s')$.*

2. *If $\{H\}^{rel}(s, s')$ then there exists l, l' such that $toset(l) = s$, $toset(l') = s'$, and $\langle H\rangle(l, l')$.* □

The following extends *toset* to higher-order list and set values.

Definition 4.7 List and set values, l, s, of related types T^{list}, T^{set}, are **analogous**, denoted $l \xrightarrow{\text{l to s}} s$,[7] if the following holds. The definition is by induction on the structures of their types:
Base type: $l = s$
Product: the analogy holds component-wise
Lists and sets ($T^{list} = \langle\bar{T}^{list}\rangle$, $T^{set} = \{\bar{T}^{set}\}$): if each element l_i of l can be replaced by an analogous set value, giving a list of sets l', such that $toset(l') = s$
Functions($l = f_l, s = f_s$): if whenever $x_l \xrightarrow{\text{l to s}} x_s$ then also $f_l(x_l) \xrightarrow{\text{l to s}} f_s(x_s)$
$\forall$ *types:* for all base types α, $l[\alpha] \xrightarrow{\text{l to s}} s[\alpha]$. □

[7]Formally, the type has to be included in the notation; for simplicity, it is omitted.

For example, $+\!\!+ \xrightarrow{\text{l to s}} \cup$, because for flat lists l, l', $toset(l+\!\!+l') = toset(l) \cup toset(l')$.

For complex value types (no $\rightarrow, \forall$), l is analogous to s if they are related by *toset* extended to all nesting levels, so $\xrightarrow{\text{l to s}}$ is a total, surjective function from lists to sets. However, there are list *functions* having no analogous set function (e.g. the function *head*) so in general the relationship is partial.

We would like to use this correspondence to pull parametricity from lists to sets. That is, if $\tilde{T}^{list}(l,l)$ and $l \xrightarrow{\text{l to s}} s$, then $\tilde{T}^{set}(s,s)$. Unfortunately, we can not show this in general; our technique breaks down for types with complex combinations of bulk and function constructors. The restricted set of types that we do cover is sufficient for all set queries of current interest.

One technical problem we face is that we need to show that related sets have analogous related lists. We can prove this in a restricted case:

Definition 4.8 A list type expression is an **s-to-l** type if it contains no universal quantifiers, and also no $\langle\rangle$ appears in it under $\rightarrow$. □

Lemma 4.9 *Let $T^{list}(X_1,\ldots,X_n)$ be s-to-l type. Given mappings on base types $H_i : \alpha_i \times \beta_i$, if $\tilde{T}^{set}(H_1,\ldots,H_n)(s_1,s_2)$ then there exists l_1,l_2 such that $l_i \xrightarrow{\text{l to s}} s_i, i = 1,2$, and $\tilde{T}^{list}(H_1,\ldots,H_n)(l_1,l_2)$.* □

Proof:(sketch) For bulk types $\langle\rangle$ and $\{\}$, we use the second part of lemma 4.6. The difficult case is the function constructor. However, in an *s-to-l* function type T^{list}, there are no list constructors. Therefore the related set type T^{set} is identical. The same goes for mappings $\tilde{T}^{list}$ and $\tilde{T}^{set}$. Moreover, functions of these types are analogous iff they are identical. Therefore, given s_i, we choose $l_i = s_i$. □ For related set functions of general types, we do not know how to find analogous related list functions.

Definition 4.10 A list type expression is an **l-to-s** type if for each $T_1 \rightarrow T_2$ it contains, T_1 is an s-to-l type. No universal quantifiers are allowed. □

Lemma 4.11 *Let $T^{list}(X_1,\ldots,X_n)$ be an l-to-s type, and $H_i : \alpha_i \times \beta_i$ mappings on base types. If $\tilde{T}^{list}(H_1,\ldots,H_n)(l_1,l_2)$ and $l_i \xrightarrow{\text{l to s}} s_i$ then $\tilde{T}^{set}(H_1,\ldots,H_n)(s_1,s_2)$.*

Proof:(sketch) For bulk types, we use the first part of lemma 4.6. The difficult case is for function types: say we are given $(\tilde{T}_1^{list} \rightarrow \tilde{T}_2^{list}(f_1^l, f_2^l)$ for list functions f_1^l, f_2^l. That is,

$$\tilde{T}_1^{list}(l_1,l_2) \Rightarrow \tilde{T}_2^{list}(f_1^l(l_1), f_2^l(l_2)). \qquad (*)$$

Also, $f_i^l \xrightarrow{\text{l to s}} f_i^s$, that is,

$$l_i \xrightarrow{\text{l to s}} s_i \Rightarrow f_i^l(l_i) \xrightarrow{\text{l to s}} f_i^s(s_i). \qquad (**)$$

We need to show $(\tilde{T}_1^{set} \rightarrow \tilde{T}_2^{set})(f_1^s, f_2^s)$ that is,

$$\tilde{T}_1^{set}(s_1,s_2) \Rightarrow \tilde{T}_2^{set}(f_1^s(s_1), f_2^s(s_2)). \qquad (***)$$

Given $\tilde{T}_1^{set}(s_1,s_2)$, since T_1^{list} is an s-to-l type, by Lemma 4.9, there exist lists l_1,l_2 such that $l_i \xrightarrow{\text{l to s}} s_i$ and $\tilde{T}_1^{list}(l_1,l_2)$. Therefore, by (**) we have $f_i^l(l_i) \xrightarrow{\text{l to s}} f_i^s(s_i)$, and by (*) we have $\tilde{T}_2^{list}(f_1^l(l_1), f_2^l(l_2))$. Therefore, by induction, we have $\tilde{T}_2^{set}(f_1^s(s_1), f_2^s(s_2))$ as required by (***). □

Definition 4.12 A list type (without free variables) is **LtoS** if it is of the form $\forall\vec{X}.T^{list}$, where T^{list} is l-to-s. □

We can now state the main theorem that relates list values to set values.

Theorem 4.13 *Let T^{list} be an LtoS type. If $\tilde{T}^{list}(l_1,l_2)$ and $l_i \xrightarrow{\text{l to s}} s_i$ then $\tilde{T}^{set}(s_1,s_2)$.* □

The proof is component-wise on the α-components of polymorphic values l_i,s_i. The heart of the proof is in the lemma for l-to-s types.

So far we have considered only mappings whose domains and codomains are base types, whereas in the definitions of $\mathcal{T}^{list}, \mathcal{T}^{set}$ mappings on all monomorphic types were included. We do need this added generality. For example, if we have $(\forall\mathcal{X}.\{\mathcal{X}\} \times \{\mathcal{X}\} \rightarrow \{\mathcal{X}\})(\cup,\cup)$, if $\mathcal{X}$ ranges only over mappings between base types, this says nothing about the behavior of $\cup$ on nested sets. We note that in the 2nd-order λ calculus we can choose base types arbitrarily. In particular, we can embed the domains of monomorphic set types as base types in the list universe. With this, the theorem can be stated for $\mathcal{T}^{set}$ rather than $\tilde{T}^{set}$. Details are given in the full paper.

Example 4.14 The selection operation σ on lists has LtoS type $\forall X.(X \rightarrow \texttt{bool}) \rightarrow \langle X\rangle \rightarrow \langle X\rangle$, because $X \rightarrow \texttt{bool}$ is s-to-l. But the type $\forall X.(\langle X\rangle \rightarrow \texttt{bool}) \rightarrow \langle X\rangle \rightarrow \langle X\rangle$ is not LtoS, since $\langle X\rangle \rightarrow \texttt{bool}$ is not s-to-l. Also, *fold* : $\forall X.\forall Y.(X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow \langle X\rangle \rightarrow Y$ is LtoS, whereas *ext* : $\forall X.\forall Y.(X \rightarrow \langle Y\rangle) \rightarrow \langle X\rangle \rightarrow \langle Y\rangle$ is not. □

For set value s, let $s : T^{set}$ denote that s has an analogous list value with type T^{list}. As a corollary to the parametricity theorem and theorem 4.13, we have:

Corollary 4.15 *Let T^{list} be an LtoS type. If $s : T^{set}$ then $\mathcal{T}^{set}(s, s)$.* □

Now we are able to prove parametricity properties for set queries. To do this for a set query q_s, one needs only to find an analogous list query q_l, and to know its type T. For example, we know that $+\!\!+ \xrightarrow{1 \text{ to } s} \cup$, and that $+\!\!+ : \forall X.\langle X\rangle \times \langle X\rangle \to \langle X\rangle$. Therefore, we can deduce that $(\forall \mathcal{X}.\{\mathcal{X}\} \times \{\mathcal{X}\} \to \{\mathcal{X}\})(\cup, \cup)$. which implies that $\cup$ is *rel*-fully generic, and more, as seen below. The type could be found using type inference, or could be verified using type checking. We note that there are **type inference** algorithms for the ML subset of the 2nd-order λ calculus.

4.3 Genericity vs. Parametricity

Genericity is a concept: a query being invariant under a class of mappings. Its definition gives no tool for deriving occurrences of the property. Parametricity is also a concept of invariance: We call a query Q *parametric w.r.t. a type* T if $\mathcal{T}(q, q)$. But the parametricity theorem is a powerful tool for deriving invariances from query types. Here we compare and contrast the two notions. We also consider which can provide us better information about the invariance properties of a query.

The following are important differences between the two notions.

1. Parametricity deals only with typed queries (i.e., expressible in the 2nd-order λ calculus), whereas genericity is unrestricted.
2. Genericity considers only extensions of mappings between *base* types, but parametricity includes invariance under *all* mappings.
3. Genericity considers extensions of mappings on base types to all *complex value* types, but there it stops. Invariance is defined as a separate notion. In parametricity, given mappings are extended to all *types*, including that of the query. This, in contrast to genericity, allows to consider *different* mappings with the same domain, either because they are obtained from different type variables, or because one is obtained from a type, the other is a constant.

These points are elaborated and illustrated below.

Regarding item 2, an extension of mappings on base types can only relate values of the same structure, e.g., the same set nesting. For a given H, $H^{rel}(\{2\}, \{3\})$ and $H^{rel}(\{\{2\}\}, \{\{3\}\})$ are possible, but $H^{rel}(\{\{2\}\}, \{3\})$ is not. Hence genericity provides only invariance of restricted (i.e. structure-preserving) classes of mappings. In contrast, for a polymorphic query of type $\forall X.T(X)$, X ranges over all mappings, including mappings between non base types, having different structures. Therefore, for such a query, and such are most queries of interest, parametricity considers invariance under a *larger* class of mappings, and so gives a tighter fit. (This will also prove important for optimization.) Of course, the idea implies that certain queries are generic but not parametric (for any "reasonable" type). Parametric queries, in contrast to generic queries, must be invariant under mappings which don't preserve structure, hence cannot use any information about structure. Consider the query *nest parity* (*np*) which accepts a nested set and returns *true* if the depth of nesting is even, and *false* otherwise.

Proposition 4.16 np *is fully generic but is not parametric for any type of the form $\forall X.\{^n X\}^n \to$* `bool` *(where $\{^n$ stands for n nested set brackets).*□

This result is also related to point 1. It illustrates the advantage of having different concepts; each provides more information on some queries. It also illustrates the use of parametricity for inexpressibility results.

Regarding point 1, union can be given an untyped interpretation, so $\{1\} \cup \{\{2\}\}$ is legal; it is fully generic. Parametricity, however, applies only to the typed version, $\cup : \forall X.\{X\} \times \{X\} \to \{X\}$.

Regarding point 3, consider again the query *count*, which now counts the number of elements in a *set*. We have $count(\{2\}) = 1$; if *count* is invariant under any $H : int \times int$, then necessarily $H(1, 1)$. The same applies to each cardinality, hence *count* is rel-generic *only* w.r.t. the identity mapping, which provides no information at all — every function is generic w.r.t. this mapping. However, by parametricity we have $(\forall \mathcal{X}^{=}.\{\mathcal{X}^{=}\} \to int)(count, count)$, i.e. *count* is invariant under all mappings which are injective on its argument and the identity on its result. Note that in Section

2 we consider invariance under mappings that preserve constants and functions. Here essentially we generalize that to preservation of a domain.

The point is that, in genericity, if we are restricted to the identity mapping for the query's output type, the same applies to all its occurrences in the type. Parametricity allows to use different mappings in different places in the query. The information as to what extent this is possible is neatly captured by its type.

We have noted that queries with polymorphic types are rel fully-generic, since they are invariant under mappings between monomorphic types, hence are certainly invariant under extensions of mappings on base types. Typed union is an example. Similarly, queries with types quantified by $\forall X^{=}$ are rel-generic w.r.t all injective mappings. Set difference, $- : \forall X^{=}.\{X^{=}\} \times \{X^{=}\} \rightarrow \{X^{=}\}$, is an example. In both cases, parametricity provides invariance under more mappings than genericity. But, we have defined genericity w.r.t. constants and functions. Can we derive such invariances from parametricity? The answer is positive, provided that the use of the constant/function in the query is *well-behaved* in the sense that it can be captured by type information. This is essentially *orthogonality* in the use of constants/functions in a query language.

Let ins_c be the query that inserts constant c into a set, $ins_c(R) = R \cup \{c\}$. Assuming c can be *any* constant, ins is a higher-order function, $ins : \forall X.X \rightarrow \{X\} \rightarrow \{X\}$. By corollary 4.15, if $H(c, c')$ then if $\{H\}^{rel}(R, R')$ then $\{H\}^{rel}(ins_c(R), ins_{c'}(R'))$. Similarly, the general form for the relational select accepts the predicate as a parameter. $\sigma_p(S)$ returns the elements of S which satisfy p, with type $\sigma : \forall X.(X \rightarrow bool) \rightarrow \{X\} \rightarrow \{X\}$. We have : if $(H \rightarrow bool)(p, p')$ then if $\{H\}^{rel}(R, R')$ then $\{H\}^{rel}(\sigma_p(R), \sigma_{p'}(R'))$. For $p = p'$, we have σ_p is generic w.r.t. all mappings that preserve p, as a function. When p is $=_c$, that returns *true* iff its argument is c, $\sigma_{=c}$ is generic w.r.t. all mappings that *strictly* preserve c. Comparing the two examples, the intuition is that if we need to test equality with a constant, we need strict preservation, otherwise preservation is sufficient.

Of course, if a language allows only $\sigma_{=5}$, but not σ then the type will be $\{int\} \rightarrow \{int\}$, because this query can only be applied to sets of integers. Therefore, we only have invariance under the identity mapping, and we cannot derive that the query is generic w.r.t. mappings that strictly preserve 5. The same holds if given $\sigma_{=5}$ we infer the restricted type, and do not view it as an instance of the more general σ with a more general type. The more general the type we have for a query, the more information that can be gained.

4.4 Optimization

Invariance of a query under mappings means it *commutes* with them. We now use genericity and parametricity to derive generalizations of some well known algebraic equivalences, and also some new ones.

Given a functional mapping f, $\{f\}^{rel}$ is the function $map(f)$, which applies f to each element of a set, $\{\{f\}\}^{rel}$ is $map(map(f))$, and so on. Thus, rel-generic queries commute with $map(f)$, for f in an appropriate class. For fully generic queries, e.g., $\cup$, f is not restricted at all. Thus, for all f

$$\text{if } (map(f))(R) = R' \text{ and } (map(f))(S) = S' \\ \text{then } (map(f))(R \cup S) = R' \cup S'$$

This is equivalent to

$$(map(f))(R \cup S) = (map(f))(R) \cup (map(f))(S)$$

In particular, f could be any user-defined method, in any programming language, about which we know nothing. But, if a query is invariant only under the mappings that preserve constant c, then we require $f(c) = c$.

Within the genericity framework, values related by $\{f\}^{rel}$ must have exactly the same structure, whereas for parametricity this need not be the case. For example, consider $f = \pi_1$, where $\pi_1([a, b]) = a$. In this case, $map(\pi_1)$ is the relational projection, $\Pi_{\$1}$. It follows that we can push projection through fully parametric queries : e.g. for $\cup$:

$$\Pi_{\$1}(R \cup S) = \Pi_{\$1}(R) \cup \Pi_{\$1}(S)$$

a well known equivalence. Note that the full genericity of $\cup$ does not imply the above.

π_1, above, is not injective since different tuples may have the same first component. Therefore, in general, we cannot push projection through queries which are invariant only under injective mappings, such as set difference. However, in special cases π_1 might be injective. For example let R and S be relations of employees and students, where their first columns are a common key (i.e. a key for $R \cup S$) such as a social security number. Then no two tuples can have the same first component, and π_1 is injective on $R \cup S$. Then:

$$\Pi_{\$1}(R - S) = \Pi_{\$1}(R) - \Pi_{\$1}(S).$$

5 Discussion

We have generalized the classical notion of genericity from one (almost) abstract domain to many domains, and defined it so no predicate, even equality, need be preserved, and have accounted for preservation of any function or predicate. Equality still differs from other predicates in that it is used *implicitly* in many operations. We showed that leads to a rich structure of genericity notions, and classified many languages and operations w.r.t. their genericity properties.

More importantly, we have shown genericity and parametricity to be closely related, although incomparable. For queries with a general polymorphic type the latter, deriving from the type, can provide more information, such as invariance under mappings on non-base types, and preservation of interpreted domains. We have also shown that many algebraic laws can be derived from parametricity. It follows that, hopefully, type checking and type inference algorithms can be used to verify or discover such properties automatically. This opens new possibilities for optimizing queries defined in general declarative languages, and over user-defined data types. This is a promising direction for future research.

We point out that the L-to-S types are rich enough to capture the entire nested relational algebra. The core constructs of the monadic algebra of [5] can be expressed using only regular universal quantification and are thus fully generic. The addition of = to their algebra gives the full nested relational algebra, and now we have rel-generic queries w.r.t. injective mappings. Indeed their naturality theorem states that their language is parametric. Our results provide more.

Issues for future research include: Extending the results to other data types and type systems, removing some of the restrictions on types in the proofs, and exploring the applications to query optimization.

Acknowledgements

We would like to thank the anonymous referees for their helpful and detailed comments.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. *VLDB Journal*, 1996.

[2] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *POPL*, pages 110 - 120, 1979.

[3] A. Aylamazyan, M. Gilula, A. Stolbushkin, and G. Schwartz. Reduction of the relational model with infinite domains to the case of finite domains (russian). *Proc. USSR Acad. Sci. (Doklady)*, 2(286):308–311, 1986.

[4] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *DBPL*, 1991.

[5] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *ICDT* 1992. Springer-Verlag. LNCS 646.

[6] A. Chandra. Programming primitives for database languages. In *POPL*, 1981.

[7] A. Chandra and D. Harel. Computable queries for relational data bases. *JCSS*, 21(2):156 - 178, 1980.

[8] J.-Y. Girard. interprétation fonctionelle et élimination des coupures de l'arithmétique d'order supérieur. Thèse D'Etat, Université de Paris VII, Paris 1972.

[9] T. Hirst and D. Harel. Completeness results for recursive databases. In *PODS*, 1993.

[10] R. Hull and C. K. Yap. The format model : a theory of database organization. *JACM*, 31(3):518 - 537, 1984.

[11] G. Hillebrand and P. Kannelakis. Functional query languages as typed lambda calculi of fixed order. In *PODS*, pages 222–231, 1993.

[12] J. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, chapter 8. The MIT press/Elsevier, 1990.

[13] J. Paredaens, J. V. den Bussche, and D. V. Gucht. Towards a theory of spatial database queries. In *PODS*, pages 279 - 288, 1994.

[14] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*. Springer-Verlag, 1974. LNCS 19.

[15] P. Wadler. Theorems for free! In *FPCA*, pages 347 - 359, 1989.