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.*

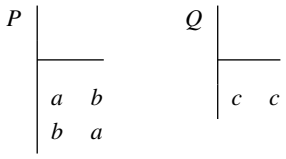
This chapter lays the groundwork for the study of the complexity and expressiveness of query languages. First the notion of query is carefully reconsidered and formally defined. Then, the complexity of individual queries is considered. Finally definitions that allow comparison of query languages and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

17 First Order, Fixpoint, and While

Alice: *I get it, now we'll match languages to complexity classes.*
Sergio: *It's not that easy—data independence adds some spice.*
Riccardo: *You can think of it as not having order.*
Vittorio: *It's a lot of fun, and we'll play some games along the way.*

In Chapter 16, we laid the framework for studying the expressiveness and complexity of query languages. In this chapter, we evaluate three of the most important classes of languages discussed so far—CALC, *fixpoint*, and *while*—with respect to expressiveness and complexity. We show that CALC is in LOGSPACE and AC_0, that *fixpoint* is complete in PTIME, and that *while* is complete in PSPACE.[1] We also investigate the impact of the presence of an ordering of the constants in the input.

We first show that CALC can be evaluated in LOGSPACE. This complexity result partly explains the success of relational database systems: Relational queries can be evaluated efficiently. Furthermore, it implies that these queries are within NC and thus that they have a high potential of intrinsic parallelism (not yet fully exploited in actual systems). We prove that CALC queries can be evaluated in constant time in a particular (standard) model of parallel computation based on circuits.

While looking at the expressive power of CALC and the other two languages, we study their limitations by examining queries that cannot be expressed in these languages. This leads us to introduce important tools that are useful in investigating the expressive power of query languages. We first present an elegant characterization of CALC based on *Ehrenfeucht-Fraissé games*. This is used to show limitations in the expressive power of CALC, such as the nonexpressibility of the transitive closure query on a graph. A second tool related to expressiveness, which applies to all languages discussed in this chapter, consists of proving *0-1 laws* for languages. This powerful approach, based on probabilities, allows us to show that certain queries (such as *even*) are not expressible in *while* and thus not in *fixpoint* or CALC.

As discussed in Section 16.3, there are simple queries that these languages cannot express (e.g., the prototypical example of *even*). Together with the completeness of *fixpoint* and *while* in PTIME and PSPACE, respectively, this suggests that there is an uneasy relationship between these languages and complexity classes. As intimated in Section 16.3, the problem can be attributed to the fact that a generic query language cannot take advantage of the information provided by the internal representation of data used by Turing machines,

[1] AC_0 and NC are two parallel complexity classes defined later in this chapter.

such as an ordering of the constants. For instance, the query *even* is easily expressible in *while* if an order is provided.

A fundamental result of this chapter is that *fixpoint* expresses exactly QPTIME under the assumption that queries can access an order on the constants. It is especially surprising that a complexity class based on such a natural resource as time coincides with a logic-based language such as *fixpoint*. However, this characterization depends on the order in a crucial manner, and this highlights the importance of order in the context of generic computation. No language is known that expresses QPTIME without the order assumption; and the existence of such a language remains one of the main open problems in the theory of query languages.

This chapter concludes with two recent developments that shed further light on the interplay of order and expressiveness. The first shows that a *while* query on an unordered database can be reduced to a *while* query on an ordered database via a *fixpoint* query. The *fixpoint* query produces an ordered database from a given unordered one by grouping tuples into a sequence of blocks that are never split in the computation of the *while query*; the blocks can then be thought of as elements of an ordered database. This also allows us to clarify the connection between *fixpoint* and *while*: They are distinct, unless PTIME = PSPACE.

The second recent development considers nondeterminism as a means for overcoming limitations due to the absence of ordering of the domain. Several nondeterministic extensions of CALC, *fixpoint*, and *while* are shown.

The impact of order is a constant theme throughout the discussion of expressive power. As discussed in Chapter 16, the need to consider computation without order is a consequence of the data independence principle, which is considered important in the database perspective. Therefore computation *with* order is viewed as a metaphor for an (at least partial) abandonment of the data independence principle.

17.1 Complexity of First-Order Queries

This section considers the complexity of first-order queries and shows that they are in QLOGSPACE. This result is particularly significant given its implications about the *parallel* complexity of CALC and thus of relational languages in general. Indeed, LOGSPACE $\subseteq$ NC. As will be seen, this means that every CALC query can be evaluated in polylogarithmic time using a polynomial number of processors. Moreover, as described in this section, a direct proof shows the stronger result that the first-order queries can in fact be evaluated in AC_0. Intuitively, this says that first-order queries can be evaluated in *constant* time with a polynomial number of processors.

We begin by showing the connection between CALC and QLOGSPACE.

THEOREM 17.1.1 CALC is included in QLOGSPACE.

Proof Let φ be a query in CALC over some database schema $\mathbf{R}$. We will describe a TM M_φ, depending on φ, that solves the recognition problem for φ and uses a work tape with length logarithmic in the size of the read-only input tape.

Suppose that M_φ is started with input $enc_\alpha(\mathbf{I})\#enc_\alpha(u)$ for some instance $\mathbf{I}$ over $\mathbf{R}$,

some enumeration α of the constants, and some tuple u over $adom(\mathbf{I})$ whose arity is the same as that of the result of φ. M_φ should accept the input iff $u \in \varphi(\mathbf{I})$. We assume w.l.o.g. that φ is in prenex normal form. We show by induction on the number of quantifiers of φ that the computation can be performed using $k \cdot \log(|enc_\alpha(\mathbf{I})\#enc_\alpha(u)|)$ cells of the work tape, for some constant k.

Basis. If φ has no quantifiers, then all the variables of φ are free. Let ν be the valuation mapping the free variables of φ to u. M_φ must determine whether $\mathbf{I} \models \varphi[\nu]$. To determine the truth value of each literal L under ν occurring in φ, one needs only scan the input tape looking for $\nu(L)$. This can be accomplished by considering each tuple of $\mathbf{I}$ in turn, comparing it with relevant portions of u. For each such tuple, the address of the beginning of the tuple should be stored on the tape along with the offset to the current location of the tuple being scanned. This can be accomplished within logarithmic space.

Induction. Now suppose that each prenex normal form CALC formula with less than n quantifiers can be evaluated in LOGSPACE, and let φ be a prenex normal form formula with n quantifiers. Suppose φ is of the form $\exists x\ \psi$. (The case when φ is of the form $\forall x\ \psi$ is similar.)

All possible values of x are tried. If some value is found that makes ψ true, then the input is accepted; otherwise it is rejected. The values used for x are all those that appear on the input tape in the order in which they appear. To keep track of the current value of x, one needs $\log(n_c)$ work tape cells, where n_c is the number of constants in $\mathbf{I}$. Because n_c is less than the length of the input, the number of cells needed is no more than $\log(|enc_\alpha(\mathbf{I})\#enc_\alpha(u)|)$. The problem is now reduced to evaluating ψ for each value of x. By the induction hypothesis, this can be done using $k \cdot \log(|enc_\alpha(\mathbf{I})\#enc_\alpha(u)|)$ work tape cells for some k. Thus the entire computation takes $(k+1)\log(|enc_\alpha(\mathbf{I})\#enc_\alpha(u)|)$ work tape cells; which concludes the induction. ■

Unfortunately, CALC does not express all of QLOGSPACE. It will be shown in Section 17.3 that *even*, although clearly in QLOGSPACE, is not a first-order query.

We next consider informally the *parallel* complexity of CALC. We are concerned with two parallel complexity classes: NC and AC_0. Intuitively, NC is the class of problems that can be solved using polynomially many processors in time polynomial in the logarithm of the input size; AC_0 also allows polynomially many processors but only *constant* time. The formal definitions of NC and AC_0 are based on a circuit model in which time corresponds to the depth of the circuit and the number of gates corresponds to its size. The circuits use *and*, *or*, and *not* gates and have unbounded fan-in.[2] Thus AC_0 is the class of problems definable using circuits where the depth is constant and the size polynomial in the input.

The fact that the complexity of CALC is LOGSPACE implies that its parallel complexity is NC, because it is well known that LOGSPACE $\subseteq$ NC. However, one can prove a tighter result, which says that the parallel complexity of CALC is in fact AC_0. So only constant time is needed to evaluate CALC queries. More than any other known complexity result on CALC, this captures the fundamental intuition that first-order queries can be evaluated in

[2] The *fan-in* is the number of wires going into a gate.

parallel very efficiently and that they represent, in some sense, primitive manipulations of relations.

We sketch only the proof and leave the details for Exercise 17.2.

THEOREM 17.1.2 Every CALC query is in AC_0.

Crux Let us first provide an intuition of the result independent of the circuit model. We will use the relational algebra. We will argue that each of the operations $\pi, \sigma, \times, -, \cup$ can be performed in constant parallel time using only polynomially many processors.

Let e be an expression in the algebra over some database schema $\mathbf{R}$. Consider the following infinite space of processors. There is one processor for each pair $\langle f, u\rangle$, where f is a subexpression of e and u is a tuple of the same arity as the result of f, using constants from **dom**. Let us denote one such processor by $p_{f,u}$. Note that, in particular, for each relation name Q occurring in f and each u of the arity of Q, $p_{Q,u}$ is one of the processors. Each processor has two possible states, *true* or *false*, indicating whether u is in the result of f.

At the beginning, all processors are in state *false*. An input instance is specified by turning on the processors corresponding to tuples in the input relations (i.e., processors $p_{R,u}$ if u is in input relation R). The result consists of the tuples u for which $p_{e,u}$ is in state *true* at the end of the computation. For a given input, we are only concerned with the processors formed from tuples with constants occurring in the input. Clearly, no more than polynomially many processors will be relevant during the computation.

It remains to show that each algebra operation takes constant time. Consider, for instance, cross product. Suppose $f \times g$ is a subexpression of e. To compute $f \times g$, the processors $p_{f,u}$ and $p_{g,v}$ send the message *true* to processor $p_{(f\times g),uv}$ if their state is *true*. Processor $p_{(f\times g),uv}$ goes to state *true* when receiving two *true* messages. The other operations are similar. Thus e is evaluated in constant time in our informal model of parallel computation.

To formalize the foregoing intuition using the circuit model, one must construct, for each n, a circuit B_n that, for each input of length n consisting of an encoding over the alphabet $\{0, 1\}$ of an instance $\mathbf{I}$ and a tuple u, outputs 1 iff $u \in e(\mathbf{I})$. The idea for constructing the circuit is similar to the informal construction in the previous paragraph except that processors are replaced by wires (edges in the graph representing the circuit) that carry either the value 1 or 0. Moreover, each B_n has polynomial size. Thus only wires that can become active for some input are included. Figure 17.1 represents fragments of circuits computing some relational operations. In the figure, f is the cross product of g and h (i.e., $g \times h$); f' is the difference $g - h$; and f'' is the projection of h on the first coordinate. Observe that projection is the most tricky operation. In the figure, it is assumed that the active domain consists of four constants. Note also that because of projection, the circuits have unbounded fan-in.

We leave the details of the construction of the circuits B_n to the reader (see Exercise 17.2). In particular, note that one must use a slightly more cumbersome encoding than that used for Turing machines because the alphabet is now restricted to $\{0, 1\}$. ■

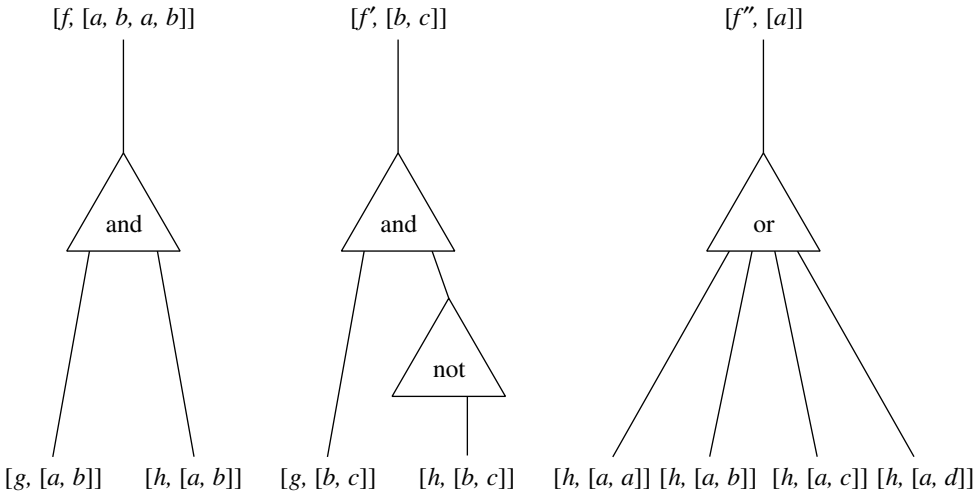


Figure 17.1: Some fragments of circuits

One might naturally wonder if CALC expresses *all* queries in AC_0. It turns out that there are queries in AC_0 that are not first order. This is demonstrated in Section 17.4.

17.2 Expressiveness of First-Order Queries

We have seen that first-order queries have desirable properties with respect to complexity. However, there is a price to pay for this in terms of expressiveness: There are many useful queries that are not first order. Typical examples of such queries are *even* and transitive closure of a graph. This section presents an elegant technique based on a two-player game that can be used to prove that certain queries (including *even* and transitive closure) are not first order. Although the game we describe is geared toward first-order queries, games provide a general technique that is used in conjunction with many other languages.

The connection between CALC sentences and games is, intuitively, the following. Consider as an example a CALC sentence of the form

$$\forall x_1 \, \exists x_2 \, \forall x_3 \, \psi(x_1, x_2, x_3).$$

One can view the sentence as a statement about a game with two players, 1 and 2, who alternate in picking values for x_1, x_2, x_3. The sentence says that Player 2 can always force a choice of values that makes $\psi(x_1, x_2, x_3)$ true. In other words, no matter which value Player 1 chooses for x_1, Player 2 can pick an x_2 such that, no matter which x_3 is chosen next by Player 1, $\psi(x_1, x_2, x_3)$ is true.

The actual game we use, called the *Ehrenfeucht-Fraissé* game, is slightly more involved, but is based on a similar intuition. It is played on two instances. Suppose that **R** is a database schema. Let **I** and **J** be instances over **R**, with disjoint sets of constants. Let r be

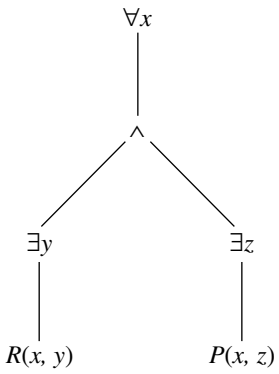


Figure 17.2: A syntax tree

a positive integer. The *game of length r associated with* $\mathbf{I}$ *and* $\mathbf{J}$ is played by two players called Spoiler and Duplicator, making r choices each. Spoiler starts by picking a constant occurring in $\mathbf{I}$ or $\mathbf{J}$, and Duplicator picks a constant in the opposite instance. This is repeated r times. At each move, Spoiler has the choice of the instance and a constant in it, and Duplicator must respond in the opposite instance.

Let a_i be the i^{th} constant picked in $\mathbf{I}$ (respectively, b_i in $\mathbf{J}$). The set of pairs $\{(a_1, b_1), \ldots, (a_r, b_r)\}$ is a *round* of the game. The *subinstance* of $\mathbf{I}$ generated by $\{a_1, \ldots, a_r\}$, denoted $\mathbf{I}/\{a_1, \ldots, a_r\}$, consists of all facts in $\mathbf{I}$ using only these constants, and similarly for $\mathbf{J}$, $\{b_1, \ldots, b_r\}$ and $\mathbf{J}/\{b_1, \ldots, b_r\}$.

Duplicator *wins the round* $\{(a_1, b_1), \ldots, (a_r, b_r)\}$ iff the mapping $a_i \to b_i$ is an isomorphism of the subinstances $\mathbf{I}/\{a_1, \ldots, a_r\}$ and $\mathbf{J}/\{b_1, \ldots, b_r\}$.

Duplicator *wins the game of length r* associated with $\mathbf{I}$ and $\mathbf{J}$ if he or she has a winning strategy (i.e., Duplicator can always win any game of length r on $\mathbf{I}$ and $\mathbf{J}$, no matter how Spoiler plays). This is denoted by $\mathbf{I} \equiv_r \mathbf{J}$. Note that the relation $\equiv_r$ is an equivalence relation on instances over $\mathbf{R}$ (see Exercise 17.3).

Intuitively, the equivalence $\mathbf{I} \equiv_r \mathbf{J}$ says that $\mathbf{I}$ and $\mathbf{J}$ cannot be distinguished by looking at just r constants at a time in the two instances. Recall that the *quantifier depth* of a CALC formula is the maximum number of quantifiers in a path from the root to a leaf in the representation of the sentence as a tree. The main result of Ehrenfeucht-Fraissé games is that the ability to distinguish among instances using games of length r is equivalent to the ability to distinguish among instances using some CALC sentence of quantifier depth r.

Example 17.2.1 Consider the sentence $\forall x\ (\exists y\ R(x, y) \wedge \exists z\ P(x, z))$. Its syntax tree is represented in Fig. 17.2. The sentence has quantifier depth 2. Note that, for a sentence in prenex normal form, the quantifier depth is simply the number of quantifiers in the formula.

The main result of Ehrenfeucht-Fraissé games, stated in Theorem 17.2.2, is that if $\mathbf{I}$ and $\mathbf{J}$ are two instances such that Duplicator has a winning strategy for the game of length r on the two instances, then $\mathbf{I}$ and $\mathbf{J}$ cannot be distinguished by any CALC sentence of

quantifier depth r. Before proving this theorem, we note that the converse of that result also holds. Thus if two instances are undistinguishable using sentences of quantifier depth r, then they are equivalent with respect to $\equiv_r$. Although interesting, this is of less use as a tool for proving expressibility results, and we leave it as a (nontrivial!) exercise. The main idea is to show that each equivalence class of $\equiv_r$ is definable by a sentence of quantifier depth r (see Exercises 17.9 and 17.10).

THEOREM 17.2.2 Let $\mathbf{I}$ and $\mathbf{J}$ be two instances over a database schema $\mathbf{R}$. If $\mathbf{I} \equiv_r \mathbf{J}$, then for each CALC sentence φ over $\mathbf{R}$ with quantifier depth r, $\mathbf{I}$ and $\mathbf{J}$ both satisfy φ or neither does.

Crux Suppose that $\mathbf{I} \models \varphi$ and $\mathbf{J} \not\models \varphi$ for some φ of quantifier depth r. We prove that $\mathbf{I} \not\equiv_r \mathbf{J}$. We provide only a sketch of the proof in an example.

Let φ be the sentence $\forall x_1\, \exists x_2\, \forall x_3\, \psi(x_1, x_2, x_3)$, where ψ has no quantifiers, and let $\mathbf{I}$ and $\mathbf{J}$ be two instances such that $\mathbf{I} \models \varphi$, $\mathbf{J} \not\models \varphi$. Then

$$\mathbf{I} \models \forall x_1\, \exists x_2\, \forall x_3\, \psi(x_1, x_2, x_3) \quad \text{and} \quad \mathbf{J} \models \exists x_1\, \forall x_2\, \exists x_3\, \neg\psi(x_1, x_2, x_3).$$

We will show that Spoiler can prevent Duplicator from winning by forcing the choice of constants a_1, a_2, a_3 in $\mathbf{I}$ and b_1, b_2, b_3 in $\mathbf{J}$ such that $\mathbf{I} \models \psi(a_1, a_2, a_3)$ and $\mathbf{J} \models \neg\psi(b_1, b_2, b_3)$. Then the mapping $a_i \to b_i$ cannot be an isomorphism of the subinstances $\mathbf{I}/\{a_1, a_2, a_3\}$ and $\mathbf{J}/\{b_1, b_2, b_3\}$, contradicting the assumption that Duplicator has a winning strategy. To force this choice, Spoiler always picks "witnesses" corresponding to the existential quantifiers in φ and $\neg\varphi$ (note that the quantifier for each variable is either $\forall$ in φ and $\exists$ in $\neg\varphi$, or vice versa).

Spoiler starts by picking a constant b_1 in $\mathbf{J}$ such that

$$\mathbf{J} \models \forall x_2\, \exists x_3\, \neg\psi(b_1, x_2, x_3).$$

Duplicator must respond by picking a constant a_1 in $\mathbf{I}$. Due to the universal quantification in φ,

$$\mathbf{I} \models \exists x_2\, \forall x_3\, \psi(a_1, x_2, x_3),$$

regardless of which a_1 was picked. Next Spoiler picks a constant a_2 in $\mathbf{I}$ such that

$$\mathbf{I} \models \forall x_3\, \psi(a_1, a_2, x_3).$$

Regardless of which constant b_2 in $\mathbf{J}$ Duplicator picks,

$$\mathbf{J} \models \exists x_3\, \neg\psi(b_1, b_2, x_3).$$

Finally Spoiler picks b_3 in $\mathbf{J}$ such that $\mathbf{J} \models \neg\psi(b_1, b_2, b_3)$; Duplicator picks some a_3 in $\mathbf{I}$, and $\mathbf{I} \models \psi(a_1, a_2, a_3)$. ■

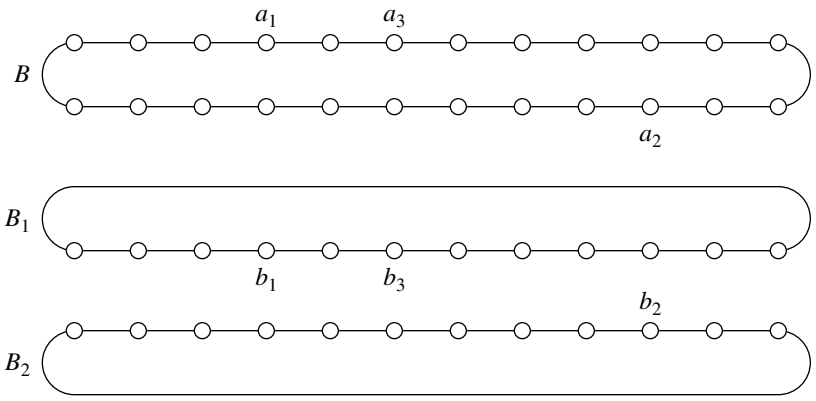


Figure 17.3: Two undistinguishable graphs

Theorem 17.2.2 provides an important tool for proving that certain properties are not definable by CALC. It is sufficient to exhibit, for each r, two instances $\mathbf{I}_r$ and $\mathbf{J}_r$ such that $\mathbf{I}_r$ has the property, $\mathbf{J}_r$ does not, and $\mathbf{I}_r \equiv_r \mathbf{J}_r$. In the next proposition, we illustrate the use of this technique by showing that graph connectivity, and therefore transitive closure, is not expressible in CALC.

PROPOSITION 17.2.3 Let $\mathbf{R}$ be a database schema consisting of one binary relation. Then the query *conn* defined by

$$conn(\mathbf{I}) = true \text{ iff } \mathbf{I} \text{ is a connected graph}$$

is not expressible in CALC.

Crux Suppose that there is a CALC sentence φ checking graph connectivity. Let r be the quantifier depth of φ. We exhibit a connected graph $\mathbf{I}_r$ and a disconnected graph $\mathbf{J}_r$ such that $\mathbf{I}_r \equiv_r \mathbf{J}_r$. Then, by Theorem 17.2.2, the two instances satisfy φ or none does, a contradiction.

For a sufficiently large n (depending only on r; see Exercise 17.5), the graph $\mathbf{I}_r$ consists of a cycle B of $2n$ nodes and the graph $\mathbf{J}_r$ of two disjoint cycles B_1 and B_2 of n nodes each (see Fig. 17.3). We outline the winning strategy for Duplicator. The main idea is simple: Two nodes a, a' in $\mathbf{I}_r$ that are far apart behave in the same way as two nodes b, b' in $\mathbf{J}_r$ that belong to different cycles. In particular, Spoiler cannot take advantage of the fact that a, a' are connected but b, b' are not. To do so, Spoiler would have to exhibit a path connecting a to a', which Duplicator could not do for b and b'. However, Spoiler cannot construct such a path because it requires choosing more than r nodes.

For example, if Spoiler picks an element a_1 in $\mathbf{I}_r$, then Duplicator picks an arbitrary element b_1, say in B_1. Now if Spoiler picks an element b_2 in B_2, then Duplicator picks an element a_2 in $\mathbf{I}_r$ far from a_1. Next, if Spoiler picks a b_3 in B_1 close to b_1, then Duplicator picks an element a_3 in $\mathbf{I}_r$ close to a_1. The graphs are sufficiently large that this can proceed

for r moves with the resulting subgraphs isomorphic. The full proof requires a complete case analysis on the moves that Spoiler can make. ■

The preceding technique can be used to show that many other properties are not expressible in CALC—for instance, *even*, 2-colorability of graphs, or Eulerian graphs (i.e., graphs for which there is a cycle that passes through each edge exactly once) (see Exercise 17.7).

17.3 *Fixpoint* and *While* Queries

That transitive closure is not expressible in CALC has been the driving force behind extending relational calculus and algebra with recursion. In this section we discuss the expressiveness and complexity of the two main extensions of these languages with recursion: the *fixpoint* and *while* queries.

It is relatively easy to place an upper bound on the complexity of *fixpoint* and *while* queries. Recall that the main distinction between languages defining *fixpoint* queries and those defining *while* queries is that the first are inflationary and the second are not (see Chapter 14). It follows that *fixpoint* queries can be implemented in polynomial time and *while* queries in polynomial space. Moreover, these bounds are tight, as shown next.

THEOREM 17.3.1

(a) The *fixpoint* queries are complete in PTIME.

(b) The *while* queries are complete in PSPACE.

Crux The fact that each *fixpoint* query is in PTIME follows immediately from the inflationary nature of languages defining the *fixpoint* queries and the fact that the total number of tuples that can be built from constants in a given instance is polynomial in the size of the instance (see Chapter 14). For *while*, inclusion in PSPACE follows similarly (see Exercise 17.11). The completeness follows from an important result that will be shown in Section 17.4. The result, Theorem 17.4.2, states that if an order on the constants of the domain is available, *fixpoint* expresses exactly QPTIME and *while* expresses exactly QPSPACE. The completeness then follows from the fact that there exist problems that are complete in PTIME and problems that are complete in PSPACE (see Exercise 17.11). ■

The Parity Query

As was the case for the first-order queries, *fixpoint* and *while* do not match precisely with complexity classes of queries. Although they are powerful, neither *fixpoint* nor *while* can express certain simple queries. The typical example is the parity query *even* on a unary relation. We next provide a direct proof that *while* (and therefore *fixpoint*) cannot express *even*. The result also follows using 0-1 laws, which are presented later. We present the direct proof here to illustrate the proof technique of hyperplanes.

PROPOSITION 17.3.2 The query *even* is not a *while* query.

Proof Let R be a unary relation. Suppose that there exists a *while* program w that computes the query *even* on input R. We can assume, w.l.o.g., that R contains a unary relation *ans* so that, on input $\mathbf{I}$, $w(\mathbf{I})(ans) = \emptyset$ if $|\mathbf{I}|$ is even, and $w(\mathbf{I}) = \mathbf{I}$ otherwise. Let $\mathbf{R}$ be the schema of w (so $\mathbf{R}$ contains R and *ans*). We will reach a contradiction by showing that the computation of w on a given input is essentially independent of its size. More precisely, for n large enough, the computations of w on all inputs of size greater than n will in some sense be identical. This contradicts the fact that *ans* should be empty at the end of some computations but not others.

To show this, we need a short digression related to computations on unary relations. We assume here that w does not use constants, but the construction can be generalized to that case (see Exercise 17.14). Let $\mathbf{I}$ be an input instance and k an integer. We consider a partition of the set of k-tuples with entries in $adom(\mathbf{I})$ into hyperplanes based on patterns of equalities and inequalities between components as follows. For each equivalence relation $\simeq$ over $\{1, \ldots, k\}$, the corresponding *hyperplane* is defined by[3]

$$H_{\simeq}(\mathbf{I}) = \{\langle u_1, \ldots, u_k\rangle \mid \text{for each } i, j \in [1, k], \\ u_i, u_j \in adom(\mathbf{I}) \text{ and } u_i = u_j \Leftrightarrow i \simeq j\}.$$

For instance, let $adom(\mathbf{I}) = \{a, b, c\}$, $k = 3$ and

$$\simeq = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 3, 3\rangle\}.$$

Then

$$H_{\simeq}(\mathbf{I}) = \{\langle a, a, b\rangle, \langle a, a, c\rangle, \langle b, b, a\rangle, \langle b, b, c\rangle, \langle c, c, a\rangle, \langle c, c, b\rangle\}.$$

Finally there are two 0-ary hyperplanes, denoted *true* and *false*, that evaluate to $\{\langle\rangle\}$ and $\{\}$, respectively.

We will see that a *while* computation cannot distinguish between two k-tuples in the same hyperplane, and so intermediate relations of arity k will always consist of a union of hyperplanes.

Now consider the *while* program w. We assume that the condition guarding each *while* loop has the form $R \neq \emptyset$ for some $R \in \mathbf{R}$, and that in each assignment $R := E$, E involves a single application of some unary or binary algebra operator. We label the statements of the program so we can talk about the program state (i.e., the label) after some number of computation steps on input $\mathbf{I}$. We include two labels in a *while* statement in the following manner:

label1 *while* ⟨*condition*⟩ *do* **label2** ⟨*statement*⟩.

[3] Note that, in logic terminology, $\simeq$ corresponds to the notion of *equality type*, and hyperplanes correspond to realizations of equality types.

Let N be the maximum arity of any relation in $\mathbf{R}$. To conclude the proof, we will show by induction on the steps of the computation that there is a number b_w such that for each input $\mathbf{I}$ with size $\geq N$, w terminates on $\mathbf{I}$ after exactly b_w steps. Furthermore,

(*) for each step $m \leq b_w$, there exists a label j_m and for each relation T of arity k a set $E_{T,m}$ of equivalence relations over $\{1, \ldots, k\}$ such that for each input $\mathbf{I}$ of size greater than N
   1. the control is at label j_m after m steps of the computation; and
   2. each T then contains $\cup\{H_{\simeq}(\mathbf{I}) \mid \simeq \text{ in } E_{T,m}\}$.

To see that this yields the result, suppose that it is true. Then for each $\mathbf{I}$ with size $\geq N$, w terminates with *ans* always empty or always nonempty, regardless of whether the size of $\mathbf{I}$ is even or odd (a contradiction).

The claim follows from an inductive proof of (*). It is clear that this holds at the 0^{th} step. At the start of the computation, all T are empty except for the input unary relation R, which contains all constants and so consists of the hyperplane $H_{\simeq}$, where $\simeq = \{\langle 1, 1\rangle\}$. Suppose now that (*) holds for each step less than m and that the program has not terminated on any $\mathbf{I}$ with size $\geq N$. We prove that (*) also holds for m. There are two cases to consider:

- Label j_{m-1} occurs before the keyword *while*. By induction, the relation controlling the loop is empty after the $(m-1)^{\text{st}}$ step, for all inputs large enough, or nonempty for all such inputs. Thus at step m, the control will be at the same label for all instances large enough, so (*1) holds. No relations have been modified, so (*2) also holds.
- Otherwise j_{m-1} labels an assignment statement. Then after the $(m-1)^{\text{st}}$ step, the control will clearly be at the label of the next statement for all instances large enough, so (*1) holds. With regard to (*2), we consider the case where the assignment is $T := Q_1 \times Q_2$ for some variables T, Q_1, and Q_2; the other relation operators are handled in a similar fashion (see Exercise 17.12). By induction, (*2) holds for all relations distinct from T because they are not modified. Consider T. After step m, T contains

$$\bigcup\{H_{\simeq_1}(\mathbf{I}) \mid \simeq_1 \text{ in } E_{Q_1,m-1}\} \times \bigcup\{H_{\simeq_2}(\mathbf{I}) \mid \simeq_2 \text{ in } E_{Q_2,m-1}\} =$$
$$\bigcup\{H_{\simeq_1}(\mathbf{I}) \times H_{\simeq_2}(\mathbf{I}) \mid \simeq_1 \text{ in } E_{Q_1,m-1}, \simeq_2 \text{ in } E_{Q_2,m-1}\}.$$

  Let k, l be the arities of Q_1, Q_2, respectively, and for each $\simeq_2$ in $E_{Q_2,m-1}$, let

$$\simeq_2^{+k} = \{(x + k, y + k) \mid (x, y) \in \simeq_2\}.$$

  For an arbitrary binary relation $\gamma \subseteq [1, k+l] \times [1, k+l]$, let γ^* denote the reflexive, symmetric, and transitive closure of γ. For $\simeq_1, \simeq_2$ in $E_{Q_1,m-1}$, $E_{Q_2,m-1}$, respectively, set

$$\simeq_1 \otimes \simeq_2 = \{(\simeq_1 \cup \simeq_2^{+k} \cup \Lambda)^* \mid \Lambda \subseteq [1,k] \times [k+1, k+l],$$
$$\text{and for all } i, i', j, j' \text{ such that } [i, j] \in \Lambda$$
$$\text{and } [i', j'] \in \Lambda, i \simeq_1 i' \text{ iff } j \simeq_2^{+k} j'\}.$$

It is straightforward to verify that for each pair $\simeq_1, \simeq_2$ in $E_{Q_1,m-1}$, $E_{Q_2,m-1}$, respectively, and $\mathbf{I}$ with size $\geq N$,

$$H_{\simeq_1}(\mathbf{I}) \times H_{\simeq_2}(\mathbf{I}) = H_{\simeq_1 \otimes \simeq_2}(\mathbf{I}).$$

Note that this uses the assumption that the size of $\mathbf{I}$ is greater than N, the maximum arity of relations in w. It follows that

$$E_{T,m} = \bigcup \{\simeq_1 \otimes \simeq_2 \mid \simeq_1 \text{ in } E_{Q_1,m-1} \text{ and } \simeq_2 \text{ in } E_{Q_2,m-1}\}.$$

Thus (*2) also holds for T at step m, and the induction is completed. ■

The hyperplane technique used in the preceding proof is based on the fact that in the context of a (sufficiently large) unary relation input, there are families of tuples (in this case the different hyperplanes) that "travel together" and hence that the intermediate and final results are unions of these families of tuples. Although there are other cases in which the technique of hyperplanes can be applied (see Exercise 17.15), in the general case the input is not a union of hyperplanes, and so the members of a hyperplane do not travel together. However, there is a generalization of hyperplanes based on automorphisms that yields the same effect. Recall that an *automorphism* of $\mathbf{I}$ is a one-to-one mapping ρ on $adom(\mathbf{I})$ such that $\rho(\mathbf{I}) = \mathbf{I}$. For fixed $\mathbf{I}$, consider the following equivalence relation $\equiv_k^{\mathbf{I}}$ on k-tuples of $adom(\mathbf{I})$: $u \equiv_k^{\mathbf{I}} v$ iff there exists an automorphism ρ of $\mathbf{I}$ such that $\rho(u) = v$. (See Exercises 16.6 and 16.7 in the previous chapter.) It can be shown that if w is a *while* query (without constants), then the members of equivalence classes $\equiv_k^{\mathbf{I}}$ travel together when w is executed on input $\mathbf{I}$. More precisely, suppose that $\mathbf{J}$ is an instance obtained at some point in the computation of w on input $\mathbf{I}$. The genericity of *while* programs implies that if ρ is an automorphism of $\mathbf{I}$, it is also an automorphism of $\mathbf{J}$. Thus for each k-tuple u in some relation of $\mathbf{J}$ and each v such that $u \equiv_k^{\mathbf{I}} v$, v also belongs to that relation. Thus each relation in $\mathbf{J}$ of arity k is a union of equivalence classes of $\equiv_k^{\mathbf{I}}$. The equivalence relation $\equiv_k^{\mathbf{I}}$ will be used in our development of 0-1 laws, presented next.

0-1 Laws

We now develop a powerful tool that provides a uniform approach to resolving in the negative a large spectrum of expressibility problems. It is based on the probability that a property is true in instances of a given size. We shall prove a surprising fact: All properties expressible by a *while* query are "almost surely" true, or "almost surely" false. More precisely, we prove the result for *while* sentences:

DEFINITION 17.3.3 A *sentence* is a total query that is Boolean (i.e., returns as answer either *true* or *false*).

Let q be a sentence over some schema $\mathbf{R}$. For each n, let $\mu_n(q)$ denote the fraction of instances over $\mathbf{R}$ with entries in $\{1, \ldots, n\}$ that satisfy q. That is,

$$\mu_n(q) = \frac{|\{\mathbf{I} \mid q(\mathbf{I}) = \mathit{true} \text{ and } \mathit{adom}(\mathbf{I}) = \{1, \ldots, n\}\}|}{|\{\mathbf{I} \mid \mathit{adom}(\mathbf{I}) = \{1, \ldots, n\}\}|}.$$

DEFINITION 17.3.4 A sentence q is *almost surely true* (*false*) if $\lim_{n\to\infty} \mu_n(q)$ exists and equals 1 (0). If every sentence in a language L is almost surely true or almost surely false, the language L *has a 0-1 law*.

To simplify the discussion of 0-1 laws, we continue to focus exclusively on constant-free queries (see Exercise 17.19).

We will show that CALC, *fixpoint*, and *while* sentences have 0-1 laws. This provides substantial insight into limitations of the expressive power of these languages and can be used to show that they cannot express a variety of properties. For example, it follows immediately that *even* is not expressible in either of these languages. Indeed, $\mu_n(\mathit{even})$ is 1 if n is even and 0 if n is odd. Thus $\mu_n(\mathit{even})$ does not converge, so *even* is not expressible in a language that has a 0-1 law.

While 0-1 laws provide an elegant and powerful tool, they require the development of some nontrivial machinery. Interestingly, this is one of the rare occasions when we will need to consider *infinite* instances even though we aim to prove something about finite instances only.

We start by proving that CALC has a 0-1 law and then extend the result to *fixpoint* and *while*. For simplicity, we consider only the case when the input to the query is a binary relation G (representing edges in a directed graph with no edges of the form $\langle a, a\rangle$). It is straightforward to generalize the development to arbitrary inputs (see Exercise 17.19).

We will use an infinite set $\mathcal{A}$ of CALC sentences called *extension axioms*, which refer to graphs. They say, intuitively, that every subgraph can be extended by one node in all possible ways. More precisely, $\mathcal{A}$ contains, for each k, all sentences of the form

$$\forall x_1 \ldots \forall x_k((\bigwedge_{i\neq j}(x_i \neq x_j)) \Rightarrow \exists y(\bigwedge_{i}(x_i \neq y) \wedge \mathit{connections}(x_1, \ldots, x_k; y))),$$

where $\mathit{connections}(x_1, \ldots, x_k; y)$ is some conjunction of literals containing, for each x_i, one of $G(x_i, y)$ or $\neg G(x_i, y)$, and one of $G(y, x_i)$ or $\neg G(y, x_i)$. For example, for $k = 3$, one of the 2^6 extension axioms is

$$\begin{aligned}\forall x_1, x_2, x_3\ ((x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_3 \neq x_1) \Rightarrow \\ \exists y\ (x_1 \neq y \wedge x_2 \neq y \wedge x_3 \neq y \wedge \\ G(x_1, y) \wedge \neg G(y, x_1) \wedge \neg G(x_2, y) \wedge \neg G(y, x_2) \wedge G(x_3, y) \wedge G(y, x_3)))\end{aligned}$$

specifying the pattern of connections represented in Fig. 17.4.

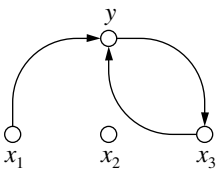


Figure 17.4: A connection pattern

A graph G satisfies this particular extension axiom if for each triple x_1, x_2, x_3 of distinct vertexes in G, there exists a vertex y connected to x_1, x_2, x_3, as shown in Fig. 17.4.

Note that $\mathcal{A}$ consists of an infinite set of sentences and that each finite subset of $\mathcal{A}$ is satisfied by some infinite instance. (The instance is obtained by starting from one node and repeatedly adding nodes required by the extension axioms in the subset.) Then by the compactness theorem there is an infinite instance satisfying all of $\mathcal{A}$, and by the Löwenheim-Skolem theorem (see Chapter 2) there is a countably infinite instance $\mathcal{R}$ satisfying $\mathcal{A}$.

The following lemma shows that $\mathcal{R}$ is unique up to isomorphism.

Lemma 17.3.5 If $\mathcal{R}$ and $\mathcal{P}$ are two countably infinite instances over G satisfying all sentences in $\mathcal{A}$, then $\mathcal{R}$ and $\mathcal{P}$ are isomorphic.

Proof Suppose that $a_1a_2\ldots$ is an enumeration of all constants in $\mathcal{R}$, and $b_1b_2\ldots$ is an enumeration of those in $\mathcal{P}$. We construct an isomorphism between $\mathcal{R}$ and $\mathcal{P}$ by alternatingly picking constants from $\mathcal{R}$ and from $\mathcal{P}$. We construct sequences $a_{i_1}\ldots a_{i_k}\ldots$ and $b_{i_1}\ldots b_{i_k}\ldots$ such that $a_{i_k} \to b_{i_k}$ is an isomorphism from $\mathcal{R}$ to $\mathcal{P}$. The procedure for picking the k^{th} constants a_{i_k} and b_{i_k} in these sequences is defined inductively as follows. For the base case, let $a_{i_1} = a_1$ and $b_{i_1} = b_1$. Suppose that sequences $a_{i_1}\ldots a_{i_k}$ and $b_{i_1}\ldots b_{i_k}$ have been defined. If k is even, let $a_{i_{k+1}}$ be the first constant in $a_1, a_2, \ldots$ that does not occur so far in the sequence. Let σ_k be the sentence in $\mathcal{A}$ describing the way $a_{i_{k+1}}$ extends the subgraph with nodes $a_{i_1}\ldots a_{i_k}$. Because $\mathcal{P}$ also satisfies σ_k, there exists a constant b in $\mathcal{P}$ that extends the subgraph $b_{i_1}\ldots b_{i_k}$ in the same manner. Let $b_{i_{k+1}} = b$. If k is odd, the procedure is reversed (i.e., it starts by choosing first a new constant from $b_1, b_2, \ldots$). This back-and-forth procedure ensures that (1) all constants from both $\mathcal{R}$ and $\mathcal{P}$ occur eventually among the chosen constants, and (2) the mapping $a_{i_k} \to b_{i_k}$ is an isomorphism. ■

Thus the foregoing proof shows that there exists a unique (up to isomorphism) countable graph $\mathcal{R}$ satisfying $\mathcal{A}$. This graph, studied extensively by Rado [Rad64] and others, is usually referred to as the *Rado graph*. We can now prove the following crucial lemma. The key point is the equivalence between (a) and (c), called the *transfer property*: It relates satisfaction of a sentence by the Rado graph to the property of being almost surely true.

Lemma 17.3.6 Let $\mathcal{R}$ be the Rado graph and σ a CALC sentence. The following are equivalent:

(a) $\mathcal{R}$ satisfies σ;

(b) $\mathcal{A}$ implies σ; and

(c) σ is almost surely true.

Proof $(a) \Rightarrow (b)$: Suppose (a) holds but (b) does not. Then there exists some instance $\mathcal{P}$ satisfying $\mathcal{A}$ but not σ. Because $\mathcal{P}$ satisfies $\mathcal{A}$, $\mathcal{P}$ must be infinite. By the Lowënheim-Skolem theorem (see Chapter 2), we can assume that $\mathcal{P}$ is countable. But then, by Lemma 17.3.5, $\mathcal{P}$ is isomorphic to $\mathcal{R}$. This is a contradiction, because $\mathcal{R}$ satisfies σ but $\mathcal{P}$ does not.

$(b) \Rightarrow (c)$: It is sufficient to show that each sentence in $\mathcal{A}$ is almost surely true. Suppose this is the case and $\mathcal{A}$ implies σ. By the compactness theorem, σ is implied by some finite subset $\mathcal{A}'$ of $\mathcal{A}$. Because every sentence in $\mathcal{A}'$ is almost surely true, the conjunction $\bigwedge \mathcal{A}'$ of these sentences is almost surely true. Because σ is true in every instance where $\bigwedge \mathcal{A}'$ is true, $\mu_n(\sigma) \geq \mu_n(\bigwedge \mathcal{A}')$, so $\mu_n(\sigma)$ converges to 1 and σ is almost surely true.

It remains to show that each sentence in $\mathcal{A}$ is almost surely true. Consider the following sentence σ_k in $\mathcal{A}$:

$$\forall x_1 \ldots \forall x_k((\bigwedge_{i \neq j}(x_i \neq x_j)) \rightarrow \exists y(\bigwedge_i (x_i \neq y) \wedge \mathit{connections}(x_1, \ldots, x_k; y))).$$

Then $\neg\sigma_k$ is the sentence

$$\exists x_1 \ldots \exists x_k((\bigwedge_{i \neq j}(x_i \neq x_j)) \wedge \forall y(\bigwedge_i (x_i \neq y) \rightarrow \neg \mathit{connections}(x_1, \ldots, x_k; y))).$$

We will show the following property on the probability that an instance with n constants *does not* satisfy σ_k:

$$(**) \qquad \mu_n(\neg\sigma_k) \leq n \cdot (n-1) \cdot \ldots \cdot (n-k) \cdot (1 - \frac{1}{2^{2k}})^{(n-k)}.$$

Because $\lim_{n\to\infty}[n \cdot (n-1) \cdot \ldots \cdot (n-k) \cdot (1 - \frac{1}{2^{2k}})^{(n-k)}] = 0$, it follows that $\lim_{n\to\infty} \mu_n(\neg\sigma_k) = 0$, so $\neg\sigma_k$ is almost surely false, and σ_k is almost surely true.

Let N be the number of instances with constants in $\{1, \ldots, n\}$. To prove (**), observe the following:

1. For some fixed distinct $a_1, \ldots, a_k, b$ in $\{1, \ldots, n\}$, the number of $\mathbf{I}$ satisfying some fixed literal in $\mathit{connections}(a_1, \ldots, a_k; b)$ is $\frac{1}{2} \cdot N$.
2. For some fixed distinct $a_1, \ldots, a_k, b$ in $\{1, \ldots, n\}$, the number of $\mathbf{I}$ satisfying $\mathit{connections}(a_1, \ldots, a_k; b)$ is $\frac{1}{2^{2k}} \cdot N$ (because there are $2k$ literals in *connections*).
3. The number of $\mathbf{I}$ *not* satisfying $\mathit{connections}(a_1, \ldots, a_k; b)$ is therefore $N - \frac{1}{2^{2k}} \cdot N = (1 - \frac{1}{2^{2k}}) \cdot N$.

4. For some fixed $a_1, \ldots, a_k$ in $\{1, \ldots, n\}$, the number of $\mathbf{I}$ satisfying

$$\forall y(\bigwedge_i (a_i \neq y) \rightarrow \neg \mathit{connections}(a_1, \ldots, a_k; y))$$

is $(1 - \frac{1}{2^{2k}})^{n-k} \cdot N$ [because there are $(n-k)$ ways of picking b distinct from $a_1, \ldots, a_k$)].

5. The number of $\mathbf{I}$ satisfying $\neg\sigma_k$ is thus at most

$$n \cdot (n-1) \cdot \ldots \cdot (n-k) \cdot (1 - \frac{1}{2^{2k}})^{(n-k)} \cdot N$$

(from the choices of $a_1, \ldots, a_k$). Hence (**) is proven.

(See Exercise 17.16.)

$(c) \Rightarrow (a)$: Suppose that $\mathcal{R}$ does not satisfy σ (i.e., $\mathcal{R} \models \neg\sigma$). Because $(a) \Rightarrow (c)$, $\neg\sigma$ is almost surely true. Then σ cannot be almost surely true (a contradiction). ■

The 0-1 law for CALC follows immediately.

THEOREM 17.3.7 Each sentence in CALC is almost surely true or almost surely false.

Proof Let σ be a CALC sentence. The Rado graph $\mathcal{R}$ satisfies either σ or $\neg\sigma$. By the transfer property [$(a) \Rightarrow (c)$ in Lemma 17.3.6], σ is almost surely true or $\neg\sigma$ is almost surely true. Thus σ is almost surely true or almost surely false. ■

The 0-1 law for CALC can be extended to *fixpoint* and *while*. We prove it next for *while* (and therefore *fixpoint*). Once again the proof uses the Rado graph and extends the transfer property to the *while* sentences.

THEOREM 17.3.8 Every *while* sentence is almost surely true or almost surely false.

Proof We use as a language for the *while* queries the partial fixpoint logic CALC+μ. The main idea of the proof is to show that every CALC+μ sentence that is defined on all instances is in fact equivalent almost surely to a CALC sentence, and so by the previous result is almost surely true or almost surely false. We show this for CALC+μ sentences. By Theorem 14.4.7, we can consider w.l.o.g. only sentences involving one application of the partial fixpoint operator μ. Thus consider a CALC+μ sentence ξ of the form

$$\xi = \exists \vec{x}\ (\mu_T(\varphi(T))(\vec{t}))$$

over schema $\mathbf{R}$, where

(a) φ is a CALC formula, and

(b) $\vec{t}$ is a tuple of variables or constants of appropriate arity, and $\vec{x}$ is the tuple of distinct free variables in $\vec{t}$.

(We need the existential quantification for binding the free variables. An alternative is to have constants in $\vec{t}$ but, as mentioned earlier we do not consider constants when discussing 0-1 laws.)

Essentially, a computation of a query ξ consists of iterating the CALC formula φ until convergence occurs (if ever). Consider the sequence $\{\varphi^i(\mathbf{I})\}_{i>0}$, where $\mathbf{I}$ is an input. If $\mathbf{I}$ is finite, the sequence is periodic [i.e., there exist N and p such that, for each $n \geq N$, $\varphi^n(\mathbf{I}) = \varphi^{n+p}(\mathbf{I})$]. If $p = 1$, then the sequence converges (it becomes constant at some point); otherwise it does not. Now consider the sequence $\{\varphi^i(\mathcal{R})\}_{i>0}$, where $\mathcal{R}$ is the Rado graph. Because the set of constants involved is no longer finite, the sequence may or may not be periodic. A key point in our proof is the observation that the sequence $\{\varphi^i(\mathcal{R})\}_{i>0}$ is indeed periodic, just as in the finite case.

To see this, we use a technique similar to the hyperplane technique in the proof of Lemma 17.3.5. Let k be some integer. We argue next that for each k, there is a finite number of equivalence classes of k-tuples induced by automorphisms of $\mathcal{R}$. For each pair u, v of k-tuples with entries in $adom(\mathcal{R})$, let $u \equiv_k^{\mathcal{R}} v$ iff there exists an automorphism ρ of $\mathcal{R}$ such that $\rho(u) = v$.

Let $u \simeq_k^{\mathcal{R}} v$ if both the patterns of equality and the patterns of connection within u and v are identical. More formally, for each $u = \langle a_1, \ldots, a_k \rangle$, $v = \langle b_1, \ldots, b_k \rangle$ (where a_i and b_i are constants in $\mathcal{R}$), $u \simeq_k^{\mathcal{R}} v$ if

- for each i, j, $a_i = a_j$ iff $b_i = b_j$, and
- for each i, j, $\langle a_i, a_j \rangle$ is an edge in $\mathcal{R}$ iff $\langle b_i, b_j \rangle$ is an edge in $\mathcal{R}$.

We claim that

$$u \equiv_k^{\mathcal{R}} v \text{ iff } u \simeq_k^{\mathcal{R}} v.$$

The "only if" part follows immediately from the definitions. For the "if" part, suppose that $u \simeq_k^{\mathcal{R}} v$. To show that $u \equiv_k^{\mathcal{R}} v$, we must build an automorphism ρ of $\mathcal{R}$ such that $\rho(u) = v$. This is done by a back-and-forth construction, as in Lemma 17.3.5, using the extension axioms satisfied by $\mathcal{R}$ (see Exercise 17.18).

Because there are finitely many patterns of connection and equality among k vertexes, there are finitely many equivalence classes of $\simeq_k^{\mathcal{R}}$, so of $\equiv_k^{\mathcal{R}}$. Due to genericity of the *while* computation, each $\varphi^i(\mathcal{R})$ is a union of such equivalence classes (see Exercise 16.6 in the previous chapter). Thus there must exist $m, l, 0 \leq m < l$, such that $\varphi^m(\mathcal{R}) = \varphi^l(\mathcal{R})$. Let $N = m$ and $p = l - m$. Then for each $n \geq N$, $\varphi^n(\mathcal{R}) = \varphi^{n+p}(\mathcal{R})$. It follows that:

(1) $\{\varphi^i(\mathcal{R})\}_{i>0}$ is periodic.

Using this fact, we show the following:

(2) The sequence $\{\varphi^i(\mathcal{R})\}_{i>0}$ converges.

(3) The sentence ξ is equivalent almost surely to some CALC sentence σ.

Before proving these, we argue that (2) and (3) will imply the statement of the theorem. Suppose that (2) and (3) holds. Suppose also that σ is false in $\mathcal{R}$. By Lemma 17.3.6, σ is almost surely false. Then $\mu_n(\xi) \leq \mu_n(\xi \not\equiv \sigma) + \mu_n(\sigma)$ and both $\mu_n(\xi \not\equiv \sigma)$ and $\mu_n(\sigma)$

converge to 0, so $\lim_{n\to\infty}(\mu_n(\xi)) = 0$. Thus ξ is also almost surely false. By a similar argument, ξ is almost surely true if σ is true in $\mathcal{R}$.

We now prove (2). Let Σ_{ij} be the CALC sentence stating that φ^i and φ^j are equivalent. Suppose $\{\varphi^i(\mathcal{R})\}_{i>0}$ does not converge. Thus the period of the sequence is greater than 1, so there exist $m, j, l, m < j < l$, such that

$$\varphi^m(\mathcal{R}) = \varphi^l(\mathcal{R}) \neq \varphi^j(\mathcal{R}).$$

Thus $\mathcal{R}$ satisfies the CALC sentence

$$\chi = \Sigma_{ml} \wedge \neg\Sigma_{mj}.$$

Let $\mathbf{I}$ range over finite databases. Because ξ is defined on all finite inputs, $\{\varphi^i(\mathbf{I})\}_{i\geq 0}$ converges. On the other hand, by the transfer property (Lemma 17.3.6), χ is almost surely true. It follows that the sequence $\{\varphi^i(\mathbf{I})\}_{i>0}$ diverges almost surely. In particular, there exist finite $\mathbf{I}$ for which $\{\varphi^i(\mathbf{I})\}_{i>0}$ diverges (a contradiction).

The proof of (3) is similar. By (1) and (2), the sequence $\{\varphi^i(\mathcal{R})\}_{i>0}$ becomes constant after finitely many iterations, say N. Then ξ is equivalent on $\mathcal{R}$ to the CALC sentence $\sigma = \exists\vec{x}(\varphi^N(\vec{t}))$. Suppose $\mathcal{R}$ satisfies ξ. Thus $\mathcal{R}$ satisfies σ. Furthermore, $\mathcal{R}$ satisfies $\Sigma_{N(N+1)}$ because $\{\varphi^i(\mathcal{R})\}_{i>0}$ becomes constant at the N^{th} iteration. Thus $\mathcal{R}$ satisfies $\sigma \wedge \Sigma_{N(N+1)}$. By the transfer property for CALC, $\sigma \wedge \Sigma_{N(N+1)}$ is almost surely true. For each finite instance $\mathbf{I}$ where $\Sigma_{N(N+1)}$ holds, $\{\varphi^i(\mathbf{I})\}_{i>0}$ converges after N iterations, so ξ is equivalent to σ. It follows that ξ is almost surely equivalent to σ. The case where $\mathcal{R}$ does not satisfy ξ is similar. ■

Thus we have shown that *while* sentences have a 0-1 law. It follows immediately that many queries, including *even*, are not *while* sentences. The technique of 0-1 laws has been extended successfully to languages beyond *while*. Many languages that do not have 0-1 laws are also known, such as *existential second-order logic* (see Exercise 17.21). The precise border that separates languages that have 0-1 laws from those that do not has yet to be determined and remains an interesting and active area of research.

17.4 The Impact of Order

In this section, we consider in detail the impact of order on the expressive power of query languages. As mentioned at the beginning of this chapter, we view the assumption of order as, in some sense, suspending the data independence principle in a database. Because data independence is one of the main guiding principles of the pure relational model, it is important to understand its consequences in the expressiveness and complexity of query languages.

As illustrated by the *even* query, order can considerably affect the expressiveness of a language and the difficulty of computing some queries. Without the order assumption, no expressiveness results are known for the complexity classes of PTIME and below; that is, no

P		
b	a	c
b	b	d
c	a	d
d	b	a

$succ$	
a	b
b	c
c	d

Figure 17.5: An ordered instance

languages are known that express precisely the queries of those complexity classes. With order, there are numerous such results. We present two of the most prominent ones.

At the end of this section, we present two recent developments that further explore the interplay of order and expressiveness. The first is a normal form for *while* queries that, speaking intuitively, separates a *while* query into two components: one unordered and the second ordered. The second development increases expressive power on unordered input by introducing nondeterminism in queries.

We begin by making the notion of an ordered database more precise. A database is said to be *ordered* if it includes a designated binary relation *succ* that provides a successor relation on the constants occurring in the database. A *query on an ordered database* is a query whose input database schema contains *succ* and that ranges only over the ordered instances of the input database schema.

EXAMPLE 17.4.1 Consider the database schema $\mathbf{R} = \{P, succ\}$, where P is ternary. An ordered instance of $\mathbf{R}$ is represented in Fig. 17.5. According to *succ*, a is the first constant, b is the successor of a, c is the successor of b, and d is the successor of c. Thus a, b, c, d can be identified with the integers 1, 2, 3, 4, respectively.

We now consider the power of *fixpoint* and *while* on ordered databases. In particular, we prove the fundamental result that *fixpoint* expresses precisely QPTIME on ordered databases, and *while* expresses precisely QPSPACE on ordered databases. This shows that order has a far-reaching impact on expressiveness, well beyond isolated cases such as the *even* query. More broadly, the characterization of QPTIME by *fixpoint* (with the order assumption) provides an elegant logical description of what have traditionally been considered the tractable problems. Beyond databases, this is significant to both logic and complexity theory.

THEOREM 17.4.2

(a) *Fixpoint* expresses QPTIME on ordered databases.

(b) *While* expresses QPSPACE on ordered databases.

Proof Consider (a). We have already seen that *fixpoint* $\subseteq$ QPTIME (see Exercise 17.11), and so it remains to show that all QPTIME queries on ordered databases are expressible in *fixpoint*. Let q be a query on a database with schema $\mathbf{R}$ that includes *succ*, such that q is

in QPTIME on the ordered instances of $\mathbf{R}$. Thus there is a polynomial p and Turing machine M' that, on input $enc(\mathbf{I})\#enc(u)$, terminates in time $p(|enc(\mathbf{I})\#enc(u)|)$ and accepts the input iff $u \in q(\mathbf{I})$. (In this section, encodings of ordered instances are with respect to the enumeration of constants provided by *succ*; see also Chapter 16.) Because $q(\mathbf{I})$ has size polynomial in $\mathbf{I}$, a TM M can be constructed that runs in polynomial time and that, on input $enc(\mathbf{I})$, produces as output $enc(q(\mathbf{I}))$. We now describe the construction of a CALC+μ^+ query q_M that is equivalent to q on ordered instances of $\mathbf{R}$.

The fixpoint query q_M we construct, when given ordered input $\mathbf{I}$, will operate in three phases: (α) construct an encoding of $\mathbf{I}$ that can be used to simulate M; (β) simulate M; and (γ) decode the output of M. A key point throughout the construction is that q_M is inflationary, and so it must compute without ever deleting anything from a relation. Note that this restriction does not apply to (b), which simplifies the simulation in that case.

We next describe the encoding used in the simulation of M. The encoding is centered around a relation that holds the different configurations reached by M.

Representing a tape. Because the tape is infinite, we only represent the finite portion, polynomial in length, that is potentially used. We need a way to identify each cell of the tape. Let n_c be the number of constants in $\mathbf{I}$. Because M runs in polynomial time, there is some k such that M on input $enc(\mathbf{I})$ takes time $\leq n_c^k$, and thus $\leq n_c^k$ tape cells (see also Exercise 16.12 in the previous chapter). Consider the world of k-tuples with entries in the constants from $\mathbf{I}$. Note that there are n_c^k such tuples and that they can be lexicographically ordered using *succ*. Thus each cell can be uniquely identified by a k-tuple of constants from $\mathbf{I}$. One can define by a *fixpoint* query a $2k$-ary relation $succ_k$ providing the successor relation on k-tuples, in the lexicographic order induced by *succ* (see Exercise 17.23a). The ordered k-tuples thus allow us to represent a sequence of cells and hence M's tape.

Representing all the configurations. Note that one cannot remove the tuples representing old configurations of M due to the inflationary nature of *fixpoint* computations. Thus one represents all the configurations in a single relation. To distinguish a particular configuration (e.g., that at time i, $i \leq n_c^k$), k-columns are used as timestamp. Thus to keep track of the sequence of configurations in a computation of M, one can use a $(2k+2)$-ary relation R_M where

1. the first k columns serve as a timestamp for the configuration,
2. the next k identify the tape cells,
3. column $(2k+1)$ holds the content of the cell, and
4. column $(2k+2)$ indicates the state and position of the head.

Note that now we are dealing with a double encoding: The database is encoded on the tape, and then the tape is encoded back into R_M.

To illustrate this simple but potentially confusing situation, we consider an example. Let $\mathbf{R} = \{P, succ\}$, and let $\mathbf{I}$ be the ordered instance of $\mathbf{R}$ represented in Fig. 17.5. Then $enc(\mathbf{I})$ is represented in Fig. 17.6. We assume, without loss of generality, that symbols in the tape alphabet and the states of M are in **dom**. Parts of the first two configurations are represented in the relation shown in Fig. 17.7. The representation assumes that $k = 4$, so the arity of the relation is 10. Because this is a single-volume book, only part of the relation is shown. More precisely, we show the first tuples from the representation of the

Figure 17.6: Encoding of **I** and u on a TM tape

first two configurations. It is assumed that the original state is s and the head points to the first cell of the tape; and that in that state, the head moves to the right, changing P to 0, and the machine goes to state r. Observe that the timestamp for the first configuration is $\langle a, a, a, a\rangle$, and $\langle a, a, a, b\rangle$ for the second. Observe also the numbering of tape cells: $\langle a, a, a, a\rangle, \ldots, \langle a, a, c, d\rangle$, etc.

We can now describe the three phases of the operation of q_M more precisely: For a given ordered instance **I**, q_M

(α) computes, in R_M, a representation of the initial configuration of M on input $enc(\mathbf{I})$;

(β) computes, also in R_M, the sequence of consecutive configurations of M until termination; and

(γ) decodes the final tape contents of M, as represented in R_M, into the output relation.

We sketch the construction of the *fixpoint* queries realizing (α) and (β) here, and we leave (γ) as an exercise (17.23).

Consider phase (α). Recall that each constant is encoded on the tape of M as the binary representation of its rank in the successor relation *succ* (e.g., c as 10). To perform the encoding of the initial configuration, it is useful first to construct an auxiliary relation that provides the encoding of each constant. Because there are n_c constants, the code of each constant requires $\leq \lceil \log(n_c) \rceil$ bits, and thus less than n_c bits. We can therefore use a ternary relation *constant_coding* to record the encoding. A tuple $\langle x, y, z\rangle$ in that relation indicates that the k^{th} bit of the encoding of constant x is z, where k is the rank of constant y in the *succ* relation. For instance, the relation *constant_coding* corresponding to the *succ* in Fig. 17.5 is represented in Fig. 17.8. The tuples $\langle c, a, 1\rangle$ and $\langle c, b, 0\rangle$ indicate, for instance, that c is encoded as 10. It is easily seen that *constant_coding* is definable from *succ* by a *fixpoint* query (see Exercise 17.23b).

With relation *constant_coding* constructed, the task of computing the encoding of **I** and u into R_M is straightforward. We will illustrate this using again the example in Fig. 17.5. To encode relation P, one steps through all 3-tuples of constants and checks if a tuple in P has been reached. To step through the 3-tuples, one first constructs the successor relation $succ_3$ on 3-tuples. The first tuple in P that is reached is $\langle b, a, c\rangle$. Because this is the first tuple encoded, one first inserts into R_M the identifying information for P (the first tuple in Fig. 17.7). This proceeds, yielding the next tuples in Fig. 17.7. The binary representation for each of b, a, c is obtained from relation *constant_coding*. This proceeds by moving to the next 3-tuple. It is left to the reader to complete the details of the *fixpoint* query constructing R_M (see Exercise 17.23c). Several additional relations have to be used for bookkeeping purposes. For instance, when stepping through the tuples in $succ_3$, one must keep track of the last tuple that has been processed.

We next outline the construction for (β). One must simulate the computation of M starting from the initial configuration represented in R_M. To construct a new configuration from the current one, one must simulate a move of M. This is repeated until M reaches

R_M

a	a	a	a	a	a	a	a	P	s
a	a	a	a	a	a	a	b	[	0
a	a	a	a	a	a	a	c	1	0
a	a	a	a	a	a	a	d	#	0
a	a	a	a	a	a	b	a	0	0
a	a	a	a	a	a	b	b	#	0
a	a	a	a	a	a	b	c	1	0
a	a	a	a	a	a	b	d	0	0
a	a	a	a	a	a	c	a	]	0
a	a	a	a	a	a	c	b	[	0
a	a	a	a	a	a	c	c	1	0
a	a	a	a	a	a	c	d	#	0
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
a	a	a	b	a	a	a	a	0	0
a	a	a	b	a	a	a	b	[	r
a	a	a	b	a	a	a	c	1	0
a	a	a	b	a	a	a	d	#	0
a	a	a	b	a	a	b	a	0	0
a	a	a	b	a	a	b	b	#	0
a	a	a	b	a	a	b	c	1	0
a	a	a	b	a	a	b	d	0	0
a	a	a	b	a	a	c	a	]	0
a	a	a	b	a	a	c	b	[	0
a	a	a	b	a	a	c	c	1	0
a	a	a	b	a	a	c	d	#	0
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

Figure 17.7: Coding of part of the (first two) configurations

a final state (accepting or rejecting), which, as we assumed earlier, happens after at most n_c^k steps. The iteration can be performed using the fixpoint operator in $CALC + \mu^+$. Each step consists of defining the new configuration from the current one, timestamping it, and adding it to R_M. This can be done with a CALC formula. For instance, suppose the current state of M is q, the content of the current cell is 0, and the corresponding move of M is to change 0 to 1, move right, and change states from q to r. Suppose also that

constant_coding

a	a	0
b	a	1
c	a	1
c	b	0
d	a	1
d	b	1

Figure 17.8: The relation *constant_coding* corresponding to *a,b,c,d*

- $\vec{t}$ is the timestamp (in the example this is a 4-tuple) identifying the current configuration,
- R_M contains the tuple $\langle \vec{t}, \vec{j}, 0, q \rangle$, where $\vec{j}$ specifies a tape cell (in the example again with a 4-tuple), and
- $\vec{t}'$ is the next timestamp and $\vec{j}'$ the next cell [i.e., $succ_k(\vec{t}, \vec{t}')$ and $succ_k(\vec{j}, \vec{j}')$].

The tuples describing the new configuration of M are

(a) $\langle \vec{t}', \vec{i}, x, y \rangle$ if $\vec{i} \neq \vec{j}, \vec{i} \neq \vec{j}'$ and $\langle \vec{t}, \vec{i}, x, y \rangle \in R_M$;
(b) $\langle \vec{t}', \vec{j}, 1, 0 \rangle$;
(c) $\langle \vec{t}', \vec{j}', x, r \rangle$ if $\langle \vec{t}, \vec{j}', x, 0 \rangle \in R_M$.

In other words, (a) says that the cells other than the j^{th} cell and the next cell remain unchanged; (b) says that the content of cell j changes from 0 to 1, and the head no longer points to the j^{th} cell; finally, (c) says that the head points to the right adjacent cell, the new state is r, and the content of that cell is unchanged. Clearly, (a) through (c) can be expressed by a CALC formula (Exercise 17.23d). One such formula is needed for each move of M, and the formula corresponding to the finite set of possible moves is obtained by their disjunction.

We have outlined queries that realize (α) and (β) (i.e., perform the encoding needed to run M and then simulate the run of M). Using these *fixpoint* queries and their analog for phase (γ), it is now easy to construct the *fixpoint* query q_M that carries out the complete computation of q. This completes the proof of (a).

The construction for (b) is similar. The difference lies in the fact that a *while* computation need not be inflationary, unlike *fixpoint* computations. This simplifies the simulation. For instance, only the tuples corresponding to the current configuration of M are kept in R_M (Exercise 17.24). ■

Although PTIME is considered synonymous with tractability in many circumstances, complexity classes lower than PTIME are most useful in practice in the context of potentially large databases. There are numerous results that extend the logical characterization of QPTIME to lower complexity classes for ordered databases. For instance, by limiting the fixpoint operator in *fixpoint* to simpler operators based on various forms of transitive

closure, one can obtain languages expressing QLOGSPACE and QNLOGSPACE on ordered databases.

Theorem 17.4.2 implies that the presence of order results in increased expressive power for the *fixpoint* and *while* queries. For these languages, this is easily seen (for instance, *even* can be expressed by *fixpoint* when an order is provided). For weaker languages, the impact of order may be harder to see. For instance, it is not obvious whether the presence of order results in increased expressive power for CALC. The query *even* is of no immediate help, because it cannot be expressed by CALC even in the presence of order (Exercise 17.8). However, a more complicated query based on *even* can be used to show that CALC does indeed become more expressive with an order (Exercise 17.27). Because the CALC queries on ordered instances remain in AC_0, this shows in particular that there are queries in AC_0 that CALC cannot express.

From Chaos to Order: A Normal Form for *While*

We next discuss informally a normal form for the *while* queries that provides a bridge between computations without order and computations with order. This helps us understand the impact of order and the cost of computation without order.

The normal form says, intuitively, that each *while* query on an unordered instance can be reduced to a *while* query over an *ordered* instance via a *fixpoint* query. More precisely, a *while* program in the normal form consists of two phases. The first is a *fixpoint* query that performs an analysis of the input. It computes an equivalence relation on tuples that is a congruence with respect to the rest of the computation, in that equivalent tuples are treated identically throughout the computation. Thus each equivalence class is treated as an indivisible block of tuples that is never split later in the computation. The *fixpoint* query outputs the equivalence classes in some order, so that each class can be thought of abstractly as an integer. The second phase consists of a *while* query that can be viewed as computing on an *ordered* database obtained by replacing each equivalence class produced in the analysis phase by its corresponding integer.

The normal form also allows the clarification of the relationship between *fixpoint* and *while*. Because on ordered databases the two languages express QPTIME and QPSPACE, respectively, the languages are equivalent on ordered databases iff PTIME = PSPACE. What about the relationship of these languages without the order assumption? It turns out that the normal form can be used to extend this result to the general case when no order is present.

We do not describe the normal form in detail, but we provide some intuition on how a query on an unordered database reduces to a query on an ordered database.

Consider a *while* program q and a particular instance. There are only finitely many CALC queries that are used in q, and the number of their variables is bounded by some integer, say k. To simplify, assume that the input instance consists of a single relation I of arity k and that all relations used in q also have arity k. We can further assume that all queries used in assignment statements are either conjunctive queries or the single algebra operations $-, \cup$, and that no relation name occurs twice in a query. For a query φ in q, $\varphi(R_1, \ldots, R_n)$ indicates that $R_1, \ldots, R_n$ are the relation names occurring in φ.

Consider the set J of k-tuples formed with the constants from I. First we can distinguish between tuples based on their presence in (or absence from) I. This yields a first par-

tition of J. Now using the conjunctive queries occurring in q, we can iteratively refine this partition in the following way: If for some conjunctive query $\varphi(R_1, \ldots, R_n)$ occurring in q and some blocks $B_1, \ldots, B_n$ of the current partition $\varphi(B_1, \ldots, B_n)$ and $\neg\varphi(B_1, \ldots, B_n)$ have nonempty intersection with some block B' of the current partition, we refine the partition by splitting the block B' into $B' \cap \varphi(B_1, \ldots, B_n)$ and $B' \cap \neg\varphi(B_1, \ldots, B_n)$. This is repeated until no further refinement occurs, yielding a final partition of J. Furthermore, the blocks can be numbered as they are produced, which provides an ordering $\langle J_1, \ldots, J_m \rangle$ of the blocks of the partition. The entire computation can be performed by a *fixpoint* query constructed from q.

It is important to note that two tuples u, v in one block of the final partition cannot be separated by the computation of q on input I (i.e., at each step of this computation, each relation either contains both u and v or none). In other words, each relation contains a union of blocks of the final partition. Then one can reduce the original computation to an abstract computation q' on the integers by replacing the i^{th} block of the partition by integer i. Thus the original query q can be rewritten as the composition of a *fixpoint* query f followed by a *while* query q' that essentially operates on an ordered input.

Using this normal form, one can show the following:

THEOREM 17.4.3 *While* = *fixpoint* iff PTIME = PSPACE.

Crux The "only if" part follows from Theorem 17.4.2. The normal form is used for the "if" part as follows. Suppose PTIME = PSPACE. Then QPTIME = QPSPACE. Let q be a *while* query. By the normal form, $q = fq'$, where f is a *fixpoint* query and q' is a *while* query whose computation is isomorphic to that of a *while* query on an ordered domain. Because q' is in PSPACE and PSPACE = PTIME, q' is in PTIME. By Theorem 17.4.2(a), there exists a *fixpoint* query f' equivalent to q' on the ordered domain. Thus q is equivalent to ff' and is a *fixpoint* query. ■

An Alternative to Order: Nondeterminism

Results such as Theorem 17.4.2 show that the presence of order can solve some of the problems of expressiveness of query languages. This can be interpreted as a trade-off between expressiveness and the data independence provided by the abstract interface to the database system. We conclude this section by considering an alternative to order for increasing expressive power. It is based on the use of nondeterminism.

We will use the following terminology. A *deterministic* query is a classical query that always produces at most one output for each input instance. A *nondeterministic* query is a query that may have more than one possible outcome on a given input instance. Generally we assume that all possible outcomes are acceptable as answers to the query. For example, the query "Find *one* cinema showing *Casablanca*" is nondeterministic.

Consider again the query *even*, which is not expressible by *fixpoint* or *while*. The query *even* is easily computed by *fixpoint* in the presence of order (see Exercise 17.25). Another way to circumvent the difficulty of computing *even* is to relax the *determinism* of the query language. If one could choose, whenever desired, an *arbitrary* element from the set, this would provide another way of enumerating the elements of the set and computing *even*.

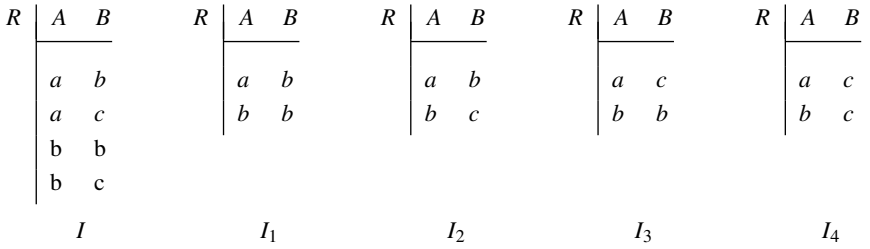


Figure 17.9: An application of *witness*

The drawback is that, with such a nondeterministic construct in the language, determinism of queries can no longer be guaranteed.

The trade-offs based on order and nondeterminism are not unrelated, as it may seem at first. Suppose that an order is given. As argued earlier, this comes down to suspending the data independence principle and accessing the internal representation. In general, the computation may depend on the particular order accessed. Then at the conceptual level, where the order is not visible, the mapping defined by the query appears as nondeterministic. Different outcomes are possible for the same conceptual-level view of the input. Thus the trade-offs based on order and on relaxing determinism are intimately connected.

To illustrate this, we exhibit nondeterministic versions of the $while^{(+)}$ and CALC+$\mu^{(+)}$ queries. In both cases we obtain exactly the (deterministic and nondeterministic) queries computable in polynomial space (time). Analogous results can be shown for lower complexity classes of queries.

Consider first the algebraic setting. We introduce a new operator called *witness* that provides the nondeterminism. To illustrate the use of this operator, consider the relation I in Fig. 17.9. An application of $witness_B$ to I may lead to several results [i.e., $witness_B(I)$ is either I_1, I_2, I_3 or I_4]. Intuitively, for each x occurring in the A column, $witness_B$ selects some tuple $\langle x, y\rangle$ in I, thus choosing nondeterministically a B value y for x. More generally, for each relation J over some schema $U = XY$, $X \cap Y = \emptyset$, $witness_Y(I)$ selects one tuple $\langle \vec{x}, \vec{y}\rangle$ for each $\langle \vec{x}\rangle$ occurring in $\Pi_X(J)$. Observe that from this definition, $witness_U(J)$ selects one tuple in J (if any).

It is also possible to describe the semantics of the *witness* operator using functional dependencies: For each instance J over some schema XY, $X \cap Y = \emptyset$, a possible result of $witness_Y(J)$ is a maximal subinstance J' of J satisfying $X \to Y$ (i.e., such that the attributes in X form a key).

The *witness* operator provides, more generally, a uniform way of obtaining nondeterministic counterparts for traditional deterministic languages.

The extension of $while^{(+)}$ with *witness* is denoted by $while^{(+)}+W$. Following is a useful example that shows that an arbitrary order can be constructed using the *witness* operator.

Example 17.4.4 Consider an input instance over some unary relation schema R. The following *while*+W query defines *all possible* successor relations on the constants from

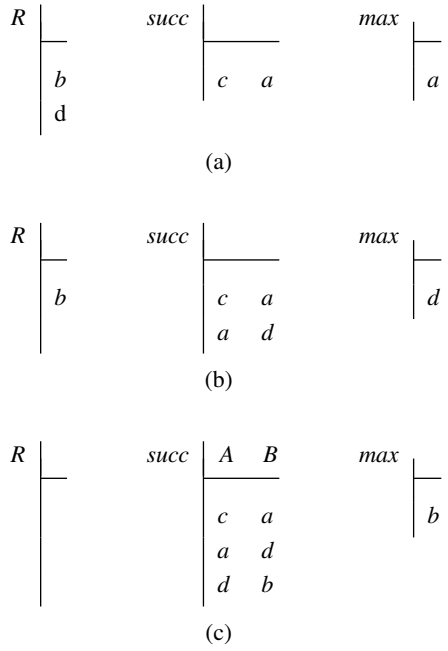


Figure 17.10: Some steps in the computation of an ordering

the input (i.e., each run constructs some ordering of the constants from the input; we use the unnamed perspective):

$succ := witness_{12}(\sigma_{1\neq 2}(R \times R))$;
$max := \pi_2(succ)$; $R := R - (\pi_1(succ) \cup \pi_2(succ))$;
while change do
  begin
  $succ := succ \cup witness_{12}(max \times R)$;
  $max := \pi_2(succ) - \pi_1(succ)$;
  $R := R - max$
  end

The result is constructed in a binary relation *succ*. A unary relation *max* contains the current maximum element in *succ*. Some steps of a possible computation on input $R = \{a, b, c, d\}$ are shown in Fig. 17.10: (a) shows the state before the loop is first entered, (b) the state after the first execution of the loop, and (c) the final state. Note that the output is empty if R contains fewer than two constants. It is of interest to observe that the program uses only the ability of *witness* to pick an arbitrary tuple from a relation.

This query can also be expressed in $while^+ + W$. (See Exercise 17.31.)

To continue with the nondeterministic languages, we next consider the language

CALC+$\mu^{(+)}$. The nondeterminism is again provided by a logical operator called *witness*[4] and denoted W. Suppose $\varphi(\vec{x}, \vec{y})$ is a formula with free variables $\vec{x}, \vec{y}$. Intuitively, $W\vec{y}\varphi(\vec{x}, \vec{y})$ indicates that one "witness" $\vec{y}_x$ is chosen for each $\vec{x}$ satisfying $\exists\ \vec{y}\ \varphi(\vec{x}, \vec{y})$. For example, if R consists of the relation I in Fig. 17.9, the formula $W_y R(x, y)$ defines the possible answers I_1, I_2, I_3, I_4 in the same figure. [Thus $W_y R(x, y)$ is equivalent to $witness_B(R)$.] More precisely, for each formula $\varphi(\vec{x}, \vec{y})$ (where $\vec{x}$ and $\vec{y}$ are vectors of the variables that are free in φ), $W\vec{y}\varphi(\vec{x}, \vec{y})$ is a formula (where the $\vec{y}$ remain free) defining the *set* of relations $\mathbf{I}$ such that for some $\mathbf{J}$ defined by φ: $\mathbf{I} \subseteq \mathbf{J}$; and for each $\vec{x}$ for which $\langle \vec{x}, \vec{y} \rangle$ is in $\mathbf{J}$ for some $\vec{y}$, there exists a *unique* $\vec{y}_x$ such that $\langle \vec{x}, \vec{y}_x \rangle$ is in $\mathbf{I}$.

The extension of CALC+$\mu^{(+)}$ with the *witness* operator is denoted by CALC+$\mu^{(+)}$+W. Following is a useful example that shows that an arbitrary order can be constructed using CALC+μ^{+}+W.

Example 17.4.5 Consider the (unary) relation schema R of Example 17.4.4. The following CALC+μ^{+}+W query defines, on each instance I of R, *all possible* successor relations on the constants in I. (The output is empty if I contains fewer than two constants.) The query uses a binary relation schema *succ*, which is used to construct the successor relation iteratively. The query is $\mu^{+}_{succ}(\varphi(succ))(x, y)$, where $\varphi = \varphi_1 \vee \varphi_2$ and

$$\varphi_1(x, y) = \neg\exists x \exists y(succ(xy)) \ \wedge\ Wxy(R(x) \ \wedge\ R(y) \ \wedge\ x \neq y),$$
$$\varphi_2(x, y) = Wy(R(y) \wedge \neg\exists z(succ(yz) \vee succ(zy))) \wedge \exists z(succ(zx)) \wedge \neg\exists z(succ(xz)).$$

The formula φ_1 initializes the iteration when *succ* is empty; φ_2 adds to *succ* a tuple $\langle x, y \rangle$, where y is an arbitrarily chosen element of $I(R)$ not yet in *succ* and x is the current maximum element in *succ*.

The ability of *while*$^{+}$+W and CALC+μ^{+}+W to define nondeterministically a successor relation on the constants suggests that the impact of nondeterminism on expressive power is similar to that of order. This is confirmed by the following result.

Theorem 17.4.6 The set of deterministic queries that are expressed by *while*$^{+}$+W or CALC+μ^{+}+W is QPTIME.

Proof It is easy to verify that each deterministic query expressed by *while*$^{+}$ + W is in QPTIME. Conversely, let q be a query in QPTIME. By Theorem 17.4.2, there exists a *while*$^{+}$ query w that expresses q if a successor relation *succ* on the constants is given. Then the *while*$^{+}$+W query expressing q consists of the following:

(i) construct a successor relation *succ* on the constants, as in Example 17.4.5;

(ii) apply query w to the input instance together with *succ*. ■

---

[4] The *witness* operator is related to Hilbert's ε-symbol [Lei69], but its semantics is different. In particular, the ε-symbol does not yield nondeterminism.

An analogous result holds for *while*+W and CALC+μ+W. Specifically, the set of deterministic queries expressible by these languages is precisely QPSPACE.

Note that Theorem 17.4.6 does *not* provide a *language* that expresses precisely QPTIME, because nondeterministic queries can also be expressed and it is undecidable if a *while*$^+$+W or CALC+μ^++W query defines a deterministic query (Exercise 17.32). Instead the result shows the power of nondeterministic constructs and so points to a trade-off between expressive power and determinism.

Bibliographic Notes

The sequential data complexity of CALC was investigated by Vardi [Var82a], who showed that CALC is included in LOGSPACE. The parallel complexity of CALC, specifically the connection with AC_0, was studied by Immerman [Imm87a]. In [DV91], a database model for parallel computation is defined, and CALC is shown to coincide *exactly* with its restriction to constant time and polynomial size. This differs from AC_0 in that the match is precise. Intuitively, this is due to the fact that the model in [DV91] is generic and does not assume an ordered encoding of the input.

The first results on the expressiveness and complexity of *fixpoint* and *while* were obtained by Chandra and Harel, Vardi, and Immerman. In [CH80b] it is shown by a direct proof that *fixpoint* cannot express *even*. The result is extended to *while* in [Cha81a]. The fundamental result that *fixpoint* expresses QPTIME on ordered instances was obtained independently by Immerman [Imm86] and Vardi [Var82a]. The fact that *while* on ordered instances expresses QPSPACE is shown in [Var82a].

Languages expressing complexity classes of queries below QPTIME are investigated in [Imm87b]. They are based on augmenting CALC with operators providing limited recursion, such as various forms of transitive closure. The classes of queries expressed by the resulting languages on ordered databases include deterministic logspace, denoted LOGSPACE, nondeterministic logspace, denoted NLOGSPACE, and symmetric logspace, denoted SLOGSPACE.

There has been a long quest for a language expressing precisely QPTIME on arbitrary (unordered) databases. The problem is formalized in a general setting in [Gur88], where it is also conjectured that no such language exists. The issue is further investigated in [Daw93], where, in particular, it is shown that there exists a language for QPTIME iff there exists some problem complete in P via an extended kind of first-order reductions. To date, the problem of the existence of a language for QPTIME remains open.

In the absence of a language for QPTIME, there have been several proposals to extend the *fixpoint* queries to capture more of QPTIME. Recall that queries involving counting (such as *even*) are not in *fixpoint*. Therefore it is natural to consider extensions of *fixpoint* with counting constructs. An early proposal by Chandra [Cha81a] is to add a *bounded looping* construct of the form *For* $|R|$ *do*, which iterates the body of the loop $|R|$ times. Clearly, this construct allows us to express *even*. However, it has been shown that bounded looping is not sufficient to yield all of QPTIME, because tests $|R_1| = |R_2|$ cannot be expressed (see [Cha88]). More recently, extensions of *fixpoint* with counting constructs have been considered and studied in [CFI89, GO93]. They allow access to the cardinality of relations as well as limited integer manipulation. These languages are more powerful than *fixpoint*

but, as shown in [CFI89], still fall short of expressing all of QPTIME. Other results of this flavor are proven in [Daw93, Hel92]. They show that extending *fixpoint* with a finite set of polynomial-time computable constructs of certain forms (generalized quantifiers acting much like oracles) cannot yield a language expressing exactly QPTIME (see Exercise 17.35 for a simplified version of this result).

The normal form for *while* was proven in [AV91b, AV94]. It was also shown there, using the normal form, that *fixpoint* and *while* are equivalent iff PTIME = PSPACE. The cost of computing without an order is also investigated in [AV91b, AV94]. This is formalized using an alternative model of computation called *generic machine* (GM). Unlike Turing machines, GMs do not require an ordered encoding of the input and use only the information provided by the input instance. Based on GM, *generic* complexity classes of queries are defined. For example, GEN-PTIME and GEN-PSPACE are obtained by taking polynomial time and space restrictions of GM. As a typical result, it is shown that *even* is not in GEN-PSPACE, which captures the intuition that this query is hard to compute without order. Another more restricted device, also operating without encodings, is the *relational machine*, also considered in [AV91b, AV94]. There is a close match between complexity classes defined using this device, called *relational complexity classes*, and various languages. For example, relational polynomial time coincides with *fixpoint* and relational polynomial space with *while*. Further connections between languages and relational complexity classes are shown in [AVV92].

Nondeterministic languages and their expressive power are investigated in [ASV90, AV91a, AV91c]. The languages include nondeterministic extensions of CALC+μ^+ and CALC+μ and of rule-based languages such as datalog$^\neg$. Strong connections between these languages are shown (see Exercise 17.33). Nondeterministic languages that can express all the QPTIME queries are exhibited.

A construct related to the witness operator described in this chapter is the *choice* operator, first presented in [KN88]. This construct has been included in the language LDL, an implementation of datalog$^\neg$ [NT89] (see also Chapter 15). Variations of the choice operator, and its connection with stable models of datalog$^\neg$ programs, are further studied in [SZ90, GPSZ91]. The expressive power of the choice operator in the context of datalog is investigated in [CGP93] (see Exercise 17.34).

The Ehrenfeucht-Fraissé games are due to Ehrenfeucht [Ehr61] and Fraissé [Fra54]. Since their work, extensions of the games have been proposed and related to various languages such as datalog [LM89], fragments of infinitary logic [KV90c], *fixpoint* queries, and second-order logic [Fag75, AF90, dR87]. In [Imm82, CFI89], games are used to prove lower bounds on the number of variables needed to express certain graph properties. Typically, in the extensions of Ehrenfeucht-Fraissé games, choosing a constant in an instance is thought of as placing a pebble over that constant (the games are often referred to as *pebble games*). Like the Ehrenfeucht-Fraissé games, these are two-player games in which one player attempts to prove that the instances are not the same and the other attempts to prove the contrary by placing the pebbles such that the corresponding subinstances are isomorphic. The games differ in the rules for taking turns among players and instances, the number of pebbles placed in one move, whether the pebbles are colored, etc. In games corresponding to languages with recursion, players have more than one chance for achieving

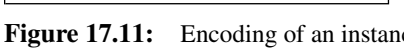


Figure 17.11: Encoding of an instance and tuple

their objective by removing some of the pebbles and restarting the game. Our presentation of Ehrenfeucht-Fraissé games was inspired by Kolaitis's excellent lecture notes [Kol83].

The study of 0-1 laws was initiated by Fagin and Glebskiĭ. The 0-1 law for CALC was proven in [Fag72, Fag76] and independently by Glebskiĭ et al. [GKLT69]. The 0-1 law for *fixpoint* was shown by Blass, Gurevich, and Kozen [BGK85] and Talanov and Knyazev [TK84]. This was extended to *while* by Kolaitis and Vardi, who proved further extensions of 0-1 laws for certain fragments of second-order logic [KV87, KV90b] and for *infinitary logic* with finitely many variables [KV92], both of which subsume *while*. For instance, 0-1 laws were proven for existential second-order sentences $\exists Q_1 \ldots \exists Q_k \sigma$, where the Q_i are relation variables and σ is a CALC formula in prenex form, whose quantifier portion has one of the shapes $\exists^*\forall^*$ or $\exists^*\forall\exists^*$. It is known that arbitrary existential second-order sentences do not have a 0-1 law (see Exercise 17.21). Infinitary logic is an extension of CALC that allows infinite disjunctions and conjunctions. Kolaitis and Vardi proved that the language consisting of infinitary logic sentences that use only finitely many variables has a 0-1 law. Note that this language subsumes *while* (Exercise 17.22). Another aspect of 0-1 laws that has been studied involves the difficulty of deciding whether a sentence in a language that has a 0-1 law is almost surely true or whether it is almost surely false. For instance, Grandjean proved that the problem is PSPACE complete for CALC [Gra83]. The problem was investigated for other languages by Kolaitis and Vardi [KV87]. A comprehensive survey of 0-1 laws is provided by Compton [Com88].

Fagin [Fag93] presents a survey of finite model theory including 0-1 laws that inspired our presentation of this topic.

Exercises

Exercise 17.1 Consider the CALC query on a database schema with one binary relation G:

$$\varphi = \{x \mid \exists y \forall z (G(x, y) \wedge \neg G(z, x))\}.$$

Consider the instance $\mathbf{I}$ over G and tuple encoded on a Turing input tape, as shown in Fig. 17.11. Describe in detail the computation of the Turing machine M_φ, outlined in the proof of Theorem 17.1.1, on this input.

♠ **Exercise 17.2** Prove Theorem 17.1.2.

Exercise 17.3 Prove that $\equiv_r$ is an equivalence relation on instances.

Exercise 17.4 Outline the crux of Theorem 17.2.2 for the case where

$$\varphi = \forall x \, (\exists y \, (R(xy)) \vee \forall z \, (R(zx))).$$

(Note that the quantifier depth of φ is 2, so this case involves games with two moves.)

★ **Exercise 17.5** Provide a complete description of the winning strategy outlined in the crux of Proposition 17.2.3. *Hint:* For the game with r moves, choose cycles of size at least $r(2^{r+1} - 1)$.

Exercise 17.6 Extend Proposition 17.2.3 by showing that connectivity of graphs is not first-order definable even if an order $\leq$ on the constants is provided. More precisely, let $\mathbf{R}$ be the database schema consisting of two binary relations G and $\leq$. Let $\mathcal{I}_{\leq}$ be the family of instances $\mathbf{I}$ over $\mathbf{R}$ such that $\mathbf{I}(\leq)$ provides a total order on the constants of $\mathbf{I}(G)$. Outline a proof that there is no CALC sentence σ such that, for each $\mathbf{I} \in \mathcal{I}_{\leq}$,

$$\sigma(\mathbf{I}) \text{ is true iff } \mathbf{I}(\mathrm{G}) \text{ is a connected graph.}$$

♠ **Exercise 17.7** [Kol83] Use Ehrenfeucht-Fraissé games to show that the following properties of graphs are not first-order definable:

(i) the number of vertexes is even;

(ii) the graph is 2-colorable;

(iii) the graph is Eulerian (i.e., there exists a cycle that passes through each edge exactly once).

★ **Exercise 17.8** Show that the property that the number of elements in a unary relation is even is not first-order definable even if an order on the constants is provided.

The following two exercises lead to a proof of the converse of Theorem 17.2.2. It states that instances that are undistinguishable by CALC sentences of quantifier depth r are equivalent with respect to $\equiv_r$. This is shown by proving that each equivalence class of $\equiv_r$ is definable by a special CALC sentence of quantifier depth r, called the r-type of the equivalence class. Intuitively, the r-type sentence describes all patterns that can be detected by playing games of length r on pairs of instances in the equivalence class.

To define the r-types, one first defines formulas with m free variables, called (m, r)-types. An r-type is defined as a $(0, r)$-type. The set of (m, r)-types is defined by backward induction on m as follows.

An (r, r)-type consists of all satisfiable formulas φ with variables $x_1, \ldots, x_r$ such that φ is a conjunction of literals over R and for each $i_1, \ldots, i_k$, either $R(x_{i_1}, \ldots, x_{i_k})$ or $\neg R(x_{i_1}, \ldots, x_{i_k})$ is in φ. Suppose the set of $(m+1, r)$-types has been defined. Each set S of $(m+1, r)$-types gives rise to one (m, r)-type defined by

$$\bigvee \{ \exists x_{m+1}\, \varphi \mid \varphi \in S\} \vee \bigvee \{\forall x_{m+1}\, (\neg(\varphi)) \mid \varphi \notin S\}.$$

♠ **Exercise 17.9** [Kol83] Let r and m be positive integers such that $0 \leq m \leq r$. Prove that

(a) every (m, r)-type is a CALC formula with free variables $x_1, \ldots, x_m$ and quantifier depth $(r - m)$;

(b) there are only finitely many distinct (m, r)-types; and

(c) for every instance $\mathbf{I}$ and sequence $a_1, \ldots, a_m$ of constants in $\mathbf{I}$, there is exactly one (m, r)-type φ such that $\mathbf{I}$ satisfies $\varphi(a_1, \ldots, a_m)$.

♠ **Exercise 17.10** [Kol83] Prove that each equivalence class of $\equiv_r$ is definable by a CALC sentence of quantifier depth r. *Hint:* For a given equivalence class of $\equiv_r$, consider an instance in the class and the unique r-type satisfied by the instance.

Exercise 17.11 Complete the proof of Theorem 17.3.1; specifically show that

(a) *fixpoint* $\subseteq$ QPTIME and *while* $\subseteq$ QPSPACE, and

(b) *fixpoint* is complete in PTIME and *while* is complete in PSPACE.

Exercise 17.12 In the proof of Proposition 17.3.2, the case of assignments of the form $T := Q_1 \times Q_2$ was discussed. Describe the constructions needed for the other algebra operators. Point out where the assumption that the size of $\mathbf{I}$ is greater than N is used.

★ **Exercise 17.13** Prove that the *while* queries collapse to CALC on unary relation inputs. More precisely, let $\mathbf{R}$ be a database schema consisting of unary relations. Show that for each *while* query w on $\mathbf{R}$ there exists a CALC query φ equivalent to it. *Hint:* Use the same approach as in the proof of Proposition 17.3.2 to show that there is a constant bound on the length of runs of a given *while* program on unary inputs.

★ **Exercise 17.14** Describe how to generalize the proof of Proposition 17.3.2 so that it handles *while* queries that have constants. In particular, describe how the notion of hyperplanes needs to be generalized.

Exercise 17.15 Recall the technique of hyperplanes used in the proof of Proposition 17.3.2.

(a) Let $D \subseteq \mathbf{dom}$ be finite. For a relation schema R, the *cross-product instance* of R over D is $I^R_{\times D} = D \times \cdots \times D$ (arity of R times). The *cross-product instance* of database schema $\mathbf{R}$ over D is the instance $\mathbf{I}^{\mathbf{R}}_{\times D}$, where $\mathbf{I}^{\mathbf{R}}_{\times D}(R) = I^R_{\times D}$ for each $R \in \mathbf{R}$. Let P be a datalog$^\neg$ program with no constants, input schema $\mathbf{R}$, and output schema S with arity k. Prove that there is an $N > 0$ and a set E_P of equivalence relations over $[1, k]$ such that for each set $D \subseteq \mathbf{dom}$: if $|D| \geq N$ then

$$P(\mathbf{I}^{\mathbf{R}}_{\times D}) = \bigcup \{H_{\simeq}(D) \mid \simeq \in E_P\}.$$

(b) Prove (a) for datalog$^{\neg\neg}$ programs.

(c) Generalize your proofs to permit constants in P.

Exercise 17.16 In the proof of Lemma 17.3.6, prove more formally the bound on $\mu_n(\neg\sigma_k)$. Prove that its limit is 0 when n goes to ∞.

Exercise 17.17 Determine whether the following properties of graphs are almost surely true or whether they are almost surely false.

(a) Existence of a cycle of length three

(b) Connectivity

(c) Being a tree

Exercise 17.18 Prove that there is a finite number of equivalence classes of k-tuples induced by automorphisms of the Rado graph. *Hint:* Each class is completely characterized by the pattern of connection and equality among the coordinates of the k-tuple. To see this, show that for all tuples u and v satisfying this property, one can construct an automorphism ρ of the Rado graph such that $\rho(u) = v$. The automorphism is constructed using the extension axioms, similar to the proof of Lemma 17.3.5.

♠ **Exercise 17.19** Describe how to generalize the development of 0-1 laws for arbitrary input and for queries involving constants.

Exercise 17.20 Prove or disprove: The properties expressible in *fixpoint* are exactly the PTIME properties that have a 0-1 law.

Exercise 17.21 The language *existential second-order logic*, denoted ($\exists SO$), consists of sentences of the form $\exists Q_i \ldots \exists Q_k \sigma$, where Q_i are relations and σ is a first-order sentence using the relations Q_i (among others). Show that $\exists SO$ does not have a 0-1 law. *Hint:* Exhibit a property expressible in $\exists SO$ that is neither almost surely true nor almost surely false.

★ **Exercise 17.22** Infinitary logic with finitely many variables, denoted $L^{\omega}_{\infty\omega}$, is an extension of CALC that allows formulas with infinitely long conjunctions and disjunctions but using only a finite number of variables. Show that each *while* query can be expressed in $L^{\omega}_{\infty\omega}$. *Hint:* Start with a specific example, such as transitive closure.

Exercise 17.23 The following refer to the proof of Theorem 17.4.2.

(a) Describe a *fixpoint* query that, given a successor relation *succ* on constants, constructs a $2k$-ary successor relation $succ_k$ on k-tuples of constants, in the lexicographical order induced on k-tuples by *succ*.

(b) Show that the relation *constant_coding* can be defined from *succ* using a *fixpoint* query.

(c) Complete the details of the construction of R_M by a *fixpoint* query.

(d) Describe in detail the CALC formula corresponding to the move of M considered in the proof of Theorem 17.4.2.

(e) Describe in detail the CALC formula used to perform phase γ in the computation of q_M.

(f) Show where the proof of Theorem 17.4.2 breaks down if it is not assumed that the input instance is ordered.

Exercise 17.24 Spell out the differences in the proofs of (a) and (b) in Theorem 17.4.2.

Exercise 17.25 Write a *fixpoint* query that computes the parity query *even* on ordered databases.

Exercise 17.26 Consider queries of the form

Does the diameter of G have property P?

where P is an EXPTIME property of the integers (i.e., a property that can be checked, for integer n, in time exponential in $\log n$, or polynomial in n). Show that each query as above is a *fixpoint* query.

♠ **Exercise 17.27** [Gur] This exercise shows that there is a query expressible in CALC in the presence of order that is not expressible in CALC without order. Let $\mathbf{R} = \{D, S\}$, where D is unary and S is binary. Consider an instance $\mathbf{I}$ of $\mathbf{R}$. Suppose the second column of $\mathbf{I}(S)$ contains only constants from $\mathbf{I}(D)$. Then one can view each constant s in the first column of $\mathbf{I}(S)$ as denoting a subset of $\mathbf{I}(D)$, namely $\{x \mid S(s, x)\}$. Call an instance $\mathbf{I}$ of $\mathbf{R}$ *good* if for each subset of $\mathbf{I}(D)$, there exists a constant representing it. In other words, for each subset T of $\mathbf{I}(D)$, there exists a constant s such that

$$T = \{x \mid S(s, x)\}.$$

Consider the query q defined by $q(\mathbf{I}) = \mathbf{true}$ iff $\mathbf{I}$ is a good input and $|\mathbf{I}(D)|$ is even.

(a) Show that q is not expressible by CALC.

(b) Show that q is expressible on instances extended with an order relation $\leq$ on the constants.

(c) Note that in (b), an order is used instead of the usual successor relation on constants. Explain the difficulty of proving (b) if a successor relation is used instead of $\leq$.

Hint: For (a), use Ehrenfeucht-Fraissé games. Consider (b). To check that the input is good, check that (1) all singleton subsets of $\mathbf{I}(D)$ are represented, and (2) if T_1 and T_2 are represented, so is $T_1 \cup T_2$. To check evenness of $|\mathbf{I}(D)|$ on good inputs, define first from $\leq$ a successor relation $succ_D$ on the constants in $\mathbf{I}(D)$; then check that there exists a subset T of $\mathbf{I}(D)$ consisting of the even constants according to $succ_D$ and that the last element in $succ_D$ is in T.

♠ **Exercise 17.28** (Expression complexity [Var82a])

(a) Show that the *expression* complexity of CALC is within PSPACE. That is, consider a fixed instance $\mathbf{I}$ and tuple u, and a TM $M_{\mathbf{I},u}$ depending on $\mathbf{I}$ and u that, given as input some standard encoding of a query φ in CALC, decides if $u \in \varphi(\mathbf{I})$. Show that there is such a TM $M_{\mathbf{I},u}$ whose complexity is within PSPACE with respect to $|enc(\varphi)|$, when φ ranges over CALC.

(b) Prove that in terms of expression complexity, CALC is complete in PSPACE. *Hint:* Use a reduction to quantified propositional calculus (see Chapter 2 and [GJ79]).

(c) Let CALC$^-$ consist of the quantifier-free queries in CALC. Show that the expression complexity of CALC$^-$ is within LOGSPACE.

Exercise 17.29 Show that

(a) $Wx(WyR(x, y))$ is not equivalent[5] to $Wxy\varphi(x, y)$;

(b) $Wx(WyR(x, y))$ is not equivalent to $Wy(WxR(x, y))$.

Exercise 17.30 Write a CALC$+\mu^+ + W$ formula defining the query *even*.

Exercise 17.31 Express the query of Example 17.4.4 in *while*$^+ + W$.

♠ **Exercise 17.32** [ASV90] Show that it is undecidable whether a given CALC$+\mu^+ + W$ formula defines a deterministic query. *Hint:* Use the undecidability of satisfiability of CALC sentences.

♠ **Exercise 17.33** [AV91a, AV91c]. As seen, the *witness* operator can be used to obtain nondeterministic versions of *while*$^{(+)}$ and CALC$+\mu^{(+)}$. One can obtain nondeterministic versions of datalog$^{\neg(\neg)}$ as follows. The syntax is the same, except that heads of rules may contain several literals, and equality may be used in bodies of rules. The rules of the program are fired one rule at a time and one instantiation at a time. The nondeterminism is due to the choice of rule and instantiation used in each firing. The languages thus obtained are denoted *N-datalog*$^{\neg(\neg)}$.

(a) Prove that N-datalog$^{\neg\neg}$ is equivalent to CALC$+\mu + W$ and *while*$+W$ and expresses all nondeterministic queries computable in polynomial space.[6]

(b) Show that N-datalog$^{\neg}$ cannot compute the query $P - \pi_A(Q)$, where Q is of sort AB and P of sort A.

---

[5] Two formulas are *equivalent* iff they define the same set of relations for each given instance.

[6] This includes QPSPACE, the *deterministic* queries computable in polynomial space.

(c) Let N-datalog$^{\neg}\forall$ be the language obtained by extending N-datalog$^{\neg}$ with universal quantification in bodies of rules. For example, the program

$$answer(x) \leftarrow \forall y[P(x), \neg Q(x, y)]$$

computes the query $P - \pi_A(Q)$. Prove that N-datalog$^{\neg}\forall$ is equivalent to CALC$+\mu^+ + W$ and *while*$^+ + W$ and expresses all nondeterministic queries computable in polynomial time.

(d) Prove that N-datalog$^{\neg}$ and N-datalog$^{\neg}\forall$ are equivalent on ordered databases.

♠ **Exercise 17.34** (Dynamic choice operator [CGP93]) The following extension of datalog$^{\neq}$ with a variation of the *choice* operator (see Bibliographic Notes) is introduced in [CGP93]. Datalog$^{\neq}$ programs are extended by allowing atoms of the form *choice(X,Y)* in rules of bodies, where X and Y are disjoint sets of variables occurring in regular atoms of the rule. Several *choice* atoms can appear in one rule. The language obtained is called datalog$^{\neq}+$*choice*. The semantics is the following. The *choice* atoms render the immediate consequence operator of a datalog$^{\neq}+$*choice* program P nondeterministic. In each application of T_P, a subset of the applicable valuations is chosen so that for each rule containing an occurrence *choice(X,Y)*, the functional dependency $X \rightarrow Y$ holds. That is, one instantiation for the Y variables is chosen for each instantiation of the X variables. Moreover, the nondeterministic choices operated at each application of T_P for a given occurrence of a *choose* atom *extend* the choices made in previous applications of T_P for that atom. (Thus *choose* has a more global nature than the witness operator.) Although negation is not used in datalog$^{\neq}+$*choice*, it can be simulated. The following datalog$^{\neq}+$*choice* program computes in $\bar{P}$ the complement of a nonempty relation P with respect to a universal relation T of the same arity [CGP93]:

$$\begin{aligned}
TAG(X, 0) &\leftarrow P(X)\\
TAG(X, 1) &\leftarrow T(X), COMP(Y, 0)\\
COMP(X, I) &\leftarrow TAG(X, I), choose(X, I)\\
\bar{P}(X) &\leftarrow COMP(X, 1)
\end{aligned}$$

The role of *choose* in the preceding program is simple. When first applied, it associates with each X in P the tag $I = 0$. At the second application, it chooses a tag of 0 or 1 for all tuples in T. However, tuples in P have already been tagged by 0 in the previous application of *choose*, so the tuples tagged by 1 are precisely those in the complement.

(a) Exhibit a datalog$^{\neq}+$*choice* program that, given as input a unary relation P, defines nondeterministically the successor relations on the constants in P.

(b) Show that every N-datalog$^{\neg}$ query is expressible in datalog$^{\neq}+$*choice* (see Exercise 17.33).

(c) Prove that datalog$^{\neq}+$*choice* expresses exactly the nondeterministic queries computable in polynomial time.

♠ **Exercise 17.35** [Daw93, Hel92] As shown in this chapter, the *fixpoint* queries fall short of expressing all of QPTIME. For example, they cannot express *even*. A natural idea is to enrich the *fixpoint* queries with additional constructs in the hope of obtaining a language expressing exactly QPTIME. This exercise explores one (unsuccessful) possibility, which consists of adding some finite set of PTIME oracles to the *fixpoint* queries.

A *property* of instances over some database schema $\mathbf{R}$ is a subset of $inst(\mathbf{R})$ closed under isomorphisms of **dom**. Let $\mathbf{Q}$ be a finite set of properties, each of which can be checked in PTIME. Let $while^+(\mathbf{Q})$ be the extension of $while^+$ allowing loops of the form $while\ q(R_1, \ldots, R_n)\ do$, where $q \in \mathbf{Q}$ and $R_1, \ldots, R_n$ are relation variables compatible with the schema of q. Intuitively, this allows us to ask whether $R_1, \ldots, R_n$ have property q. Clearly, $while^+(\mathbf{Q})$ generally has more power than $while^+$. For example, the query *even* is trivially expressible in $while^+(\{even\})$. One might wonder if there is choice of $\mathbf{Q}$ such that $while^+(\mathbf{Q})$ expresses exactly QPTIME.

(a) Show that for every finite set $\mathbf{Q}$ of PTIME properties, there exists a single PTIME property q such that $while^+(\mathbf{Q}) \equiv while^+(\{q\})$.

(b) Let $while_1^+(\{q\})$ denote all $while^+(\{q\})$ programs whose input is one unary relation. Let PTIME$[k]$ denote the set of properties whose time complexity is bounded by some polynomial of degree k. Show that, for each PTIME property q, the properties of unary relations definable in $while_1^+(\{q\})$ are in PTIME$[k]$ for some k depending only on q. *Hint:* Show that for each $while_1^+(\{q\})$ program there exist $N > 0$ and properties $q_1, \ldots, q_m$ of integers where each $q_i(n)$ can be checked in time polynomial in n, such that the program is equivalent to a Boolean combination of tests $n \geq j$, $n = j$, $q_i(n)$, where n is the size of the input, $0 \leq j \leq N$ and $1 \leq i \leq m$. Use the hyperplane technique developed in the proof of Proposition 17.3.2.

(c) Prove that there is no finite set $\mathbf{Q}$ of PTIME properties such that $while^+(\mathbf{Q})$ expresses QPTIME. *Hint:* Use (a), (b), and the fact that PTIME$[k] \subset$ PTIME by the time hierarchy theorem.

18 Highly Expressive Languages

Alice: *I still cannot check if I have an even number of shoes.*
Riccardo: *This will not stand!*
Sergio: *We now provide languages that do just that.*
Vittorio: *They can also express any query you can think of.*

In previous chapters, we studied a number of powerful query languages, such as the *fixpoint* and *while* queries. Nonetheless, there are queries that these languages cannot express. As pointed out in the introduction to Chapter 14, *fixpoint* lies within PTIME, and *while* within PSPACE. The complexity bound implies that there are queries, of complexity higher than PSPACE, that are not expressible in the languages considered so far. Moreover, we showed simple, specific queries that are not in *fixpoint* or *while*, such as the query *even*.

In this chapter, we exhibit several powerful languages that have no complexity bound on the queries they can express. We build up toward languages that are complete (i.e., they express all queries). Recall that the notion of query was made formal in Chapter 16. Basically, a query is a mapping from instances of a fixed input schema to instances of a fixed answer schema that is computable and generic. Recall that, as a consequence, answers to queries contain only constants from the input (except possibly for some fixed, finite set of new constants).

We begin with a language that extends *while* by providing arbitrary computing power *outside* the database; this yields a language denoted $while_N$, in the style of embedded relational languages like C+SQL. This would seem to provide the simplest cure for the computational limitations of the languages exhibited so far. There is no complexity bound on the queries $while_N$ can express. Surprisingly, we show that, nonetheless, $while_N$ is not complete. In fact, $while_N$ cannot express certain simple queries, including the infamous query *even*. Intuitively, $while_N$ is not complete because the external computation has limited interaction with the database. Complete languages are obtained by overcoming this limitation. Specifically, we present two ways to do this: (1) by extending *while* with the ability to create new values in the course of the computation, and (2) by extending *while* with an untyped version of relational algebra that allows relations of variable arity.

For conciseness, in this chapter we do not pursue the simultaneous development of languages in the three paradigms—algebraic, logic, and deductive. Instead we choose to focus on the algebraic paradigm. However, analogous languages could be developed in the other paradigms (see Exercise 18.22).

18.1 $While_N$—*while* with Arithmetic

The language *while* is the most powerful of the languages considered so far. We have seen that it lies within PSPACE. Thus it does not have full computing power. Clearly, a complete language must provide such power. In this section, we consider an extension of *while* that does provide full computing power *outside* the database. Nonetheless, we will show that the resulting language is *not* complete; it is important to understand why this is so before considering more exotic ways of augmenting languages.

The extension of *while* that we consider allows us to perform, outside the database, arbitrary computations on the integers. Specifically, the following are added to the *while* language:

(i) integer variables, denoted $i, j, k, \ldots$;

(ii) the integer constant 0 (zero);

(iii) instructions of the form *increment(i), decrement(i)*, where i is an integer variable;

(iv) conditional statements of the form *if* $i = 0$ then s *else* s', where i is an integer variable and s, s' are statements in the language;

(v) loops of the form *while* $i > 0$ *do* s, where i is an integer variable and s a program.

The semantics is straightforward. All integer variables are initialized to zero. The semantics of the *while change* construct is not affected by the integer variables (i.e., the loop is executed as long as there is a change in the content of a *relational* variable). The resulting language is denoted by $while_N$.

Because the language $while_N$ can simulate an arbitrary number of counters, it is computationally complete on the integers (see Chapter 2). More precisely, the following holds:

Fact For every computable function $f(i_1, \ldots, i_k)$ on integers, there exists a $while_N$ program w_f that computes $f(i_1, \ldots, i_k)$ for every integer initialization of $i_1, \ldots, i_k$. In particular, w_f stops on input $i_1, \ldots, i_k$ iff f is defined on $(i_1, \ldots, i_k)$.

In view of this fact, one can use in $while_N$ programs, whenever convenient, statements of the form $n := f(i_1, \ldots, i_k)$, where $n, i_1, \ldots, i_k$ are integer variables and f is a computable function on the integers. This is used in the following example.

EXAMPLE 18.1.1 Let G be a binary relation with attributes AB. Consider the query on the graph G:

$$square(G) = \emptyset \text{ if the diameter of } G \text{ is a perfect square, and } G \text{ otherwise.}$$

The following $while_N$ program computes $square(G)$ (the output relation is *answer*; it is assumed that $G \neq \emptyset$):

$i := 0; T := G;$

while change do
  begin
  $T := T \cup \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
  increment(i);
  end;
$j := f(i)$;
answer $:= G$;
if $j > 0$ *then answer* $:= \emptyset$.

where f is the function such that $f(x) = 1$ if x is a perfect square and $f(x) = 0$ otherwise. (Clearly, f is computable.) Note that, after execution of the while loop, the value of i is the diameter of G.

It turns out that the preceding program can been expressed in *while* alone, and even *fixpoint*, without the need for arithmetic (see Exercise 18.2). However, this is clearly not the case in general. For instance, consider the $while_N$ program obtained by replacing f in the preceding program by some arbitrary computable function.

Despite its considerable power, $while_N$ cannot express certain simple queries, such as *even*. There are several ways to show this, just as we did for *while*. Recall that, in Chapter 17, it was shown that *while* has a 0-1 law. It turns out that $while_N$ also has a 0-1 law, although proving this is beyond the scope of this book. Thus there are many queries, including *even*, that $while_N$ cannot express. One can also give a direct proof that *even* cannot be expressed by $while_N$ by extending straightforwardly the hyperplane technique used in the direct proof that *while* cannot express *even* (Proposition 17.3.2, see Exercise 18.3).

As in the case of other languages we considered, order has a significant impact on the expressiveness of $while_N$. Indeed, $while_N$ is complete on ordered databases.

THEOREM 18.1.2 The language $while_N$ expresses all queries on ordered databases.

Crux Let q be a query on an ordered database with schema $\mathbf{R}$. Let $\mathbf{I}$ denote an input instance over $\mathbf{R}$ and α the enumeration of constants in $\mathbf{I}$ given by the relation *succ*. By the definition of query, there exists a Turing machine M_q that, given as input $enc_\alpha(\mathbf{I})$, produces as output $enc_\alpha(q(\mathbf{I}))$ (whenever q is defined on $\mathbf{I}$). Because $while_N$ manipulates integers, we wish to encode $\mathbf{I}$ as an integer rather than a Turing machine tape. This can be done easily because each word over some finite alphabet with k symbols (with some arbitrary order among the symbols) can be viewed as an integer in base k. For any instance $\mathbf{J}$, let $enc^*_\alpha(\mathbf{J})$ denote the integer encoding of $\mathbf{J}$ obtained by viewing $enc_\alpha(\mathbf{J})$ as an integer. It is easy to see that there is a computable function f_q on the integers such that $f_q(enc^*_\alpha(\mathbf{I})) = enc^*_\alpha(q(\mathbf{I}))$ whenever q is defined on $\mathbf{I}$. Furthermore, because $while_N$ can express any computable function over the integers (see the preceding Fact), there exists a $while_N$ program $w_{f_q}(i)$ that computes f_q. It is left to show that $while_N$ can compute $enc^*_\alpha(\mathbf{I})$ and can decode $q(\mathbf{I})$ from $enc^*_\alpha(q(\mathbf{I}))$. Recall that, in the proof of Theorem 17.4.2, it was shown that *while* can compute a relational representation of $enc_\alpha(\mathbf{I})$ and, conversely, it can decode $q(\mathbf{I})$ from the representation of $enc_\alpha(q(\mathbf{I}))$. A slight modification of that construction can be used to

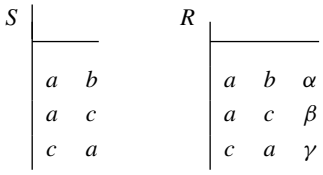


Figure 18.1: An application of *new*

show that $while_N$ can compute the desired integer encoding and decoding. Thus a $while_N$ program computes q in three phases:

1. compute $enc^*_\alpha(\mathbf{I})$;
2. compute $f_q(enc^*_\alpha(\mathbf{I})) = enc^*_\alpha(q(\mathbf{I}))$;
3. compute $q(\mathbf{I})$ from $enc^*_\alpha(q(\mathbf{I}))$. ■

18.2 $While_{new}$—*while* with New Values

Recall that, as discussed in the introduction to Chapter 14, *while* cannot go beyond PSPACE because (1) throughout the computation it uses only values from the input, and (2) it uses relations of fixed arity. The addition of integers as in $while_N$ is one way to break the space barrier. Another is to relax (1) or (2). Relaxing (1) is done by allowing the creation of new values not present in the input. Relaxing (2) yields an extension of *while* with *untyped algebra* (i.e., an algebra of relations with variable arities). In this and the next section, we describe two languages obtained by relaxing (1) and (2) and prove their completeness.

We first present the extension of *while* denoted $while_{new}$, which allows the creation of new values throughout the computation. The language *while* is modified as follows:

(i) There is a new instruction $R := new(S)$, where R and S are relational variables and $arity(R) = arity(S) + 1$;

(ii) The looping construct is of the form *while R do s*, where R is a relational variable.

The semantics of (i) is as follows: Relation R is obtained by extending each tuple of S by one distinct new value from **dom** not occurring in the input, the current state, or in the program. For example, if the value of S is the relation in Fig. 18.1, then R is of the form shown in that figure. The values α, β, γ are distinct new values[1] in **dom**.

The semantics of *while R do s* is that statement s is executed while R is nonempty. We could have used *while change* instead because each looping construct can simulate the other. However, in our context of value invention, it is practical to have the more direct control on loops provided by *while R*.

[1] If $arity(S) = 0$, then R is unary and contains one new value if $S = \{\langle\rangle\}$ and is empty if $S = \emptyset$. This allows the creation of values one by one. One might wonder if this kind of one-by-one value creation is sufficient. The answer is negative. The language with one-by-one value creation is equivalent to $while_N$ (see Exercise 18.6).

Note that the *new* construct is, strictly speaking, nondeterministic. The new values are arbitrary, so several possible outcomes are possible depending on the choice of values. However, the different outcomes differ *only* in the choice of new values. This is formalized by the following:

LEMMA 18.2.1 Let w be a $while_{new}$ program with input schema $\mathbf{R}$, and let R be a relation variable in w. Let $\mathbf{I}$ be an instance over $\mathbf{R}$, and let J, J' be two possible values of R at the same point during the execution of w on $\mathbf{I}$. Then there exists an isomorphism ρ from J to J' that is the identity on the constants occurring in $\mathbf{I}$ or w.

The proof of Lemma 18.2.1 is done by a straightforward induction on the number of steps in a partial execution of w on $\mathbf{I}$ (Exercise 18.7).

Recall that our definition of *query* requires that the answer be unique (i.e., the query must be deterministic). Therefore we must consider only $while_{new}$ programs whose answer never contains values introduced by the *new* statements. Such programs are called *well-behaved* $while_{new}$ programs. It is possible to give a syntactic restriction on $while_{new}$ programs that guarantees good behavior, can be checked, and yields a class of programs equivalent to all well-behaved $while_{new}$ programs (see Exercises 18.8 and 18.9).

We wish to show that well-behaved $while_{new}$ programs can express all queries. First we have to make sure that well-behaved $while_{new}$ programs do in fact express queries. This is shown next.

LEMMA 18.2.2 Each well-behaved $while_{new}$ program with input schema $\mathbf{R}$ and output schema *answer* expresses a query from *inst*($\mathbf{R}$) to *inst(answer)*.

Proof We need to show that well-behaved $while_{new}$ programs define *mappings* from *inst*($\mathbf{R}$) to *inst(answer)* (i.e., they are deterministic with respect to the final answer). Computability and genericity are straightforward. Let w be a well-behaved $while_{new}$ program with input schema $\mathbf{R}$ and output *answer*. Let I, I' be two possible values of *answer* after the execution of w on an instance $\mathbf{I}$ of $\mathbf{R}$. By Lemma 18.2.1, there exists an isomorphism ρ from I to I' that is the identity on values in $\mathbf{I}$ or w. Because w is well behaved, *answer* contains *only* values from $\mathbf{I}$ or w. Thus ρ is the identity and $I = I'$. ■

Note that although well-behaved programs are deterministic with respect to their final answer, they are not deterministic with respect to intermediate results that may contain new values.

We next show that well-behaved $while_{new}$ programs express all queries. The basic idea is simple. Recall that $while_N$ is complete on *ordered* databases. That is, for each query q, there is a $while_N$ program w that, given an enumeration of the input values in a relation *succ*, computes q. If, given an input, we were able to construct such an enumeration, we could then simulate $while_N$ to compute any desired query. Because of genericity, we cannot hope to construct *one* such enumeration. However, constructing *all* enumerations of values in the input would not violate genericity. Both $while_{new}$ and the language with variable arities considered in the next section can compute arbitrary queries precisely in this fashion: They first compute all possible enumerations of the input values and then

simulate a $while_N$ program on the ordered database corresponding to each enumeration. These computations yield the same result for all enumerations because queries are generic, so the result is independent of the particular enumeration used to encode the database (see Chapter 16).

Before proving the result, we show how we can construct all the possible enumerations of the elements in the active domain of the input.

Representation

Let **I** be an instance over **R**. Let *Success* be the set of all binary relations defining a successor relation over $adom(\mathbf{I})$. We can represent all the enumerations in *Success* with a 3-ary relation:

$$\overline{succ} = \bigcup_{I \in Success} I \times \{\alpha_I\},$$

where $\{\alpha_I \mid I \in Success\}$ is a set of distinct new values. [Each such α_I is used to denote a particular enumeration of $adom(\mathbf{I})$.] For example, Fig. 18.2 represents an instance **I** and the corresponding $\overline{succ}$.

Computation of $\overline{succ}$

We now argue that there exists a $while_{new}$ program w that, given **I**, computes $\overline{succ}$. Clearly, there is a $while_{new}$ program that, given **I**, produces a unary relation D containing all values in **I**. Following is a $while_{new}$ program $w_{\overline{succ}}$ that computes the relation $\overline{succ}$ starting from D (using a query q explained next):

I

a	b
a	c
c	a

$\overline{succ}$

a	b	α_1
b	c	α_1
a	c	α_2
c	b	α_2
b	a	α_3
a	c	α_3
b	c	α_4
c	a	α_4
c	a	α_5
a	b	α_5
c	b	α_6
b	a	α_6

$\widehat{succ}$

a	b	a	b	c
b	c	a	b	c
a	c	a	c	b
c	b	a	c	b
b	a	b	a	c
a	c	b	a	c
b	c	b	c	a
c	a	b	c	a
c	a	c	a	b
a	b	c	a	b
c	b	c	b	a
b	a	c	b	a

Figure 18.2: An example of $\overline{succ}$ and $\widehat{succ}$

$$
\begin{array}{l}
\overline{succ} := new(\sigma_{1\neq 2}(D \times D)); \\
\Delta \quad := q; \\
\textit{while } \Delta \textit{ do} \\
\qquad \textit{begin} \\
\qquad S := new(\Delta); \\
\qquad \overline{succ} := \left\{ \langle x, y, \alpha' \rangle \;\middle|\; \begin{array}{c} \exists \alpha, x', y'[S(x', y', \alpha, \alpha') \wedge \overline{succ}(x, y, \alpha)] \\ \vee \quad \exists \alpha[S(x, y, \alpha, \alpha')] \end{array} \right\}; \\
\qquad \Delta := q; \\
\qquad \textit{end}
\end{array}
$$

The intuition is that we construct in turn enumerations of subsets of size 2, 3, etc., until we obtain the enumerations of D. (To simplify, we assume that D contains more than two elements.) An enumeration of a subset of D consists of a successor (binary) relation over that subset. As mentioned earlier, the program associates a marking (invented value) with each such successor relation.

During the computation, $\overline{succ}$ contains the successor relation of subsets of size i computed so far. A triple $\langle a, b, \alpha \rangle$ indicates that b follows a in enumeration denoted α.

The first instruction computes the enumerations of subsets of size 2 (i.e., the distinct pairs of elements of D) and marks them with new values. At each iteration, Δ indicates for each enumeration the elements that are missing in this enumeration. More precisely, relation Δ must contain the following set of triples:

$$
\left\{ \langle a, b, \alpha \rangle \;\middle|\; \begin{array}{r} b \text{ does not occur in the successor relation corresponding to } \alpha \\ \text{and the last element of } \alpha \text{ is } a. \end{array} \right\}
$$

The relational query q computes the set Δ given a particular relation $\overline{succ}$. If Δ is not empty, for each α a new value α' is created for each element missing in α (i.e., the enumeration α is extended in all possible ways with each of the missing elements). This yields as many new enumerations from each α as missing elements.

This is iterated until Δ becomes empty, at which point all enumerations are complete. Note that if D contains n elements, the final result $\overline{succ}$ contains $n!$ enumerations.

THEOREM 18.2.3 The well-behaved $while_{new}$ programs express all queries.

Crux Let q be a query from $inst(\mathbf{R})$ to $inst(answer)$. Assume the query is generic (i.e., C-generic with $C = \emptyset$). The proof is easily modified for the case when the query is C-generic with $C \neq \emptyset$. It is sufficient to observe that

(*) for each $while_N$ program, there exists an equivalent well-behaved $while_{new}$ program.

Suppose that (*) holds. Let $w_{\overline{succ}}$ be the $while_{new}$ program computing $\overline{succ}$ from given $\mathbf{I}$ over $\mathbf{R}$. By Theorem 18.1.2 and (*), there exists a $while_{new}$ program $w(succ)$ that computes q using a successor relation $succ$. We construct another $while_{new}$ program $\overline{w}(\overline{succ})$ that computes q given $\mathbf{I}$ and $\overline{succ}$. Intuitively, $w(succ)$ is run in parallel for *all* possible

enumerations *succ* provided by $\overline{succ}$. All computations produce the same result and are placed in *answer*. The computations for different enumerations in $\overline{succ}$ are identified by the α marking the enumeration in $\overline{succ}$. To this end, each relation R of arity k in $w(succ)$ is replaced by a relation $\overline{R}$ of arity $k+1$. The extended database relations are first initialized by statements of the form $\overline{R} := R \times \pi_3(\overline{succ})$. Next the instructions of $w(succ)$ are modified as follows:

- $R := \{\langle u\rangle \mid \phi(u)\}$ becomes $\overline{R} := \{\langle u, \alpha\rangle \mid \exists y \exists z \overline{succ}(y, z, \alpha) \wedge \overline{\phi}(u, \alpha)\}$, where $\overline{\phi}(u, \alpha)$ is obtained from $\phi(u)$ by replacing each atom $S(v)$ by $\overline{S}(v, \alpha)$;
- *while change do* remains unchanged.

Finally the instruction $answer := \pi_{1..n}(\overline{answer})$, where $n = arity(answer)$, is appended at the end of the program. The following can be shown by induction on the steps of a partial execution of $\overline{w}(\overline{succ})$ on $\mathbf{I}$ (Exercise 18.10):

(**) At each point in the computation of $\overline{w}(\overline{succ})$ on $\mathbf{I}$, the set of tuples in relation $\overline{R}$ marked with α coincides with the value of R at the same point in the computation when $w(succ)$ is run on $\mathbf{I}$ and *succ* is the successor relation corresponding to α.

In particular, at the end of the computation of $\overline{w}(\overline{succ})$ on $\mathbf{I}$,

$$\overline{answer} = \bigcup_{\alpha} w(\alpha)(\mathbf{I}) \times \{\alpha\},$$

where α ranges over the enumeration markers. Because $w(\alpha)(\mathbf{I}) = q(\mathbf{I})$ for each α, it follows that *answer* contains $q(\mathbf{I})$ at the end of the computation. Thus query q is computable by a well-behaved $while_{new}$ program.

Thus it remains to show (*). Integer variables are easily simulated as follows. An integer variable i is represented by a binary variable R_i. If i contains the integer n, then R_i contains a successor relation for $n+1$ distinct new values:

$$\{\langle \alpha_j, \alpha_{j+1}\rangle \mid 0 \le j < n\}.$$

(The integer 0 is represented by an empty relation and the integer 1 by a singleton $\{\langle \alpha_0, \alpha_1\rangle\}$.) It is easy to find a $while_{new}$ program for *increment* and *decrement* of i. ■

We showed that well-behaved $while_{new}$ programs are complete with respect to our definition of query. Recall that $while_{new}$ programs that are not well behaved can compute a different kind of query that we excluded deliberately, which contains new values in the answer. It turns out, however, that such queries arise naturally in the context of object-oriented databases, where new object identifiers appear in query results (see Chapter 21). This requires extending our definition of query. In particular, the query is nondeterministic but, as discussed earlier, the different answers differ only in the particular choice of new values. This leads to the following extended notion of query:

DEFINITION 18.2.4 A *determinate query* is a relation Q from $inst(\mathbf{R})$ to *inst(answer)* such that

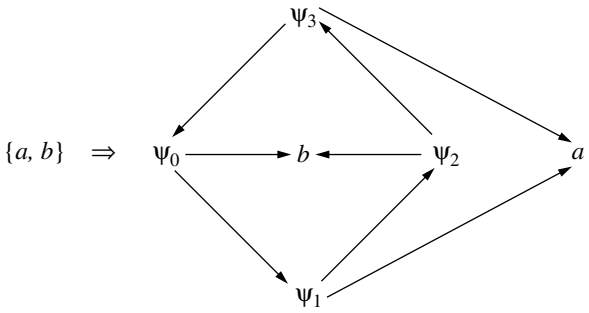


Figure 18.3: A query not expressible in $while_{new}$

- Q is computable;
- if $\langle I, J\rangle \in Q$ and ρ is a one-to-one mapping on constants, then $\langle \rho(I), \rho(J)\rangle \in Q$; and
- if $\langle I, J\rangle \in Q$ and $\langle I, J'\rangle \in Q$, then there exists an isomorphism from J to J' that is the identity on the constants in I.

A language is *determinate complete* if it expresses only determinate queries and all determinate queries.

Let Q be a determinate query. If $\langle I, J\rangle \in Q$ and ρ is a one-to-one mapping on constants leaving I fixed, then $\langle I, \rho(J)\rangle \in Q$.

The question arises whether $while_{new}$ remains complete with respect to this extended notion of query. Surprisingly, the answer is negative. Each $while_{new}$ query is determinate. However, we exhibit a simple determinate query that $while_{new}$ cannot express. Let q be the query with input schema $\mathbf{R} = \{S\}$, where S is unary, and output G, where G is binary. Let q be defined as follows: For each input I over S, if $I = \{a, b\}$ then $q(I) = \{\langle\psi_0, \psi_1\rangle, \langle\psi_1, \psi_2\rangle, \langle\psi_2, \psi_3\rangle, \langle\psi_3, \psi_0\rangle, \langle\psi_0, b\rangle, \langle\psi_1, a\rangle, \langle\psi_2, b\rangle, \langle\psi_3, a\rangle\}$ for some new elements $\psi_0, \psi_1, \psi_2, \psi_3$, and $q(I) = \emptyset$ otherwise (Fig. 18.3).

THEOREM 18.2.5 The query q is not expressible in $while_{new}$.

Proof The proof is by contradiction. Suppose w is a $while_{new}$ program expressing q. Consider the sequence of steps in the execution of w on an input $I = \{a, b\}$. We can assume without loss of generality that no invented value is ever deleted from the database (otherwise modify the program to keep all invented values in some new unary relation). For each invented value occurring in the computation, we define a trace that records how the value was invented and uniquely identifies it. More precisely, $trace(\alpha)$ is defined inductively as follows. If α is a constant, then $trace(\alpha) = \langle\alpha\rangle$. Suppose α is a new value created at step i with a *new* statement associating it with tuple $\langle x_1, \ldots, x_k\rangle$. Then $trace(\alpha) = \langle i, trace(x_1), \ldots, trace(x_k)\rangle$. Clearly, one can extend *trace* to tuples and rela-

tions in the natural manner. It is easily shown (Exercise 18.11) by induction on the number of steps in a partial execution of w on I that

(†) $trace(\alpha) = trace(\beta)$ iff $\alpha = \beta$;

(‡) for each instance J computed during the execution of w on input I, $trace(J)$ is closed under each automorphism ρ of I. In particular, for each α occurring in J, $\rho(trace(\alpha))$ equals $trace(\beta)$ for some β also occurring in J.

Consider now $trace(q(I))$ and the automorphism ρ of I [and therefore of $trace(q(I))$] defined by $\rho(a) = b$, $\rho(b) = a$. Note that $\rho^2 = id$ (the identity) and $\rho = \rho^{-1}$. Consider $\rho(trace(\psi_0))$. Because $\langle \psi_0, b\rangle \in q(I)$, it follows that $\langle trace(\psi_0), b\rangle \in trace(q(I))$. Because $\rho(b) = a$, it further follows that $\langle \rho(trace(\psi_0)), a\rangle \in trace(q(I))$ so $\rho(trace(\psi_0))$ is either $trace(\psi_1)$ or $trace(\psi_3)$. Suppose $\rho(trace(\psi_0)) = trace(\psi_1)$ (the other case is similar). From the fact that ρ is an automorphism of $trace(q(I))$ it follows that $\rho(trace(\psi_3)) = trace(\psi_0)$, $\rho(trace(\psi_2)) = trace(\psi_3)$, and $\rho(trace(\psi_1)) = trace(\psi_2)$. Consider now ρ^2. First, because $\rho^2 = id$, $\rho^2(trace(\psi_i)) = trace(\psi_i)$, $0 \le i \le 3$. On the other hand, $\rho^2(trace(\psi_0)) = \rho(\rho(trace(\psi_0))) = \rho(trace(\psi_1)) = trace(\psi_2)$. This is a contradiction. Hence q cannot be computed by $while_{new}$. ■

The preceding example shows that the presence of new values in the answer raises interesting questions with regard to completeness. There exist languages that express all queries with invented values in answers (see Exercise 18.14 for a complex construct that leads to a determinate-complete language). Value invention is common in object-oriented languages, in the form of object creation constructs (see Chapter 21).

18.3 $While_{uty}$—An Untyped Extension of *while*

We briefly describe in this section an alternative complete language obtained by relaxing the fixed-arity requirement of the languages encountered so far. This relaxation is done using an *untyped* version of relational algebra instead of the familiar typed version. We will obtain a language allowing us to construct relations of variable, data-dependent arity in the course of the computation. Although strictly speaking they are not needed, we also allow integer variables and integer manipulation, as in $while_N$. Intuitively, it is easy to see why this yields a complete language. Variable arities allow us to construct all enumerations of constants in the input, represented by sufficiently long tuples containing all constants. The ability to construct the enumerations and manipulate integers yields a complete language.

The first step in defining the untyped version of *while* is to define an untyped version of relational algebra. This means that operations must be defined so that they work on relations of arbitrary, unknown arity. Expressions in the untyped algebra are built from relation variables and constants and can also use *integer* variables and constants. Let i, j be integer variables, and for each integer k, let $\emptyset^k$ denote the empty relation of arity k. Untyped algebra expressions are built up using the following operations:

- If e, e' are expressions, then $e \cap e'$ and $e \cup e'$ are expressions; if $arity(e) = arity(e')$ the semantics is the usual; otherwise the result is $\emptyset^0$.

- If e is an expression, then $\neg e$ is an expression; the complement is with respect to the active domain (not including the integers).
- If e, f are expressions, then $e \times f$ is an expression; the semantics is the usual cross-product semantics.
- If e is an expression, then $\sigma_{i=j}(e)$ is an expression, where i, j are integer variables or constants; if $arity(e) \geq max\{i, j\}$ the semantics is the usual; otherwise the result is $\emptyset^0$.
- If e is an expression, then $\pi_{ij}(e)$ is an expression, where i, j are integer variables or constants; if $i \leq j$ and $arity(e) \geq max\{i, j\}$, this projects e on columns i through j; otherwise the result is $\emptyset^{|j-i|}$.
- If e is an expression, then $ex_{ij}(e)$ is an expression; if $arity(e) \geq max\{i, j\}$, this exchanges in each tuple in the result of e the i and j coordinates; otherwise the result is $\emptyset^0$.

We may also consider an untyped version of tuple relational calculus (see Exercise 18.15).

We can now define $while_{uty}$ programs. They are concatenations of statements of the form

- $i := j$, where i is an integer variable and j an integer variable or constant.
- $increment(i)$, $decrement(i)$, where i is an integer variable.
- *while* $i > 0$ *do* t, where i is an integer variable and t a program.
- $R := e$, where R is a relational variable and e an untyped algebra expression; the semantics here is that R is assigned the content *and arity* of e.
- *while* R *do* t, where R is a relational variable and t a program; the semantics is that the body of the loop is repeated as long as R is nonempty.

All relational variables that are not database relations are initialized to $\emptyset^0$; integer variables are initialized to 0.

Example 18.3.1 Following is a $while_{uty}$ program that computes the arity of a nonempty relation R in the integer variable n:

$S_0 := \{\langle\rangle\}$; $S_1 := S_0 \cup R$; $S_2 := \neg S_1$;
while S_2 *do*
  begin
  $n := n + 1$;
  $S_0 := S_0 \times D$;
  $S_1 := S_0 \cup R$;
  $S_2 := \neg S_1$;
  end

where D abbreviates an algebra expression computing the active domain [e.g., $\pi_{11}(R) \cup \neg\pi_{11}(R)$]. The program tries out increasing arities for R starting from 0. Recall that

whenever R and S_0 have different arities, the result of $S_0 \cup R$ is $\emptyset^0$. This allows us to detect when the appropriate arity has been found.

Remark 18.3.2 There is a much simpler set of constructs that yields the same power as $while_{uty}$. In general, programs are much harder to write in the resulting language, called QL, than in $while_{uty}$. One can show that the set of constructs of QL is minimal. The language QL is described next; it does not use integer variables. QL expressions are built from relational variables and constant relations as follows (D denotes the active domain):

- *equal* is an expression denoting $\{\langle a, a\rangle \mid a \in D\}$.
- $e \cap e'$ and $\neg e$ are defined as for $while_{uty}$; the complement is with respect to the active domain.
- If e is an expression, then $e\downarrow$ is an expression; this projects out the last coordinate of the result of e (and is $\emptyset^0$ if the arity is already zero).
- If e is an expression, then $e\uparrow$ is an expression; this produces the cross-product of e with D.
- If e is an expression, then $e\sim$ is an expression; if $arity(e) \geq 2$, then this exchanges the last two coordinates in each tuple in the result of e. Otherwise the answer is $\emptyset^0$.

Programs are built by concatenations of assignment statements ($R := e$) and *while* statements (*while R do s*). The semantics of the *while* is that the loop is iterated as long as R is nonempty.

We leave it to the reader to check that QL is equivalent to $while_{uty}$ (Exercise 18.17). We briefly describe the simulation of integers by QL. Let Z denote the constant 0-ary relation $\{\langle\rangle\}$. We can have Z represent the integer 0 and $Z\uparrow^n$ represent the integer n. Then $increment(n)$ is simulated by one application of $\uparrow$, and $decrement(n)$ is simulated by one application of $\downarrow$. A test of the form $x = 0$ becomes $e\downarrow = \emptyset$, where e is the untyped algebra expression representing the value of x. Thus we can simulate arbitrary computations on the integers.

Recall that our definition of query requires that both the input and output be instances over *fixed* schemas. On the other hand, in $while_{uty}$ relation arities are variable, so in general the arity of the answer is data dependent. This is a problem analogous to the one we encountered with $while_{new}$, which generally produces new values in the result. As in the case of $while_{new}$, we can define semantic and syntactic restrictions on $while_{uty}$ programs that guarantee that the programs compute queries. Call a $while_{uty}$ program *well behaved* if its answer is always of the same arity regardless of the input. Unfortunately, it can be shown that it is undecidable if a $while_{uty}$ program is well behaved (Exercise 18.19). However, there is a simple syntactic condition that guarantees good behavior and covers all well-behaved programs. A $while_{uty}$ program with answer relation *answer* is *syntactically well behaved* if the last instruction of the program is of the form $answer := \pi_{mn}(R)$, where m, n are integer constants. Clearly, syntactic good behavior guarantees good behavior and can be checked. Furthermore, it is obvious that each well-behaved $while_{uty}$ program is equivalent to some syntactically well-behaved program (Exercise 18.19).

We now prove the completeness of well-behaved $while_{uty}$ programs.

THEOREM 18.3.3 The well-behaved $while_{uty}$ programs express all queries.

Crux It is easily verified that all well-behaved $while_{uty}$ programs define queries. The proof that every query can be expressed by a well-behaved $while_{uty}$ program is similar to the proof of Theorem 18.2.3. Let q be a query with input schema $\mathbf{R}$. We proceed in two steps: First construct all orderings of constants from the input. Next simulate the $while_N$ program computing q on the ordered database corresponding to each ordering. The main difference with $while_{new}$ lies in how the orderings are computed. In $while_{uty}$, we use the arbitrary arity to construct a relation $R_<$ containing sufficiently long tuples each of which provides an enumeration of all constants. This is done by the following $while_{uty}$ program, where D stands for an algebra expression computing the active domain:

$R_< := \emptyset^0$;
$C := D$; $arityC := 1$;
while C *do*
  begin
  $R_< := C$;
  $C := C \times D$; $increment(arityC)$;
  for $i := 1$ *to* $(arityC - 1)$ *do*
    $C := C \cap \neg\sigma_{i=arity(C)}(C)$;
  end

Clearly, the looping construct $for\ i := 1\ to\ \ldots$ can be easily simulated. If the size of D is n, the result of the program is the set of n-tuples with distinct entries in $adom(D)$. Note that each such tuple t in $R_<$ provides a complete enumeration of the constants in D. Next one can easily construct a $while_{uty}$ program that constructs, for each such tuple t in $R_<$, the corresponding successor relation. More precisely, one can construct

$$\widehat{succ} = \bigcup_{t \in R_<} succ_t \times \{t\},$$

where $succ_t = \{\langle t(i), t(i+1)\rangle \mid 1 \leq i < n\}$ (see Fig. 18.2 and Exercise 18.20). ■

Untyped languages allow us to relax the restriction that the output schema is fixed. This may have a practical advantage because in some applications it may be necessary to have the output schema depend on the input data. However, in such cases one would likely prefer a richer type system rather than no typing at all.

The overall results on the expressiveness and complexity of relational query languages are summarized in Figs. 18.4 and 18.5. The main classes of queries and their inclusion structure are represented in Fig. 18.4 (solid arrows indicate strict inclusion; the dotted arrow indicates strict inclusion if PTIME $\neq$ PSPACE). Languages expressing each class of queries are listed in Fig. 18.5, which also contains information on complexity (first without assumptions, then with the assumption of an order on the database). In Fig. 18.5,

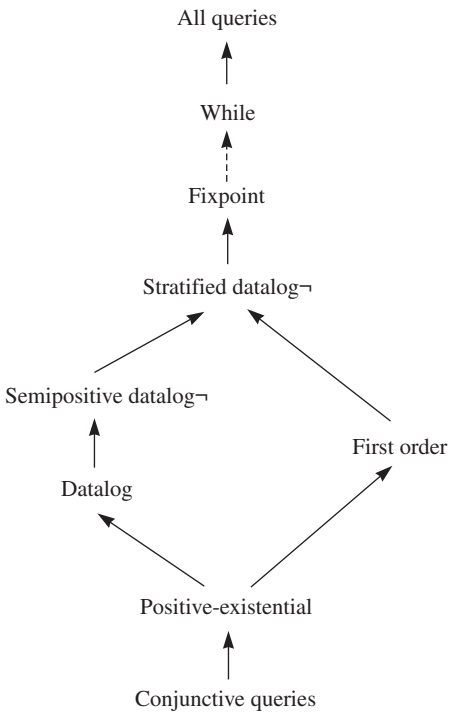


Figure 18.4: Main classes of queries

CALC($\exists$, $\wedge$) denotes the conjunctive calculus and CALC($\exists$, $\wedge$, $\vee$) denotes the positive-existential calculus.

Bibliographic Notes

The first complete language proposed was the language QL of Chandra and Harel [CH80b]. Chandra also considered a language equivalent to $while_N$, which he called LC [Cha81a]. It was shown that LC cannot compute *even*. Several other primitives are considered in [Cha81a] and their power is characterized. The language $while_{new}$ was defined in [AV90], where its completeness was also shown.

The languages considered in this chapter can be viewed as formalizing practical languages, such as C+SQL or O_2C, used to develop database applications. These languages combine standard computation (C) with database computation (SQL in the relational world or O_2 in the object-oriented world). In this direction, several computing devices were defined in [AV91b], and complexity-theoretic results are obtained using the devices. First an extension of Turing machines with a relational store, called *relational machine*, was shown to be equivalent to $while_N$. A further extension of relational machines equivalent to $while_{new}$ and $while_{uty}$, called *generic machine*, was also defined. In the generic machine,

Class of queries	*Languages*	*Complexity*	*Complexity* with order
conjunctive	CALC($\exists$, $\wedge$) SPJR algebra	$\subset$ LOGSPACE $\subset$ AC_0	$\subset$ LOGSPACE $\subset$ AC_0
positive-existential	CALC($\exists$, $\wedge$, $\vee$) SPJUR algebra nr-datalog	$\subset$ LOGSPACE $\subset$ AC_0	$\subset$ LOGSPACE $\subset$ AC_0
datalog	datalog	$\subset$ monotonic PTIME	$\subset$ monotonic PTIME
semipositive datalog$^\neg$	semipositive datalog$^\neg$	$\subset$ PTIME	$=$ PTIME (with *min*, *max*)
first order	CALC ALG nr-stratified datalog$^\neg$	$\subset$ LOGSPACE $\subset$ AC_0	$\subset$ LOGSPACE $\subset$ AC_0
stratified datalog$^\neg$	stratified datalog$^\neg$	$\subset$ PTIME	$=$ PTIME
fixpoint	CALC+μ^+ *while*$^+$ datalog$^\neg$ (fixpoint and well-founded semantics)	$\subset$ PTIME	$=$ PTIME
while	CALC+μ *while* datalog$^{\neg\neg}$ (fixpoint semantics)	$\subset$ PSPACE	$=$ PSPACE
all queries	$while_{uty}$ $while_{new}$	no bound	no bound

Figure 18.5: Languages and complexity

parallelism is used to allow simultaneous computations with all possible successor relations.

Queries with new values in their answers were first considered in [AK89], in the context of an object-oriented deductive language with object creation, called IQL. The notion of determinate query [VandBGAG92] is a recasting of the essentially equivalent notion of db transformation, formulated in [AK89]. In [AK89], the query in Theorem 18.2.5 is also exhibited, and it is shown that IQL without duplicate elimination cannot express it. Because IQL is more powerful than $while_{new}$, their result implies the result of Theorem 18.2.5. The issue of completeness of languages with object creation was further investigated in [AP92, VandBG92, VandBGAG92, VandBP95, DV91, DV93].

Finally it is easy to see that each (determinate) query can be computed in some natural *nondeterministic* extension of $while_{new}$ (e.g., with the witness operator of Chapter 17) [AV91c]. However, such programs may be nondeterministic so they do not define only determinate queries.

Exercises

Exercise 18.1 Let G be a graph. Consider a query "*Does the shortest path from a to b in G have property P?*" where G is a graph, P is a recursive property of the integers, and a, b are two particular vertexes of the graph. Show that such a query can be expressed in $while_N$.

Exercise 18.2 Prove that the query in Example 18.1.1 can be expressed (a) in *while*; (b) in *fixpoint*.

Exercise 18.3 Sketch a direct proof that *even* cannot be expressed by $while_N$ by extending the hyperplane technique used in the proof of Proposition 17.3.2.

♠ **Exercise 18.4** [AV94] Consider the language $\mathcal{L}$ augmenting $while_N$ by allowing mixing of integers with data. Specifically, the following instruction is allowed in addition to those of $while_N$: $R := \{\langle i_1, \ldots, i_k\rangle\}$, where R is a k-ary relation variable and $i_1, \ldots, i_k$ are integer variables. It is assumed that the domain of input values is disjoint from the integers. Complement (or negation) is taken with respect to the domain formed by all values in the database or program, including the integer values present in the database. The *well-behaved* $\mathcal{L}$ programs are those whose outputs never contain integers. Show that well-behaved $\mathcal{L}$ and $while_N$ are equivalent.

Exercise 18.5 Complete the proof of Theorem 18.1.2.

♠ **Exercise 18.6** [AV90] Consider a variation of the language $while_{new}$ where the $R := new(S)$ instruction is replaced by the simpler instruction "$R := new$" where R is unary. The semantics of this instruction is that R is assigned a singleton $\{\langle\alpha\rangle\}$, where α is a new value. Denote the new language by $while_{unary\text{-}new}$.

(a) Show that each query expressible in $while_N$ is also expressible in $while_{unary\text{-}new}$.
Hint: Use new values to represent integers. Specifically, to represent the integers up to n, construct a relation $succ_{int}$ containing a successor relation on n new values. The value of rank i with respect to *succ* represents integer i.

(b) Show that each query expressible in $while_{unary\text{-}new}$ is also expressible in $while_N$.
Hint: Again establish a correspondence between new values and integers. Then use Exercise 18.4.

Exercise 18.7 Prove Lemma 18.2.1.

Exercise 18.8 Prove that it is undecidable if a given $while_{new}$ program is well behaved.

★ **Exercise 18.9** In this exercise we define a syntactic restriction on $while_{new}$ programs that guarantees good behavior. Let w be a $while_{new}$ program. Without loss of generality, we can assume that all instructions contain at most one algebraic operation among $\cup, -, \pi, \times, \sigma$. Let the *not-well-behaved set of* w, denoted $Bad(w)$, be the smallest set of pairs of the form $\langle R, i\rangle$, where R is a relation in w and $1 \leq i \leq arity(R)$, such that

(a) if $S := new(R)$ is an instruction in w and $arity(S) = k$, then $\langle S, k\rangle \in Bad(w)$;

(b) if $S := T \cup R$ is in w and $\langle T, i\rangle \in Bad(w)$ or $\langle R, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$;

(c) if $S := T - R$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$;

(d) if $S := T \times R$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$; and if $\langle R, j\rangle \in Bad(w)$, then $\langle S, arity(T) + j\rangle \in Bad(w)$;

(e) if $S := \pi_{i_1 \ldots i_k}(T)$ is in w and $\langle T, i_j\rangle \in Bad(w)$, then $\langle S, j\rangle \in Bad(w)$;

(f) if $S := \sigma_{cond}(T)$ is in w and $\langle T, i\rangle \in Bad(w)$, then $\langle S, i\rangle \in Bad(w)$.

A $while_{new}$ program w is *syntactically well behaved* if

$$\{\langle answer, i\rangle \mid 1 \leq i \leq arity(answer)\} \cap Bad(w) = \emptyset.$$

(a) Outline a procedure to check that a given $while_{new}$ program is syntactically well behaved.

(b) Show that each syntactically well-behaved $while_{new}$ program is well behaved.

(c) Show that for each well-behaved $while_{new}$ program, there exists an equivalent syntactically well-behaved $while_{new}$ program.

Exercise 18.10 Prove (*) in the proof of Theorem 18.2.3.

Exercise 18.11 Prove (†) and (‡) in the proof of Theorem 18.2.5.

Exercise 18.12 Consider the query q exhibited in the proof of Theorem 18.2.5. Let q_2 be the query that, on input $I = \{a, b\}$, produces as answer two copies of $q(I)$. More precisely, for each ψ_i in $q(I)$, let ψ_i' be a distinct new value. Let $q'(I)$ be obtained from $q(I)$ by replacing ψ_i by ψ_i', and let $q_2(I) = q(I) \cup q'(I)$. Prove that q_2 can be expressed by a $while_{new}$ program.

♠ **Exercise 18.13** [DV91, DV93] Consider the instances I, J of Fig. 18.6. Consider a query q that, on input of the same pattern as I, returns J (up to an arbitrary choice of distinct β, θ_i) and otherwise returns the empty instance. Show that q is not expressible in $while_{new}$.

♠ **Exercise 18.14** (*Choose* [AK89]) Let $while_{new}^{choose}$ be obtained by augmenting $while_{new}$ with the following (determinate) *choose* construct. A program w may contain the instruction $choose(R)$ for some unary relation R. On input $\mathbf{I}$, when $choose(R)$ is applied in a state $\mathbf{J}$, the next state $\mathbf{J}'$ is defined as follows:

(a) if for each a, b in $\mathbf{J}(R)$, there is an automorphism of $\mathbf{J}$ that is the identity over $adom(\mathbf{I}, w)$ and maps a to b, $\mathbf{J}'$ is obtained from $\mathbf{J}$ by eliminating one arbitrary element in $\mathbf{J}(R)$;

(b) otherwise $\mathbf{J}'$ is just $\mathbf{J}$.

Show that $while_{new}^{choose}$ is determinate complete.

Exercise 18.15 One may consider an untyped version of tuple relational calculus. Untyped relations are used just like typed relations, except that terms of the form $t(i)$ are allowed, where t is a tuple variable and i an integer variable. Equivalence of queries now means that the queries yield the same answers given the same relations and values for the integer variables. Show that untyped relational calculus and untyped relational algebra are equivalent.

Exercise 18.16 Show that ex_{ij} is not redundant in the untyped algebra.

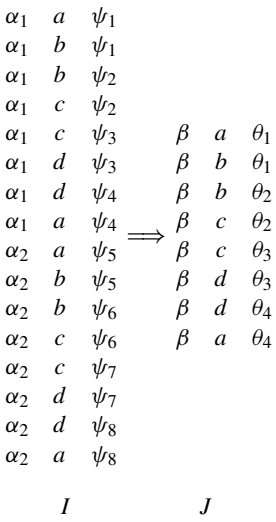


Figure 18.6: Another query not expressible in $while_{new}$

♠ **Exercise 18.17** Sketch a proof that $while_{uty}$ and the language QL described in Remark 18.3.2 are equivalent.

Exercise 18.18 Write a QL program computing the transitive closure of a binary relation.

♠ **Exercise 18.19** This exercise concerns well-behaved $while_{uty}$ programs. Show the following:

(a) It is undecidable whether a given $while_{uty}$ program is well behaved.

(b) Each syntactically well-behaved $while_{uty}$ program is well behaved.

(c) For each well-behaved $while_{uty}$ program, there exists an equivalent syntactically well-behaved $while_{uty}$ program.

Exercise 18.20 Write a $while_{uty}$ program that constructs the relation $\widehat{succ}$ from $R_<$ in the proof of Theorem 18.3.3.

♠ **Exercise 18.21** [AV91b] Prove that any query on a unary relation computed by a $while_{new}$ or $while_{uty}$ program in polynomial space is in FO. (For the purpose of this exercise, define the space used in a program execution as the maximum number of occurrences of constants in some instance produced in the execution of the program.) Note that, in particular, *even* cannot be computed in polynomial space in these languages.

♠ **Exercise 18.22** [AV91a] Consider the following extension of datalog$^{\neg\neg}$ with the ability to create new values. The rules are of the same form as datalog$^{\neg\neg}$ rules, but with a different semantics than the active domain semantics used for datalog$^{\neg\neg}$. The new semantics is the following. When rules are fired, all variables that occur in heads of rules but do not occur positively in the body are assigned distinct new values, not present in the input database, program, or any of the other relations in the program. A distinct value is assigned for each

applicable valuation of the variables positively bound in the body in each firing. This is similar to the *new* construct in $while_{new}$. For example, one firing of the rule

$$R(x, y, \alpha) \leftarrow P(x, y)$$

has the same effect as the $R := new(P)$ instruction in $while_{new}$. The resulting extension of datalog$^{\neg\neg}$ is denoted datalog$^{\neg\neg}_{new}$. The *well-behaved* datalog$^{\neg\neg}_{new}$ programs are those that never produce new values in the answer. Sketch a proof that well-behaved datalog$^{\neg\neg}_{new}$ programs express all queries.