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 **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 **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 $\{0, 1\}$. ■

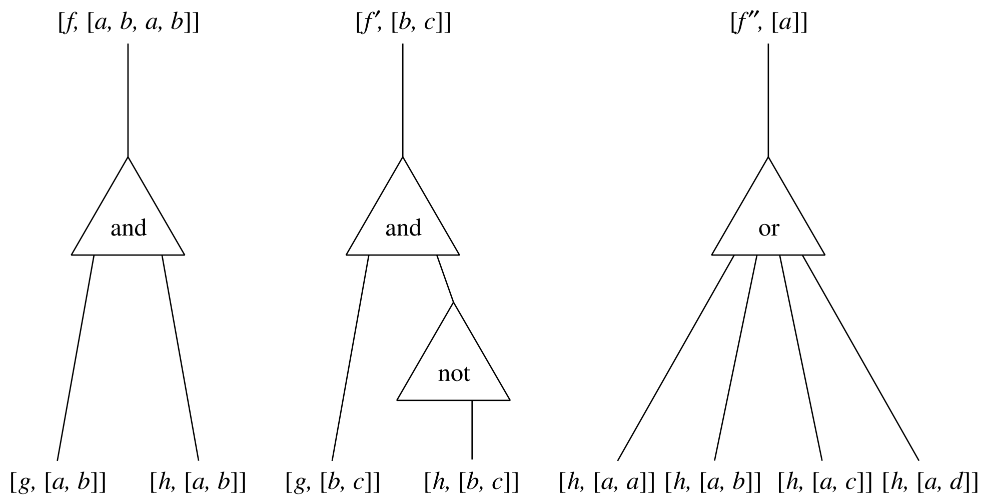

Figure 17.1: Some fragments of circuits

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

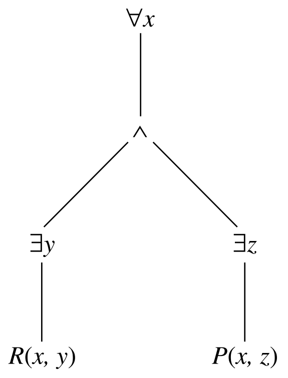

Figure 17.2: A syntax tree

a positive integer. The *game of length r associated with **I** and **J*** is played by two players called Spoiler and Duplicator, making r choices each. Spoiler starts by picking a constant occurring in **I** or **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 **I** (respectively, b_i in **J**). The set of pairs $\{(a_1, b_1), \ldots, (a_r, b_r)\}$ is a *round* of the game. The *subinstance* of **I** generated by $\{a_1, \ldots, a_r\}$, denoted $\mathbf{I}/\{a_1, \ldots, a_r\}$, consists of all facts in **I** using only these constants, and similarly for **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 **I** and **J** if he or she has a winning strategy (i.e., Duplicator can always win any game of length r on **I** and **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 **R** (see Exercise 17.3).

Intuitively, the equivalence $\mathbf{I} \equiv_r \mathbf{J}$ says that **I** and **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 **I** and **J** are two instances such that Duplicator has a winning strategy for the game of length r on the two instances, then **I** and **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 **I** and **J** be two instances over a database schema **R**. If $\mathbf{I} \equiv_r \mathbf{J}$, then for each CALC sentence φ over **R** with quantifier depth r, **I** and **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 **I** and **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 **I** and b_1, b_2, b_3 in **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 **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 **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 **I** such that

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

Regardless of which constant b_2 in **J** Duplicator picks,

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

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

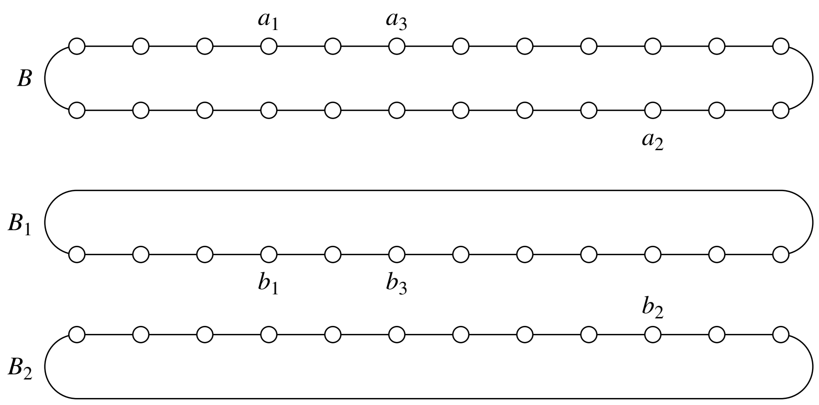

Figure 17.3: Two undistinguishable graphs

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]

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

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 0th 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 $(m-1)$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 $(m-1)$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 **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 **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 **I** is a one-to-one mapping ρ on *adom*(**I**) such that $\rho(\mathbf{I}) = \mathbf{I}$. For fixed **I**, consider the following equivalence relation $\equiv_k^{\mathbf{I}}$ on k-tuples of *adom*(**I**): $u \equiv_k^{\mathbf{I}} v$ iff there exists an automorphism ρ of **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 **I**. More precisely, suppose that **J** is an instance obtained at some point in the computation of w on input **I**. The genericity of *while* programs implies that if ρ is an automorphism of **I**, it is also an automorphism of **J**. Thus for each k-tuple u in some relation of **J** and each v such that $u \equiv_k^{\mathbf{I}} v$, v also belongs to that relation. Thus each relation in **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

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

specifying the pattern of connections represented in Fig. 17.4.

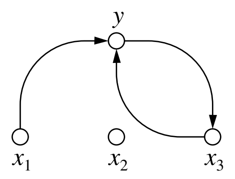

Figure 17.4: A connection pattern

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

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

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

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

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 **I** satisfying

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

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 **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 **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 **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 **R** that includes *succ*, such that q is in QPTIME on the ordered instances of **R**. Thus there is a polynomial p and Turing machine M' that, on input *enc*(**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 **I**, a TM M can be constructed that runs in polynomial time and that, on input *enc*(**I**), produces as output *enc*(q(**I**)). We now describe the construction of a CALC+μ^+ query q_M that is equivalent to q on ordered instances of **R**.

The fixpoint query q_M we construct, when given ordered input **I**, will operate in three phases: (α) construct an encoding of **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 **I**. Because M runs in polynomial time, there is some k such that M on input *enc*(**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 **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 **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 **I** be the ordered instance of **R** represented in Fig. 17.5. Then *enc*(**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

P[1#0#10][1#1#11][10#0#11][11#1#0]*succ*[0#1][1#10][10#11]

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*(**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 partition 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*.

R	A	B
	a	b
	a	c
	b	b
	b	c

I

R	A	B
	a	b
	b	b

I_1

R	A	B
	a	b
	b	c

I_2

R	A	B
	a	c
	b	b

I_3

R	A	B
	a	c
	b	c

I_4

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 $\mathit{while}^{(+)}$ 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 $\mathit{witness}_B$ to I may lead to several results [i.e., $\mathit{witness}_B(I)$ is either I_1, I_2, I_3 or I_4]. Intuitively, for each x occurring in the A column, $\mathit{witness}_B$ 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 $U = XY$, $X \cap Y = \emptyset$, $\mathit{witness}_Y(I)$ 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, $\mathit{witness}_U(J)$ 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 $\mathit{witness}_Y(J)$ 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 $\mathit{while}^{(+)}$ with *witness* is denoted by $\mathit{while}^{(+)}{+}W$. 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 $\mathit{while}{+}W$ query defines *all possible* successor relations on the constants from

R

b
d

succ

c	*a*

max

a

(a)

R

b

succ

c	*a*
a	*d*

max

d

(b)

R

succ

A	*B*
c	*a*
a	*d*
d	*b*

max

b

(c)

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$+\mu+W$. 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$+\mu^{+}+W$ 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

$\boxed{\text{G[00\#01][10\#00][10\#01][01\#01]\#[10]}}$

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 **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 **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 **R** be a database schema consisting of unary relations. Show that for each *while* query w on **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 $D \subseteq$ **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 **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 **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 $D \subseteq$ **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 $q(\mathbf{I})$ = **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+μ^++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+μ^++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+μ+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+μ^++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 **R** is a subset of *inst*(**R**) closed under isomorphisms of **dom**. Let **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 **Q** such that $while^+(\mathbf{Q})$ expresses exactly QPTIME.

(a) Show that for every finite set **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 **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.