PART

E Expressiveness and Complexity

Various query languages were presented in Parts B and D. Simple languages like conjunctive queries were successively augmented with various constructs such as union, negation, and recursion. The primary motivation for defining increasingly powerful languages was the need to express useful queries not captured by the simpler languages. In the presentation, the process was primarily example driven. The following chapters present a more advanced and global perspective on query languages. In addition to their ability to express specific queries, we consider more broadly the capability of languages to express queries of a given complexity. This leads to establishing formal connections between languages and complexity classes of queries. This approach lies on the border between databases, complexity theory, and logic. It is related to characterizations of complexity classes in terms of various logics.

The basic framework for the formal development is presented in Chapter 16, in which we discuss the notion of a query and produce a formal definition. It turns out that it is relatively easy to define languages expressing *all* queries. Such languages are called *complete*. However, the real challenge for the language designer is not simply to define increasingly powerful languages. Instead an important aspect of language design is to achieve a good balance between expressiveness and the complexity of evaluating queries. The ideal language would allow expression of most useful queries while guaranteeing that *all* queries expressible in the language can be evaluated with reasonable complexity. To formalize this, we raise the following basic question: How does one evaluate a query language with respect to expressiveness and complexity? In an attempt to answer this question, we discuss the issue of sizing up languages in Chapter 16.

Chapter 17 considers some of the classes of queries discussed in Part B from the viewpoint of expressiveness and complexity. The focus is on the relational calculus of Chapter 5 and on its extensions *fixpoint* and *while* defined in Chapter 14. We show the connection of these languages to complexity classes. Several techniques for showing the nonexpressibility of queries are also presented, including *games* and 0-1 laws.

Chapter 17 also explores the intriguing theoretical implications of one of the basic assumptions of the pure relational model—namely, that the underlying domain **dom** consists of uninterpreted, unordered elements. This assumption can be viewed as a metaphor for the data independence principle, because it implies using only logical properties of data as

opposed to the underlying implementation (which would provide additional information, such as an order).

Chapter 18 presents highly expressive (and complex) languages, all the way up to complete languages. In particular, we discuss constructs for value invention, which are similar to the object creation mechanisms encountered in object languages (see Chapter 21).

For easy reference, the expressiveness and complexity of relational query languages are summarized at the end of Chapter 18.

16 Sizing Up Languages

Alice: *Do you ever worry about how hard it is to answer queries?*
Riccardo: *Sure—my laptop can only do conjunctive queries.*
Sergio: *I can do the* while *queries on my Sun.*
Vittorio: *I don't worry about it—I have a Cray in my office.*

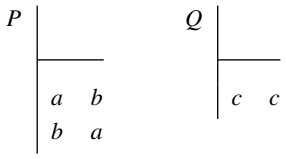
This chapter lays the groundwork for the study of the complexity and expressiveness of query languages. First the notion of query is carefully reconsidered and formally defined. Then, the complexity of individual queries is considered. Finally definitions that allow comparison of query languages and complexity classes are developed.

16.1 Queries

The goal of Part E is to develop a general understanding of query languages and their capabilities. The first step is to formulate a precise definition of what constitutes a query. The focus is on a fairly high level of abstraction and thus on the mappings expressible by queries rather than on the syntax used to specify them. Thus, unlike Part B, in this part we use the term *query* primarily to refer to mappings from instances to instances rather than to syntactic objects. Although there are several correct definitions for the set of permissible queries, the one presented here is based on three fundamental assumptions: *well-typedness*, *computability*, and *genericity*.

The first assumption involves the schemas of the input and the answer to a query. A query is over a particular database schema, say $\mathbf{R}$. It takes as input an instance over $\mathbf{R}$ and returns as answer a relation over some schema S. In principle, it is conceivable that the schema of the result may be data dependent. However, to simplify, it is assumed here (as in most query languages) that the schema of the result is fixed for a given query. This assumption is referred to as *well-typedness*. Thus, for us, a query is a partial mapping from $inst(\mathbf{R})$ to $inst(S)$ for *fixed* $\mathbf{R}$ and S. By allowing partially defined mappings, we account for queries expressed by programs that may not always terminate.

Because we are only interested in effective queries, we also make the natural assumption that query mappings are *computable*. Query computability is defined using classical models of computation, such as Turing machines (TM). The basic idea is that the query must be "implementable" by a TM. Thus there must exist a TM that, given as input a natural encoding of a database instance on the tape, produces an encoding of the output. The formalization of these notions requires some care and is done next.



(a)

P[0#1][1#0]Q[10#10]

(b)

Figure 16.1: An instance **I** and its TM encoding with respect to $\alpha = abc$

The first question in developing the formalization is, How can input and output instances be represented on a TM tape that has finite alphabet when the underlying domain **dom** is infinite? We resolve this by using standard encodings for **dom**. As we shall see later on, although this permits us to use conventional complexity theory in our study of query language expressiveness, it also takes us a bit outside of the pure relational model.

We focus on encodings of both **dom** and of subsets of **dom**, and we use the symbols 0 and 1. Let $\mathbf{d} \subseteq \mathbf{dom}$ and let $\alpha = \{d_0, d_1, \ldots, d_i, \ldots\}$ be an enumeration of **d**. The *encoding* of **d** relative to α is the function enc_α, which maps d_i to the binary representation of i (with no leading zeros) for each $d_i \in \mathbf{d}$. Note that $|enc_\alpha(d_i)| \leq \lceil \log i \rceil$ for each i.

We can now describe the encoding of instances. Suppose that a set $\mathbf{d} \subseteq \mathbf{dom}$, enumeration α for **d**, source schema $\mathbf{R} = \{R_1, \ldots, R_m\}$, and target schema S are given. The encoding of instances of **R** uses the alphabet $\{0, 1, [,], \#\} \cup \mathbf{R} \cup \{S\}$. An instance **I** over **R** with $adom(\mathbf{I}) \subseteq \mathbf{d}$ is encoded relative to α as follows:

1. $enc_\alpha(\langle a_1, \ldots, a_k \rangle)$ is $[enc_\alpha(a_1)\# \ldots \#enc_\alpha(a_k)]$.
2. $enc_\alpha(\mathbf{I}(R))$, for $R \in \mathbf{R}$, is $R\ enc_\alpha(t_1) \ldots enc_\alpha(t_l)$, where $t_1, \ldots, t_l$ are the tuples in $\mathbf{I}(R)$ in the lexicographic order induced by the enumeration α.
3. $enc_\alpha(\mathbf{I}) = enc_\alpha(\mathbf{I}(R_1)) \ldots enc_\alpha(\mathbf{I}(R_m))$.

Example 16.1.1 Let $\mathbf{R} = \{P, Q\}$, **I** be the instance over **R** in Fig. 16.1(a), and let $\alpha = abc$. Then $enc_\alpha(\mathbf{I})$ is shown in Fig. 16.1(b).

Let α be a fixed enumeration of **dom**. In this case the encoding enc_α described earlier is one-to-one on instances and thus has an inverse enc_α^{-1} when considered as a mapping on instances. We are now ready to formalize the notion of computability *relative to an encoding* of **dom**.

Definition 16.1.2 Let α be an enumeration of **dom**. A mapping q from $inst(\mathbf{R})$ to $inst(S)$ is *computable* relative to α if there exists a TM M such that for each instance **I** over **R**

(a) if $q(\mathbf{I})$ is undefined, then M does not terminate on input $enc_\alpha(\mathbf{I})$, and

(b) if $q(\mathbf{I})$ is defined, M halts on input $enc_\alpha(\mathbf{I})$ with output $enc_\alpha(q(\mathbf{I}))$ on the tape.

As will be seen shortly, the third assumption about queries (namely, genericity) will permit us to reformulate the preceding definition to be independent of the encoding of **dom** used. Before introducing that notion, we consider more carefully the representation of database instances on TM tapes. In some sense, TM encodings on the tape are similar to the internal representation of the database on some physical storage. In both cases, the representation contains more information than the database itself. In the case of the TM representation, the extra information consists primarily of the enumeration α of constants necessary to define enc_α. In the pure relational model, this kind of information is not part of the database. Instead, the database is an abstraction of its internal (or TM) representation. This additional information can be viewed as noise associated with the internal representation and thus should not have any visible impact for the user at the conceptual level. This is captured by the *data independence* principle in databases, which postulates that a database provides an abstract interface that hides the internal representation of data.

We can now state the intuition behind the third and last requirement of queries, which formalizes the data independence principle. Although computations performed on the internal representation may take advantage of all information provided at this level, it is explicitly prohibited, in the definition of a query, that the result depend on such information. (In some cases this restriction may be relaxed; see Exercise 16.4.)

For example, consider a database that consists of a binary relation specifying the edges of a directed graph. Consider a query that returns as answer a subset of the vertexes in the graph. One can imagine queries that extract (1) all vertexes with positive in-degree, or (2) all vertexes belonging to some cycle, or (3) the first vertex of the graph as presented in the TM tape representation. Speaking intuitively, (1) and (2) are independent of the internal representation used, whereas (3) depends on it. Queries such as (3) will be excluded from the class of queries.

The property that a query depends only on information provided by the input instance is called genericity and is formalized next. The idea is that the constants in the database have no properties other than the relationships with each other specified by the database. (In particular, their internal representation is irrelevant.) Thus the database is essentially unchanged if all constants are consistently renamed. Of course, a query can always explicitly name a finite set of constants, which can then be treated differently from other constants. (The set of such constants is the set C in Definition 16.1.3.)

A *permutation* of **dom** is a one-to-one, onto mapping from **dom** to **dom**. As done before, each mapping ρ over **dom** is extended to tuples and database instances in the obvious way.

Definition 16.1.3 Let $\mathbf{R}$ and S be database schemas, and let C be a finite set of constants. A mapping q from $inst(\mathbf{R})$ to $inst(S)$ is *C-generic* iff for each $\mathbf{I}$ over $\mathbf{R}$ and each permutation ρ of **dom** that is the identity on C, $\rho(q(\mathbf{I})) = q(\rho(\mathbf{I}))$. When C is empty, we simply say that the query is *generic*.

The previous definition is best visualized using the following commuting diagram:

$$\begin{array}{ccc} \mathbf{I} & \stackrel{q}{\longrightarrow} & q(\mathbf{I}) \\ \downarrow\rho & & \downarrow\rho \\ \rho(\mathbf{I}) & \stackrel{q}{\longrightarrow} & \rho(q(\mathbf{I})) = q(\rho(\mathbf{I})). \end{array}$$

In other words, a query is C-generic if it commutes with permutations (that leave C fixed).

Genericity states that the query is insensitive to renaming of the constants in the database (using the permutation ρ). It uses only the relationships among constants provided by the database and is independent of any other information about the constants. The set C specifies the exceptional constants named explicitly in the query. These cannot be renamed without changing the effect of the query.

Permutations ρ for which $\rho(\mathbf{I}) = \mathbf{I}$ are of special interest. Such ρ are called *automorphisms* for $\mathbf{I}$. If ρ is an automorphism for $\mathbf{I}$ and $\rho(a) = b$, this says intuitively that a and b cannot be distinguished using the structure of $\mathbf{I}$. Let q be a generic query, $\mathbf{I}$ an instance, and ρ an automorphism for $\mathbf{I}$. Then, by genericity,

$$\rho(q(\mathbf{I})) = q(\rho(\mathbf{I})) = q(\mathbf{I}),$$

so ρ is also an automorphism for $q(\mathbf{I})$. In particular, a generic query cannot distinguish between constants that are undistinguishable in the input (see Exercise 16.5). Of course, this is not the case if the query explicitly names some constants.

We illustrate these various aspects of genericity in an example.

Example 16.1.4 Consider a database over a binary relation G holding the edges of a directed graph. Let $\mathbf{I}$ be the instance $\{\langle a, b\rangle, \langle b, a\rangle, \langle a, c\rangle, \langle b, c\rangle\}$.

Let σ be the CALC query

$$\{\langle x\rangle \mid \exists y G(x, y)\}.$$

Note that $\sigma(\mathbf{I}) = \{\langle a\rangle, \langle b\rangle\}$. Let ρ be the permutation defined by $\rho(a) = b$, $\rho(b) = c$, and $\rho(c) = d$. Then $\rho(\mathbf{I}) = \{\langle b, c\rangle, \langle c, b\rangle, \langle b, d\rangle, \langle c, d\rangle\}$. Genericity requires that $\sigma(\rho(\mathbf{I})) = \{\langle b\rangle, \langle c\rangle\}$. This is true in this case.

Note also that a and b are undistinguishable in $\mathbf{I}$. Formally, the renaming ρ defined by $\rho(a) = b$, $\rho(b) = a$, and $\rho(c) = c$ has the property that $\rho(\mathbf{I}) = \mathbf{I}$ and thus is an automorphism of $\mathbf{I}$. Let q be a generic query on G. By genericity of q, either a and b both belong to $q(\mathbf{I})$, or neither does. Thus a generic query cannot distinguish between a and b. Of course, this is not true for C-generic queries (for C nonempty). For instance, let $q_b = \pi_1(\sigma_{2=b}(G))$. Now q_b is $\{b\}$-generic, and $q_b(\mathbf{I}) = \{\langle a\rangle\}$. Thus q_b distinguishes between a and b.

It is easily verified that if a database mapping q is C-generic, then for each input instance $\mathbf{I}$, $adom(q(\mathbf{I})) \subseteq C \cup adom(\mathbf{I})$ (see Exercise 16.1).

In most cases we will ignore the issue of constants in queries because it is not central. Note that a C-generic query can be viewed as a generic query by including the constants in C in the input, using one relation for each constant. For instance, the $\{b\}$-generic query q_b over G in Example 16.1.4 is reduced to a generic query q' over $\{G, R_b\}$, where $R_b = \{\langle b \rangle\}$, defined as follows:

$$q' = \pi_1(\sigma_{2=3}(G \times R_b)).$$

In the following, we will usually assume that queries have no constants unless explicitly stated.

Suppose now that α and β are two enumerations of **dom** and that a generic mapping q from $\mathbf{R}$ to S is computed by a TM M using enc_α. It is easily verified that the same query is computed by M if enc_β is used in place of enc_α (see Exercise 16.2). This permits us to adopt the following notion of computable, which is equivalent to "computable relative to enumeration α" in the case of generic queries. This definition has the advantage of relying on finite rather than infinite enumerations.

Definition 16.1.5 A generic mapping q from $inst(\mathbf{R})$ to $inst(S)$ is *computable* if there exists a TM M such that for each instance $\mathbf{I}$ over $\mathbf{R}$ and each enumeration α of $adom(\mathbf{I})$,

(a) if $q(\mathbf{I})$ is undefined, then M does not terminate on input $enc_\alpha(\mathbf{I})$, and

(b) if $q(\mathbf{I})$ is defined, M halts on input $enc_\alpha(\mathbf{I})$ with output $enc_\alpha(q(\mathbf{I}))$ on the tape.

We are now ready to define queries formally.

Definition 16.1.6 Let $\mathbf{R}$ be a database schema and S a relation schema. A *query* from $\mathbf{R}$ to S is a partial mapping from $inst(\mathbf{R})$ to $inst(S)$ that is generic and computable.

Note that all queries discussed in previous chapters satisfy the preceding definition (modulo constants in queries).

Queries and Query Languages

We are usually interested in queries specified by the expressions (i.e., syntactic queries or programs) of a given query language. Given an expression E in query language L, the mapping between instances that E describes is called the *effect* of E. Depending on the language, there may be several alternative semantics (e.g., inflationary versus noninflationary) for defining the query expressed by an expression. A related issue concerns the specification of the output schema of an expression. In calculus-based languages, the output schema is unambiguously specified by the form of the expression. The situation is more ambiguous for other languages, such as datalog and *while*. Programs in these languages typically manipulate several relations and may not specify explicitly which is to be taken as the answer to the query. In such cases, the concepts of *input*, *output*, and *temporary relations* may become important. Thus, in addition to semantically significant input and output relations, the programs may use temporary relations whose content is immaterial outside the

computation. We will state explicitly which relations are temporary and which constitute the output whenever this is not clear from the context.

A query language or computing device is called *complete* if it expresses all queries. We will discuss such languages in Chapter 18.

16.2 Complexity of Queries

We now develop a framework for measuring the complexity of queries. This is done by reference to TMs and classical complexity classes defined using the TM model.

There are several ways to look at the complexity of queries. They differ in the parameters with respect to which the complexity is measured. The two main possibilities are as follows:

- *data complexity*: the complexity of evaluating a *fixed query* for variable database inputs; and
- *expression complexity*: the complexity of evaluating, on a *fixed database instance*, the various queries specifiable in a given query language.

Thus in the data complexity perspective, the complexity is with respect to the database input and the query is considered constant. Conversely, with expression complexity, the database input is fixed and the complexity is with respect to the size of the query expression. Clearly, the measures provide different information about the complexity of evaluating queries. The usual situation is that the size of the database input dominates by far the size of the query, and so data complexity is typically most relevant. This is the primary focus of Part E, and we use the term *complexity* to refer to data complexity unless otherwise stated.

The complexity of queries is defined based on the *recognition problem* associated with the query. For a query q, the recognition problem is as follows: Given an instance $\mathbf{I}$ and a tuple u, determine if u belongs to the answer $q(\mathbf{I})$. To be more precise, the recognition problem of a query q is the language

$$\{enc_\alpha(\mathbf{I})\#enc_\alpha(u) \mid u \in q(\mathbf{I}), \alpha \text{ an enumeration of } adom(\mathbf{I})\}.$$

The (*data*) *complexity* of q is the (conventional) complexity of its recognition problem. Technically, the complexity is with respect to the size of the input [i.e., the length of the word $enc_\alpha(\mathbf{I})\#enc_\alpha(u)$]. Because for an instance $\mathbf{I}$ the size (number of tuples) in $\mathbf{I}$ is closely related to the length of $enc_\alpha(\mathbf{I})$ (see Exercise 16.12), the size of $\mathbf{I}$ is usually taken as the measure of the input.

For each Turing time or space complexity class C, one can define a corresponding *complexity class of queries*, denoted by QC . The class of queries QC consists of all queries whose recognition problem is in C. For example, the class QPTIME consists of all queries for which the recognition problem is in PTIME.

There is another way to define the complexity of queries that is based on the complexity of actually *constructing the result* of the query rather than the recognition problem for individual tuples. The two definitions are in most cases interchangeable (see Exercise 16.13). In particular, for complexity classes insensitive to a polynomial factor, the

definitions are equivalent. In general, the definition based on constructing the result distinguishes between a query with a large answer and one with a small answer, which is irrelevant to the definition based on recognition. On the other hand, the definition based on constructing the result may not distinguish between easy and hard queries with large results.

Example 16.2.1 Consider a database consisting of one binary relation G and the three queries *cross*, *path*, and *self* on G defined as follows:

$$\begin{aligned} cross(G) &= \pi_1(G) \times \pi_2(G), \\ path(G) &= \{\langle x, y\rangle \mid x \text{ and } y \text{ are connected by a path in } G\}, \\ self(G) &= G. \end{aligned}$$

Consider first *cross* and *path*. Both have potentially large answers, but *cross* is clearly easier than *path*, even though the time complexity of constructing the result is $O(n^2)$ for both *cross* and *path*. The time complexity of the recognition problem is $O(n)$ for *cross* and $O(n^2)$ for *path*. Thus the measure based on constructing the result does not detect a difference between *cross* and *path*, whereas this is detected by the complexity of the recognition problem. Next consider *cross* and *self*. The time complexity of the recognition problem is in both cases $O(n)$, but the complexity of computing the result is $O(n)$ for *self* whereas it is $O(n^2)$ for *cross*. Thus the complexity of the recognition problem does not distinguish between *cross* and *self*, although *cross* can potentially generate a much larger answer. This difference is detected by the complexity of constructing the result.

In Part E, we will use the definition of query complexity based on the associated recognition problem.

16.3 Languages and Complexity

In the previous section we studied a definition of the complexity of an *individual query*. To measure the complexity of a query language L, we need to establish a correspondence between

- the class of queries expressible in L, and
- a complexity class QC of queries.

Expressiveness with Respect to Complexity Classes

The most straightforward connection between L and a class of queries QC is when L and QC are precisely the same.[1] In this case, it is said that L *expresses* QC. In every case, each query in L has complexity C, and conversely L can express every query of complexity C.

---

[1] By abuse of notation, we also denote by L the set of queries expressible in L.

Ideally, one would be able to perform complexity-tailored language design; that is, for a desired complexity C, one would design a language expressing precisely QC. Unfortunately, we will see that this is not always possible. In fact, there are no such results for the pure relational model for complexity classes of polynomial time and below, that are of most interest. We consider this phenomenon at length in the next chapter. Intuitively, the shapes of classes of queries of low complexity do not match those of classes of queries defined by any known language. Therefore we are led to consider a less straightforward way to match languages to complexity classes.

Completeness with Respect to Complexity Classes

Consider a language L that does not correspond precisely to any natural complexity class of queries. Nonetheless we would like to say something about the complexity of queries in L. For instance, we may wish to guarantee that all queries in L lie within some complexity class C, even though L may not express *all* of QC. For the bound to be meaningful, we would also like that C is, in some sense, a tight upper bound for the complexity of queries in L. In other words, L should be able to express at least some queries that are among the hardest in QC. The property of a problem being hardest in a complexity class C is captured, in complexity theory, by the notion of *completeness* of the problem in the class (see Chapter 2). By extension to a language, this leads to the following:

DEFINITION 16.3.1 A language L is *complete with respect to a complexity class* C if

(a) each query in L is also in QC, and

(b) there exists a query in L for which the associated recognition problem is complete with respect to the complexity class C.

As in the classical definition of completeness of a problem in a complexity class, we qualify, when necessary, the notion of a completeness in a complexity class by the complexity of the reduction. For instance, *L is logspace complete with respect to* C qualifies (b) by stating that the query expressible in L whose recognition problem is complete in C is in fact *logspace complete* in C.

In some sense, completeness without expressiveness says something negative about the language L. L can express some queries that are as hard as any query in QC; on the other hand, there may be *easy* queries in QC that are not expressible in L. This may at first appear contradictory because L expresses some queries that are complete in C, and any problem in C can be reduced to the complete problem. However, there is no contradiction. The *reduction* of the "easy" query to the complete query may be computationally easy but nevertheless not expressible in L. Examples of this situation involve the familiar languages *fixpoint* and *while*. As will be shown in Section 17.3, these languages are complete in PTIME and PSPACE, respectively. However, neither can express the simple parity query on a unary relation R:

$$even(R) = true \text{ if } |R| \text{ is even, and } false \text{ otherwise.}$$

Complexity and Genericity

To conclude this chapter, we consider the delicate impact of genericity on complexity. The foregoing query *even* illustrates a fundamental phenomenon relating genericity to the complexity of queries. As stated earlier, *even* cannot be computed by *fixpoint* or by *while*, both of which are powerful languages. The difficulty in computing *even* is due to the lack of information about the elements of the set. Because the database only provides a set of undifferentiated elements, genericity implies that they are treated uniformly in queries. This rules out the straightforward solution of repeatedly extracting one arbitrary element from the set until the set is empty while keeping a binary counter: How does one specify the first element to be extracted?

On the other hand, consider the problem of computing *even* with a TM. The additional information provided by the encoding of the input on the tape makes the problem trivial and allows a linear-time solution.

This highlights the interesting fact that genericity may complicate the task of computing a query, whereas access to the internal representation may simplify this task considerably. Thus this suggests a trade-off between genericity and complexity. This can be formalized by defining complexity classes based on a computing device that is generic by definition in place of a TM. Such a device cannot take advantage of the representation of data in the same manner as a TM, and it treats data generically at all points in the computation. It can be shown that *even* is hard with respect to complexity measures based on such a device. The query *even* will be used repeatedly to illustrate various aspects of the complexity of queries.

Bibliographic Notes

The study of computable queries originated in the work of Chandra and Harel [CH80b, Cha81a, CH82]. In addition to well-typed languages, they also considered languages defining queries with data-dependent output schemas. The data and expression complexity of queries were introduced and studied in [CH80a, CH82] and further investigated in [Var82a]. Data complexity is most widely used and is based on the associated recognition problem. Data complexity based on constructing the result of the query is discussed in [AV90].

The notion of genericity was formalized in [AU79, CH80b] with different terminology. The term *C-genericity* was first used in [HY84]. Other notions related in spirit to genericity are studied in [Hul86]. The definition of genericity is extended in [AK89] to object-oriented queries that can produce new constants in the result (arising from new object identifiers); see also [VandBGAG92, HY90]. This is further discussed in Chapters 18 and 21.

A modified notion of Turing machine is introduced in [HS93] that permits domain elements to appear on the Turing tape, thus obviating the need to encode them. However, this device still uses an ordered representation of the input instance. A device operating directly on relations is the on-site acceptor of [Lei89a]. This extends the formal algorithmic procedure (FAP) proposed in [Fri71] in the context of recursion theory. Another variation of this device is presented in [Lei89b]. Further generalizations of TMs, which do not assume an ordered input, are introduced in [AV91b, AV94]. These are used to define nonstandard

complexity classes of queries and to investigate the trade-off between genericity and complexity.

Informative discussions of the connection between query languages and complexity classes are provided in [Gur84, Gur88, Imm87b, Lei89a].

Exercises

Exercise 16.1 Let q be a C-generic mapping. Show that, for each input instance $\mathbf{I}$, $adom(q(\mathbf{I})) \subseteq C \cup adom(\mathbf{I})$.

Exercise 16.2 (Genericity) Let q be a generic database mapping from $\mathbf{R}$ to S.

(a) Let α and β be enumerations of **dom**, and suppose that M computes q using enc_α. Prove that for each instance $\mathbf{I}$ over $\mathbf{R}$,

$$enc_\alpha \circ M \circ enc_\alpha^{-1} = enc_\beta \circ M \circ enc_\beta^{-1}.$$

Conclude that M computes q using enc_β.

(b) Verify that the definitions of *computable relative to* α and *computable* are equivalent for generic database mappings.

★ **Exercise 16.3** Let $\mathbf{R}$ be a database schema and S a relation schema.

(a) Prove that it is undecidable to determine, given TM M that computes a mapping q from $inst(\mathbf{R})$ to $inst(S)$ relative to enumeration α of **dom**, whether q is generic.

(b) Show that the set of TMs that compute queries from $\mathbf{R}$ to S is co-r.e.

Exercise 16.4 In many practical situations the underlying domains used (e.g., strings, integers) have some structure (e.g., an ordering relationship that is visible to both user and implementation). For each of the following, develop a natural definition for *generic* and exhibit a nongeneric query, if there is one.

(a) **dom** is partitioned into several sorts $\mathbf{dom}_1, \ldots, \mathbf{dom}_n$.

(b) **dom** has a dense total order $\leq$. [A total order $\leq$ is *dense* if $\forall x, y(x < y \rightarrow \exists z(x < z \wedge z < y))$.]

(c) **dom** has a discrete total order $\leq$. [A total order $\leq$ is *discrete* if $\forall x[\exists y(x < y \rightarrow \exists z(x < z \wedge \neg\exists w(x < w \wedge w < z))) \wedge \exists y(y < x \rightarrow \exists z(z < x \wedge \neg\exists w(z < w \wedge w < x)))]$.]

(d) **dom** is the set of nonnegative integers and has the usual ordering $\leq$.

Exercise 16.5 Let q be a C-generic query, and let $\mathbf{I}$ be an input instance. Let ρ be an automorphism of $\mathbf{I}$ that is the identity on C, and let a, b be constants in $\mathbf{I}$, such that $\rho(a) = b$. Show that a occurs in $q(\mathbf{I})$ iff b occurs in $q(\mathbf{I})$.

The next several exercises use the following notions. Let $\mathbf{R}$ be a database schema. Let k be a positive integer and $\mathbf{I}$ an instance over $\mathbf{R}$. $\Delta_k^{\mathbf{I}}$ denotes the set of k-tuples that can be formed using just constants in $\mathbf{I}$. Define the following relation $\equiv_k^{\mathbf{I}}$ on $\Delta_k^{\mathbf{I}}$: $u \equiv_k^{\mathbf{I}} v$ iff there exists an automorphism ρ of $\mathbf{I}$ such that $\rho(u) = v$. The *k-type index* of $\mathbf{I}$, denoted $\#_k(\mathbf{I})$, is the number of equivalence classes of $\equiv_k^{\mathbf{I}}$.

Exercise 16.6 (Equivalence induced by automorphisms) Let $\mathbf{R}$ be a database schema and $\mathbf{I}$ an instance of R.

(a) Show that $\equiv_k^{\mathbf{I}}$ is an equivalence relation on $\Delta_k^{\mathbf{I}}$.

(b) Let q be a generic query on $\mathbf{R}$, whose output is a k-ary relation. Show that $q(\mathbf{I})$ is a union of equivalence classes of $\equiv_k^{\mathbf{I}}$.

♠ **Exercise 16.7** (Type index) Let G be a binary relation schema corresponding to the edges of a directed graph. Show the following:

(a) The k-type index of a complete graph is a constant independent of the size of the graph, as long as it has at least k vertexes.

(b) The k-type index of graphs consisting of a simple path is polynomial in the size of the graph.

(c) [Lin90, Lin91] The k-type index of a complete binary tree is polynomial in the *depth* of the tree.

Exercise 16.8 Let k, n be integers, $0 < n < k$, and $\mathbf{I}$ an instance over schema $\mathbf{R}$.

(a) Show how to compute $\equiv_n^{\mathbf{I}}$ from $\equiv_k^{\mathbf{I}}$.

(b) Prove that $\#_n(\mathbf{I}) < \#_k(\mathbf{I})$, unless $\mathbf{I}$ has just one constant.

★ **Exercise 16.9** (Fixpoint queries and type index) Let φ be a *fixpoint* query on database schema $\mathbf{R}$. Show that there exists a polynomial p such that, for each instance $\mathbf{I}$ over $\mathbf{R}$, φ on input $\mathbf{I}$ terminates after at most $p(\#_k(\mathbf{I}))$ steps, for some $k > 0$.

♠ **Exercise 16.10** (Fixpoint queries on special graphs) Show that every *fixpoint* query terminates in

(a) constant number of steps on complete graphs;

(b) [Lin90, Lin91] $p(\log(|\mathbf{I}|))$ number of steps on complete binary trees $\mathbf{I}$, for some polynomial p. *Hint:* Use Exercises 16.7 and 16.9.

♠ **Exercise 16.11** [Ban78, Par78] Let $\mathbf{R}$ be a schema, $\mathbf{I}$ a fixed instance over $\mathbf{R}$, and $a_1, \ldots, a_n$ an enumeration of $adom(\mathbf{I})$. For each automorphism ρ on $\mathbf{I}$, let $t_\rho = \langle \rho(a_1), \ldots, \rho(a_n) \rangle$, and let

$$auto(\mathbf{I}) = \{t_\rho \mid \rho \text{ an automorphism of } \mathbf{I}\}.$$

(a) Prove that there is a CALC query q with no constants (depending on $\mathbf{I}$) such that $q(\mathbf{I}) = auto(\mathbf{I})$.

(b) Prove that for each relation schema S and instance J over S with $adom(J) \subseteq adom(\mathbf{I})$,

$$\begin{gathered}\text{there is a CALC query } q \text{ with no constants} \\ \text{(depending on } \mathbf{I} \text{ and } J) \\ \text{such that } q(\mathbf{I}) = J \\ \text{iff} \\ \text{for each automorphism } \rho \text{ of } \mathbf{I},\ \rho(J) = J.\end{gathered}$$

A query language is called BP-*complete* if it satisfies the "if" direction of part (b).

Exercise 16.12 (Tape encoding of instances) Let $\mathbf{I}$ be a nonempty instance of a database schema $\mathbf{R}$. Let n_c be the number of constants in $\mathbf{I}$, n_t the number of tuples, and α an enumeration of the constants in $\mathbf{I}$. Show that there exist integers k_1, k_2, k_3 depending only on $\mathbf{R}$ such that

(a) $n_c \leq k_1 n_t \leq |enc_\alpha(\mathbf{I})|$,

(b) $|enc_\alpha(\mathbf{I})| \leq k_2 n_t \log(n_t)$,

(c) $|enc_\alpha(\mathbf{I})| \leq (n_c)^{k_3}$.

Exercise 16.13 (Recognition versus construction complexity) Let f be a time or space bound for a TM, and let q be a query. The notation *r-complexity* abbreviates the complexity based on recognition, and *a-complexity* stands for complexity based on constructing the answer. Show the following:

(a) If the time r-complexity of q is bounded by f, then there exists $k, k > 0$, such that the time a-complexity of q is bounded by $n^k f$, where n is the number of constants in the input instance.

(b) If the space r-complexity of q is bounded by f, then there exists $k, k > 0$, such that the space a-complexity of q is bounded by $n^k + f$, where n is the number of constants in the input instance.

(c) If the time a-complexity of q is bounded by f, then there exists $k, k > 0$, such that the time r-complexity of q is bounded by kf.

(d) If the space a-complexity of q is bounded by f, then the space r-complexity of q is bounded by f.

Exercise 16.14 (Data complexity of algebra) Determine the time and space complexity of each of the relational algebra operations (show the lowest complexity you can).

★ **Exercise 16.15**

(a) Develop an algorithm for computing the transitive closure of a graph that uses only the information provided by the graph (i.e., a generic algorithm).

(b) Develop algorithms for a TM to compute the transitive closure of a graph (starting from a standard encoding of the graph on the tape) that use as little time (space) as you can manage.

(c) Write a datalog program defining the transitive closure of a graph so that the number of stages in the bottom-up evaluation is as small as you can manage.