14 Recursion and Negation

Vittorio: *Let's combine recursion and negation.*
Riccardo: *That sounds hard to me.*
Sergio: *It's no problem, just add* fixpoint *to the calculus, or* while *to the algebra.*
Riccardo: *That sounds hard to me.*
Vittorio: *OK—how about datalog with negation?*
Riccardo: *That sounds hard to me.*
Alice: *Riccardo, you are recursively negative.*

The query languages considered so far were obtained by augmenting the conjunctive queries successively with disjunction, negation, and recursion. In this chapter, we consider languages that provide both negation and recursion. They allow us to ask queries such as, *"Which are the pairs of metro stops which are* **not** *connected?"*. This query is not expressible in relational calculus and algebra or in datalog.

The integration of recursion and negation is natural and yields highly expressive languages. We will see how it can be achieved in the three paradigms considered so far: algebraic, logic, and deductive. The algebraic language is an extension of the algebra with a looping construct and an assignment, in the style of traditional imperative programming languages. The logic language is an extension of the calculus in which recursion is provided by a fixpoint operator. The deductive language extends datalog with negation.

In this chapter, the semantics of datalog with negation is defined from a purely computational perspective that is in the spirit of the algebraic approach. More natural and widely accepted model-theoretic semantics, such as stratified and well-founded semantics, are presented in Chapter 15.

As we consider increasingly powerful languages, the complexity of query evaluation becomes a greater concern. We consider two flavors of the languages in each paradigm: the inflationary one, which guarantees termination in time polynomial in the size of the database; and the noninflationary one, which only guarantees that a polynomial amount of space is used.[1] In the last section of this chapter, we show that the polynomial-time-bounded languages defined in the different paradigms are equivalent. The set of queries they define is called the *fixpoint queries*. The polynomial-space-bounded languages are also equivalent, and the corresponding set of queries is called the *while* queries. In Chapter 17, we examine in more detail the expressiveness and complexity of the *fixpoint* and *while* queries. Note that, in particular, the polynomial time and space bounds on the complexity

[1] For comparison, it is shown in Chapter 17 that CALC requires only logarithmic space.

of such queries imply that there are queries that are not *fixpoint* or *while* queries. More powerful languages are considered in Chapter 18.

Before describing specific languages, we present an example that illustrates the principles underlying the two flavors of the languages.

EXAMPLE The following is based on a version of the well-known "game of life," which is used to model biological evolution. The game starts with a set of cells, some of which are alive and some dead; the alive ones are colored in blue or red. (One cell may have two colors.) Each cell has other cells as neighbors. Suppose that a binary relation *Neighbor* holds the neighbor relation (considered as a symmetric relation) and that the information about living cells and their color is held in a binary relation *Alive* (see Fig. 14.1). Suppose first that a cell can change status from dead to alive following this rule:

(α) A dead cell becomes alive if it has at least two neighbors that are alive and have the same color. It then takes the color of the "parents."

The evolution of a particular population for the *Neighbor* graph of Fig. 14.1(a) is given in Fig. 14.1(b). Observe that the sets of tuples keep increasing and that we reach a stable state. This is an example of inflationary iteration.

Now suppose that the evolution also obeys the second rule:

(β) A live cell dies if it has more than three live neighbors.

The evolution of the population with the two rules is given in Fig. 14.1(c). Observe that the number of tuples sometimes decreases and that the computation diverges. This is an example of noninflationary iteration.

All languages that we consider use a fixed set of relation schemas throughout the computation. At any point in the computation, intermediate results contain only constants from the input database or that are specified in the query. Suppose the relations used in the computation have arities $r_1, \ldots, r_k$, the input database contains n constants, and the query refers to c constants. Then the number of tuples in any intermediate result is bounded by $\sum_{i=1}^{k} (n + c)^{r_i}$, which is a polynomial in n. Thus such queries can be evaluated in polynomial space. As will be seen when the formal definitions are in place, this implies that each noninflationary iteration, and hence each noninflationary query, can be evaluated in polynomial space, whether or not it terminates. In contrast, the inflationary semantics ensures termination by requiring that a tuple can never be deleted once it has been inserted. Because there are only polynomially many tuples, each such program terminates in polynomial time.

To summarize, the *inflationary* languages use iteration based on an "inflation of tuples." In all three paradigms, inflationary queries can be evaluated in polynomial time, and the same expressive power is obtained. The *noninflationary* languages use noninflationary or destructive assignment inside of iterations. In all three paradigms, noninflationary queries can be evaluated in polynomial space, and again the same expressive power is

Neighbor

a	*e*
b	*e*
c	*e*
d	*e*

(a) Neighbor

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*	
b	*red*	
c	*blue*	. . .
d	*red*	
e	*blue*	
e	*red*	

(b) Inflationary evolution

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*

Alive

a	*blue*
b	*red*
c	*blue*
d	*red*
e	*blue*
e	*red*

Alive

a	*blue*	
b	*red*	. . .
c	*blue*	
d	*red*	

(c) Noninflationary evolution

Figure 14.1: Game of life

obtained. (We note, however, that it remains open whether the inflationary and the non-inflationary languages have equivalent expressive power; we discuss this issue later.)

14.1 Algebra + *While*

Relational algebra is essentially a procedural language. Of the query languages, it is the closest to traditional imperative programming languages. Chapters 4 and 5 described how it can be extended syntactically using assignment (:=) and composition (;) without increasing its expressive power. The extensions of the algebra with recursion are also consistent with

the imperative paradigm and incorporate a *while* construct, which calls for the iteration of a program segment. The resulting language comes in two flavors: inflationary and noninflationary. The two versions of the language differ in the semantics of the assignment statement. The noninflationary version was the one first defined historically, and we discuss it next. The resulting language is called the *while* language.

Noninflationary Semantics

Recall from Chapter 4 that assignment statements can be incorporated into the algebra using expressions of the form $R := E$, where E is an algebra expression and R a relational variable of the same sort as the result of E. (The difference from Chapter 4 is that it is no longer required that each successive assignment statement use a distinct, previously unused variable.) In the *while* language, the semantics of an assignment statement is as follows: The value of R becomes the result of evaluating the algebra expression E on the current state of the database. This is the usual destructive assignment in imperative programming languages, where the old value of a variable is overwritten.

While statements have the form

while change do
  begin
  ⟨*loop body*⟩
  end

There is no explicit termination condition. Instead a loop runs as long as the execution of the body causes some change to some relation (i.e., until a stable state is reached). At the end of this section, we consider the introduction of explicit terminating conditions and see that this does not affect the language in an essential manner.

Nesting of loops is permitted. A *while program* is a finite sequence of assignment or while statements. The program uses a finite set of relational variables of specified sorts, including the names of relations in the input database. Relational variables that are not in the input database are initialized to the empty relation. A designated relational variable holds the output to the program at the end of the computation. The *image* (or *value*) of program P on $\mathbf{I}$, denoted $P(\mathbf{I})$, is the value finally assigned to the designated variable if P terminates on $\mathbf{I}$; otherwise $P(\mathbf{I})$ is undefined.

Example 14.1.1 (Transitive Closure) Consider a binary relation $G[AB]$, specifying the edges of a graph. The following *while* program computes in $T[AB]$ the transitive closure of G.

$T := G$;
while change do
  begin
  $T := T \cup \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
  end

A computation ends when T becomes stable, which means that no new edges were added in the current iteration, so T now holds the transitive closure of G.

EXAMPLE 14.1.2 (Add-Remove) Consider again a binary relation G specifying the edges of a graph. Each loop of the following program

- removes from G all edges $\langle a, b\rangle$ if there is a path of length 2 from a to b, and
- inserts an edge $\langle a, b\rangle$ if there is a vertex not directly connected to a and b.

This is iterated while some change occurs. The result is placed into the binary relation T. In addition, the binary relation variables *ToAdd* and *ToRemove* are used as "scratch paper." For the sake of readability, we use the calculus with active domain semantics whenever this is easier to understand than the corresponding algebra expression.

$T := G$;
while change do
  begin
  $ToRemove := \{\langle x, y\rangle \mid \exists z(T(x, z) \wedge T(z, y))\}$;
  $ToAdd := \{\langle x, y\rangle \mid \exists z(\neg T(x, z) \wedge \neg T(z, x) \wedge \neg T(y, z) \wedge \neg T(z, y))\}$;
  $T := (T \cup ToAdd) - ToRemove$;
  end

In the *Transitive Closure* example, the transitive closure query always terminates. This is not the case for the *Add-Remove* query. (Try the graph $\{\langle a, a\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle b, b\rangle\}$.) The halting problem for *while* programs is undecidable (i.e., there is no algorithm that, given a *while* program P, decides whether P halts on each input; see Exercise 14.2). Observe, however, that for a pair $(P, \mathbf{I})$, one can decide whether P halts on input $\mathbf{I}$ because, as argued earlier, *while* computations are in PSPACE.

Inflationary Semantics

We define next an inflationary version of the *while* language, denoted by *while*$^+$. The *while*$^+$ language differs with *while* in the semantics of the assignment statement. In particular, in *while*$^+$, assignment is cumulative rather than destructive: Execution of the statement assigning E to R results in *adding* the result of E to the old value of R. Thus no tuple is removed from any relation throughout the execution of the program. To distinguish the cumulative semantics from the destructive one, we use the notation $P \mathrel{+}= e$ for the cumulative semantics.

EXAMPLE 14.1.3 (Transitive Closure Revisited) Following is a *while*$^+$ program that computes the transitive closure of a graph represented by a binary relation $G[AB]$. The result is obtained in the variable $T[AB]$.

$T \mathrel{+}= G$;
while change do
  begin
  $T \mathrel{+}= \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G))$;
  end

This is almost exactly the same program as in the *while* language. The only difference is that because assignment is cumulative, it is not necessary to add the content of T to the result of the projection.

To conclude this section, we consider alternatives for the control condition of loops. Until now, we based termination on reaching a stable state. It is also common to use explicit terminating conditions, such as tests for emptiness of the form $E = \emptyset$, $E \neq \emptyset$, or $E \neq E'$, where E, E' are relational algebra expressions. The body of the loop is executed as long as the condition is satisfied. The following example shows how transitive closure is computed using explicit looping conditions.

Example 14.1.4 We use another relation schema $oldT$ also of sort AB.

$T \mathrel{+}= G;$
while $(T - oldT) \neq \emptyset$ *do*
  begin
  $oldT \mathrel{+}= T;$
  $T \mathrel{+}= \pi_{AB}(\delta_{B\to C}(T) \bowtie \delta_{A\to C}(G));$
  end

In the program, $oldT$ keeps track of the value of T resulting from the previous iteration of the loop. The computation ends when $oldT$ and T coincide, which means that no new edges were added in the current iteration, so T now holds the transitive closure of G.

It is easily shown that the use of such termination conditions does not modify the expressive power of *while*, and the use of conditions such as $E \neq E'$ does not modify the expressive power of *while*$^+$ (see Exercise 14.5).

In Section 14.4 we shall see that nesting of loops in *while* queries does not increase expressive power.

14.2 Calculus + Fixpoint

Just as in the case of the algebra, we provide inflationary and noninflationary extensions of the calculus with recursion. This could be done using assignment statements and while loops, as for the algebra. Indeed, we used calculus notation in Example 14.1.2 (*Add-Remove*). Instead we use an equivalent but more logic-oriented construct to augment the calculus. The construct, called a *fixpoint operator*, allows the iteration of calculus formulas up to a fixpoint. In effect, this allows defining relations inductively using calculus formulas. As with *while*, the fixpoint operator comes in a noninflationary and an inflationary flavor.

For the remainder of this chapter, as a notational convenience, we use active domain semantics for calculus queries. In addition, we often use a formula $\varphi(x_1, \ldots, x_n)$ as an abbreviation for the query $\{x_1, \ldots, x_n \mid \varphi(x_1, \ldots, x_n)\}$. These two simplifications do not affect the results developed.

Partial Fixpoints

The noninflationary version of the fixpoint operator is considered first. It is illustrated in the following example.

Example 14.2.1 (Transitive Closure Revisited) Consider again the transitive closure of a graph G. The relations J_n holding pairs of nodes at distance at most n can be defined inductively using the single formula

$$\varphi(T) = G(x, y) \ \vee \ T(x, y) \ \vee \ \exists z(T(x, z) \wedge G(z, y))$$

as follows:

$$J_0 = \emptyset;$$
$$J_n = \varphi(J_{n-1}), \quad n > 0.$$

Here $\varphi(J_{n-1})$ denotes the result of evaluating $\varphi(T)$ when the value of T is J_{n-1}. Note that, for each input G, the sequence $\{J_n\}_{n\geq 0}$ converges. That is, there exists some k for which $J_k = J_j$ for every $j > k$ (indeed, k is the diameter of the graph). Clearly, J_k holds the transitive closure of the graph. Thus the transitive closure of G can be defined as the limit of the foregoing sequence. Note that $J_k = \varphi(J_k)$, so J_k is also a *fixpoint* of $\varphi(T)$. The relation J_k thereby obtained is denoted by $\mu_T(\varphi(T))$. Then the transitive closure of G is defined by

$$\mu_T(G(x, y) \vee T(x, y) \vee \exists z(T(x, z) \wedge G(z, y))).$$

By definition, μ_T is an operator that produces a new relation (the fixpoint J_k) when applied to $\varphi(T)$. Note that, although T is used in $\varphi(T)$, T is not a database relation but rather a relation used to define inductively $\mu_T(\varphi(T))$ from the database, starting with $T = \emptyset$. T is said to be *bound* to μ_T. Indeed, μ_T is somewhat similar to a quantifier over relations. Note that the scope of the free variables of $\varphi(T)$ is restricted to $\varphi(T)$ by the operator μ_T.

In the preceding example, the limit of the sequence $\{J_n\}_{n\geq 0}$ happens to exist and is in fact the least fixpoint of φ. This is not always the case; the possibility of nontermination is illustrated next (and Exercise 14.4 considers cases in which a nonminimal fixpoint is reached).

Example 14.2.2 Consider

$$\varphi(T) = (x = 0 \wedge \neg T(0) \wedge \neg T(1)) \vee (x = 0 \wedge T(1)) \vee (x = 1 \wedge T(0)).$$

In this case the sequence $\{J_n\}_{n\geq 0}$ is $\emptyset$, $\{\langle 0\rangle\}$, $\{\langle 1\rangle\}$, $\{\langle 0\rangle\}$, ... (i.e., T flip-flops between zero and one). Thus the sequence does not converge, and $\mu_T(\varphi(T))$ is not defined. Situations in which μ is undefined correspond to nonterminating computations in the *while* language. The following nonterminating *while* program corresponds to $\mu_T(\varphi(T))$.

```
T := {⟨0⟩};
while change do
  begin
  T := {⟨0⟩, ⟨1⟩} − T;
  end
```

Because μ is only partially defined, it is called the *partial fixpoint operator*. We now define its syntax and semantics in more detail.

Partial Fixpoint Operator Let $\mathbf{R}$ be a database schema, and let $T[m]$ be a relation schema not in $\mathbf{R}$. Let $\mathbf{S}$ denote the schema $\mathbf{R} \cup \{T\}$. Let $\varphi(T)$ be a formula using T and relations in $\mathbf{R}$, with m free variables. Given an instance $\mathbf{I}$ over $\mathbf{R}$, $\mu_T(\varphi(T))$ denotes the relation that is the limit, *if it exists*, of the sequence $\{J_n\}_{n\geq 0}$ defined by

$$J_0 = \emptyset;$$
$$J_n = \varphi(J_{n-1}), \quad n > 0,$$

where $\varphi(J_{n-1})$ denotes the result of evaluating φ on the instance $\mathbf{J}_{n-1}$ over $\mathbf{S}$ whose restriction to $\mathbf{R}$ is $\mathbf{I}$ and $\mathbf{J}_{n-1}(T) = J_{n-1}$.

The expression $\mu_T(\varphi(T))$ denotes a new relation (if it is defined). In turn, it can be used in more complex formulas like any other relation. For example, $\mu_T(\varphi(T))(y, z)$ states that $\langle y, z\rangle$ is in $\mu_T(\varphi(T))$. If $\mu_T(\varphi(T))$ defines the transitive closure of G, the complement of the transitive closure is defined by

$$\{\langle x, y\rangle \mid \neg\, \mu_T(\varphi(T))(x, y)\}.$$

The extension of the calculus with μ is called *partial fixpoint logic*, denoted CALC+μ.

Partial Fixpoint Logic CALC+μ *formulas* are obtained by repeated applications of CALC operators ($\exists$, $\forall$, $\vee$, $\wedge$, $\neg$) and the partial fixpoint operator, starting from atoms. In particular, $\mu_T(\varphi(T))(e_1, \ldots, e_n)$, where T has arity n, $\varphi(T)$ has n free variables, and the e_i are variables or constants, is a formula. Its free variables are the variables in the set $\{e_1, \ldots, e_n\}$ [thus the scope of variables occurring inside $\varphi(T)$ consists of the subformula to which μ_T is applied]. Partial fixpoint operators can be nested. CALC+μ *queries* over a database schema $\mathbf{R}$ are expressions of the form

$$\{\langle e_1, \ldots, e_n\rangle \mid \xi\},$$

where ξ is a CALC+μ formula whose free variables are those occurring in $e_1, \ldots, e_n$. The formula ξ may use relation names in addition to those in $\mathbf{R}$; however, each occurrence P of such relation name must be bound to some partial fixpoint operator μ_P. The semantics of CALC+μ queries is defined as follows. First note that, given an instance $\mathbf{I}$ over $\mathbf{R}$ and a sentence σ in CALC+μ, there are three possibilities: σ is undefined on $\mathbf{I}$; σ is defined on $\mathbf{I}$

and is true; and σ is defined on $\mathbf{I}$ and is false. In particular, given an instance $\mathbf{I}$ over $\mathbf{R}$, the answer to the query

$$q = \{\langle e_1, \ldots, e_n\rangle \mid \xi\}$$

is undefined if the application of some μ in a subformula is undefined. Otherwise the answer to q is the n-ary relation consisting of all valuations ν of $e_1, \ldots, e_n$ for which $\xi(\nu(e_1), \ldots, \nu(e_n))$ is defined and true. The queries expressible in partial fixpoint logic are called the *partial fixpoint queries*.

Example 14.2.3 (Add-Remove Revisited) Consider again the query in Example 14.1.2. To express the query in CALC+μ, a difficulty arises: The *while* program initializes T to G before the while loop, whereas CALC+μ lacks the capability to do this directly. To distinguish the initialization step from the subsequent ones, we use a ternary relation Q and two distinct constants: 0 and 1. To indicate that the first step has been performed, we insert in Q the tuple $\langle 1, 1, 1\rangle$. The presence of $\langle 1, 1, 1\rangle$ in Q inhibits the repetition of the first step. Subsequently, an edge $\langle x, y\rangle$ is encoded in Q as $\langle x, y, 0\rangle$. The *while* program in Example 14.1.2 is equivalent to the CALC+μ query

$$\{\langle x, y\rangle \mid \mu_Q(\varphi(Q))(x, y, 0)\}$$

where

$$\begin{aligned}
&\varphi(Q) = \\
&\quad [\neg Q(1,1,1) \wedge [(G(x,y) \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 1)]] \\
&\quad \vee \\
&\quad [Q(1,1,1) \wedge [(x = 1 \wedge y = 1 \wedge z = 1) \vee \\
&\qquad\qquad ((z = ((z = 0) \wedge Q(x, y, 0) \wedge \neg\exists w(Q(x, w, 0) \wedge Q(w, y, 0))) \vee \\
&\qquad\qquad ((z = ((z = 0) \wedge \exists w(\neg Q(x, w, 0) \wedge \neg Q(w, x, 0) \wedge \\
&\qquad\qquad\quad \neg Q(y, w, 0) \wedge \neg Q(w, y, 0)))]].
\end{aligned}$$

Clearly, this query is more awkward than its counterpart in *while*. The simulation highlights some peculiarities of computing with CALC+μ.

In Section 14.4 it is shown that the family of partial fixpoint queries is equivalent to the $while$ queries. In the preceding definition of $\mu_T(\varphi(T))$, the scope of all free variables in φ is defined by μ_T. For example, if T is binary in the following

$$\exists y(P(y) \wedge \mu_T(\varphi(T, x, y))(z, w)),$$

then $\varphi(T, x, y)$ has free variables x, y. According to the definition, y is *not* free in $\mu_T(\varphi(T, x, y))(z, w)$ (the free variables are z, w). Hence the quantifier $\exists y$ applies to the y in $P(y)$ alone and has no relation to the y in $\mu_T(\varphi(T, x, y))(z, w)$. To avoid confusion, it is preferable to use distinct variable names in such cases. For instance, the preceding

sentence can be rewritten as

$$\exists y(P(y) \wedge \mu_T(\varphi(T, x', y'))(z, w)).$$

A variant of the fixpoint operator can be developed that permits free variables under the fixpoint operator, but this does not increase the expressive power (see Exercise 14.11).

Simultaneous Induction

Consider the following use of nested partial fixpoint operators, where G, P, and Q are binary:

$$\mu_P(G(x, y) \wedge \mu_Q(\varphi(P, Q))(x, y)).$$

Here $\varphi(P, Q)$ involves both P and Q. This corresponds to a nested iteration. In each iteration i in the computation of $\{J_n\}_{n\geq 0}$ over P, the fixpoint $\mu_Q(\varphi(P, Q))$ is recomputed for the successive values J_i of P.

In contrast, we now consider a generalization of the partial fixpoint that permits simultaneous iteration over two or more relations. For example, let $\mathbf{R}$ be a database schema and $\varphi(P, Q)$ and $\psi(P, Q)$ be calculus formulas using P and Q not in $\mathbf{R}$, such that the arity of P (respectively Q) is the number of free variables in φ (ψ). On input $\mathbf{I}$ over $\mathbf{R}$, one can define inductively the sequence $\{\mathbf{J}_n\}_{n\geq 0}$ of relations over $\{P, Q\}$ as follows:

$$\begin{aligned}
\mathbf{J}_0(P) &= \emptyset \\
\mathbf{J}_0(Q) &= \emptyset \\
\mathbf{J}_n(P) &= \varphi(\mathbf{J}_{n-1}(P), \mathbf{J}_{n-1}(Q)) \\
\mathbf{J}_n(Q) &= \psi(\mathbf{J}_{n-1}(P), \mathbf{J}_{n-1}(Q)).
\end{aligned}$$

Such a mutually recursive definition of $\mathbf{J}_n(P)$ and $\mathbf{J}_n(Q)$ is referred to as *simultaneous induction*. If the sequence $\{\mathbf{J}_n(P), \mathbf{J}_n(Q)\}_{n\geq 0}$ converges, the limit is a fixpoint of the mapping on pairs of relations defined by $\varphi(P, Q)$ and $\psi(P, Q)$. This pair of values for P and Q is denoted by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))$, and $\mu_{P,Q}$ is a *simultaneous induction partial fixpoint operator*. The value for P in $\mu_{P,Q}$ is denoted by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))(P)$ and the value for Q by $\mu_{P,Q}(\varphi(P, Q), \psi(P, Q))(Q)$. Clearly, simultaneous induction definitions like the foregoing can be extended for any number of relations. Simultaneous induction can simplify certain queries, as shown next.

Example 14.2.4 (Add-Remove by Simultaneous Induction) Consider again the query *Add-Remove* in Example 14.2.3. One can simplify the query by introducing an auxiliary unary relation *Off*, which inhibits the transfer of G into T after the first step in a direct fashion. T and *Off* are defined in a mutually recursive fashion by φ_{Off} and φ_T, respectively:

$$\varphi_{Off}(x) = x = 1$$
$$\varphi_T(x, y) = [\neg Off(1) \wedge G(x, y)]$$
$$\vee\ [Off(1) \wedge \neg\exists z(T(x, z) \wedge T(z, y)) \wedge$$
$$(T(x, y) \vee \exists z(\neg T(x, z) \wedge \neg T(z, x) \wedge \neg T(y, z) \wedge \neg T(z, y))].$$

The *Add-Remove* query can now be written as

$$\{\langle x, y\rangle \mid \mu_{Off,T}(\varphi_{Off}(Off, T), \varphi_T(Off, T))(T)(x, y)\}.$$

It turns out that using simultaneous induction instead of regular fixpoint operators does not provide additional power. For example, a CALC+μ formula equivalent to the query in Example 14.2.4 is the one shown in Example 14.2.3. More generally, we have the following:

Lemma 14.2.5 For some n, let $\varphi_i(R_1, \ldots, R_n)$ be CALC formulas, i in $[1..n]$, such that $\mu_{R_1,\ldots,R_n}(\varphi_1(R_1, \ldots, R_n), \ldots, \varphi_n(R_1, \ldots, R_n))$ is a correct formula. Then for each $i \in [1, n]$ there exist CALC formulas $\varphi_i'(Q)$ and tuples $\vec{e}_i$ of variables or constants such that for each i,

$$\mu_{R_1,\ldots,R_n}(\varphi_1(R_1, \ldots, R_n), \ldots, \varphi_n(R_1, \ldots, R_n))(R_i) \equiv \mu_Q(\varphi_i'(Q))(\vec{e}_i).$$

Crux We illustrate the construction with reference to the query of Example 14.2.4. Instead of using two relations *Off* and T, we use a ternary relation Q that encodes both *Off* and T. The extra coordinate is used to distinguish between tuples in T and tuples in *Off*. A tuple $\langle x\rangle$ in *Off* is encoded as a tuple $\langle x, 1, 1\rangle$ in Q. A tuple $\langle x, y\rangle$ in T is encoded as a tuple $\langle x, y, 0\rangle$ in Q. The final result is obtained by selecting from Q the tuples where the third coordinate is 0 and projecting the result on the first two coordinates. ■

Note that the use of the tuples $\vec{e}_i$ allows one to perform appropriate selections and projections on $\mu_Q(\varphi_i'(Q))$ necessary for decoding. These selections and projections are essential and cannot be avoided (see Exercise 14.17c).

Inflationary Fixpoint

The nonconvergence in some cases of the sequence $\{J_n\}_{n\geq 0}$ in the semantics of the partial fixpoint operator is similar to nonterminating computations in the *while* language with noninflationary semantics. The semantics of the partial fixpoint operator μ is essentially noninflationary because in the inductive definition of J_n, each step is a destructive assignment. As with *while*, we can make the semantics inflationary by having the assignment at each step of the induction be cumulative. This yields an inflationary version of μ, denoted by μ^+ and called the *inflationary fixpoint operator*, which is defined for all formulas and databases to which it is applied.

Inflationary Fixpoint Operators and Logic The definition of $\mu_T^+(\varphi(T))$ is identical to that of the partial fixpoint operator except that the sequence $\{J_n\}_{n\geq 0}$ is defined as follows:

$$J_0 = \emptyset;$$
$$J_n = J_{n-1} \cup \varphi(J_{n-1}), \quad n > 0.$$

This definition ensures that the sequence $\{J_n\}_{n\geq 0}$ is increasing: $J_{i-1} \subseteq J_i$ for each $i > 0$. Because for each instance there are finitely many tuples that can be added, the sequence converges in all cases.

Adding μ^+ instead of μ to CALC yields *inflationary fixpoint logic*, denoted by CALC+μ^+. Note that inflationary fixpoint queries are always defined.

The set of queries expressible by inflationary fixpoint logic is called the *fixpoint queries*. The fixpoint queries were historically defined first among the inflationary languages in the algebraic, logic, and deductive paradigms. Therefore the class of queries expressible in inflationary languages in the three paradigms has come to be referred to as the fixpoint queries.

As a simple example, the transitive closure of a graph G is defined by the following CALC+μ^+ query:

$$\{\langle x, y\rangle \mid \mu_T^+(G(x, y) \vee \exists z(T(x, z) \wedge G(z, y))(x, y)\}.$$

Recall that datalog as presented in Chapter 12 uses an inflationary operator and yields the minimal fixpoint of a set of rules. One may also be tempted to assume that an inflationary simultaneous induction of the form $\mu_{P,Q}^+(\varphi(P, Q), \psi(P, Q))$ is equivalent to a system of equational definitions of the form

$$P = \varphi(P, Q)$$
$$Q = \psi(P, Q)$$

and that it computes the unique minimal fixpoint for P and Q. However, one should be careful because the result of the inflationary fixpoint computation is only one of the possible fixpoints. As illustrated in the following example, this may not be minimal or the "naturally" expected fixpoint. (There may not exist a unique minimal fixpoint; see Exercise 14.4.)

Example 14.2.6 Consider the equation

$$T(x, y) = G(x, y) \vee T(x, y) \vee \exists z(T(x, z) \wedge G(z, y))$$
$$CT(x, y) = \neg T(x, y).$$

One is tempted to believe that the fixpoint of these two equations yields the complement of transitive closure. However, with the inflationary semantics

$$
\begin{aligned}
\mathbf{J}_0(T) &= \emptyset \\
\mathbf{J}_0(CT) &= \emptyset \\
\mathbf{J}_n(T) &= \mathbf{J}_{n-1}(T) \cup \{\langle x, y\rangle \mid G(x, y) \vee \mathbf{J}_{n-1}(T)(x, y) \\
&\qquad \vee \exists z(\mathbf{J}_{n-1}(T)(x, z) \wedge G(z, y))\} \\
\mathbf{J}_n(CT) &= \mathbf{J}_{n-1}(CT) \cup \{\langle x, y\rangle \mid \neg\mathbf{J}_{n-1}(T)(x, y)\}
\end{aligned}
$$

leads to saturating CT at the first iteration.

Positive and Monotone Formulas

Making the fixpoint operator inflationary by definition is not the only way to guarantee polynomial-time termination of the fixpoint iteration. An alternative approach is to restrict the formulas $\varphi(T)$ so that convergence of the sequence $\{J_n\}_{n\geq 0}$ associated with $\mu_T(\varphi(T))$ is guaranteed. One such restriction is monotonicity. Recall that a query q is *monotone* if for each **I, J**, $\mathbf{I} \subseteq \mathbf{J}$ then $q(\mathbf{I}) \subseteq q(\mathbf{J})$. One can again show that for such formulas, a least fixpoint always exists and that it is obtained after a finite (but unbounded) number of stages of inductive applications of the formula.

Unfortunately, monotonicity is an undecidable property for CALC. One can also restrict the application of fixpoint to positive formulas. This was historically the first track that was followed and presents the advantage that positiveness is a decidable (syntactic) property. It is done by requiring that T occur only *positively* in $\varphi(T)$ (i.e., under an even number of negations in the syntax tree of the formula). All formulas thereby obtained are monotone, and so $\mu_T(\varphi(T))$ is always defined (see Exercise 14.10).

It can be shown that the approach of inflationary fixpoint and the two approaches based on fixpoint of positive or monotone formulas are equivalent (i.e., the sets of queries expressed are identical; see Exercise 14.10).

Fixpoint Operators and Circumscription

In some sense, the fixpoint operators act as quantifiers on relational variables. This is somewhat similar to the well-known technique of *circumscription* studied in artificial intelligence. Suppose $\psi(T)$ is a calculus sentence (i.e., no free variables) that uses T in addition to relations from a database schema $\mathbf{R}$. The circumscription of $\psi(T)$ with respect to T, denoted here by $circ_T(\psi(T))$, can be thought of as an operator defining a new relation, starting from the database. More precisely, let $\mathbf{I}$ be an instance over $\mathbf{R}$. Then $circ_T(\psi(T))$ denotes the relation containing all tuples belonging to every relation T such that (1) $\psi(T)$ holds for $\mathbf{I}$, and (2) T is minimal under set inclusion[2] with this property. Consider now a fixpoint query. As stated earlier, fixpoint queries can be expressed using just fixpoint operators μ_T applied to formulas positive in T (i.e., T always appears in φ under an even number of negations). We claim that $\mu_T(\varphi(T)) = circ_T(\varphi'(T))$, where $\varphi'(T)$ is a sentence

[2] Other kinds of minimality have also been considered.

obtained from $\varphi(T)$ as follows:

$$\varphi'(T) = \forall x_1, \ldots \forall x_n(\varphi(T, x_1, \ldots, x_n) \rightarrow T(x_1, \ldots, x_n)),$$

where the arity of T is n. To see this, it is sufficient to note that $\mu_T(\varphi(T))$ is the unique minimal T satisfying $\varphi'(T)$. This uses the monotonicity of $\varphi(T)$ with respect to T, which follows from the fact that $\varphi(T)$ is positive in T (see Exercise 14.10). Although computing with circumscription is generally intractable, the fixpoint operator on positive formulas can always be evaluated in polynomial time. Thus the fixpoint operator can be viewed as a tractable restriction of circumscription.

14.3 Datalog with Negation

Datalog provides recursion but no negation. It defines only monotonic queries. Viewed from the standpoint of the deductive paradigm, datalog provides a form of *monotonic reasoning*. Adding negation to datalog rules permits the specification of nonmonotonic queries and hence of *nonmonotonic reasoning*.

Adding negation to datalog rules requires defining semantics for negative facts. This can be done in many ways. The different definitions depend to some extent on whether datalog is viewed in the deductive framework or simply as a specification formalism like any other query language. In this chapter, we examine the latter point of view. Then datalog with negation can essentially be viewed as a subset of the *while* or *fixpoint* queries and can be treated similarly. This is not necessarily appropriate in the deductive framework. For instance, the basic assumptions in the reasoning process may require that once a fact is assumed false at some point in the inferencing process, it should not be proven true at a later point. This idea lies at the core of stratified and well-founded semantics, two of the most widely accepted in the deductive framework. The deductive point of view is considered in depth in Chapter 15.

The semantics given here for datalog with negation follows the semantics given in Chapter 12 for datalog, but does not correspond directly to the semantics for nonrecursive datalog$^\neg$ given in Chapter 5. The semantics in Chapter 5 is inspired by the stratified semantics but can be simulated by (either of) the semantics presented in this chapter.

As in the previous section, we consider both inflationary and noninflationary versions of datalog with negation.

Inflationary Semantics

The inflationary language allows negations in bodies of rules and is denoted by *datalog*$^\neg$. Like datalog, its rules are used to infer a set of facts. Once a fact is inferred, it is never removed from the set of true facts. This yields the inflationary character of the language.

Example 14.3.1 We present a datalog$^\neg$ program with input a graph in binary relation G. The program computes the relation $closer(x, y, x', y')$ defined as follows:

closer(x, y, x', y') means that the distance $d(x, y)$ from x to y in G is smaller than the distance $d(x', y')$ from x' to y' [$d(x, y)$ is infinite if there is no path from x to y].

$$\begin{aligned} T(x, y) &\leftarrow G(x, y) \\ T(x, y) &\leftarrow T(x, z), G(z, y) \\ \textit{closer}(x, y, x', y') &\leftarrow T(x, y), \neg T(x', y') \end{aligned}$$

The program is evaluated as follows. The rules are fired simultaneously with all applicable valuations. At each such firing, some facts are inferred. This is repeated until no new facts can be inferred. A negative fact such as $\neg T(x', y')$ is true if $T(x', y')$ has not been inferred so far. This does not preclude $T(x', y')$ from being inferred at a later firing of the rules. One firing of the rules is called a stage in the evaluation of the program. In the preceding program, the transitive closure of G is computed in T. Consider the consecutive stages in the evaluation of the program. Note that if the fact $T(x, y)$ is inferred at stage n, then $d(x, y) = n$. So if $T(x', y')$ has not been inferred yet, this means that the distance between x and y is less than that between x' and y'. Thus if $T(x, y)$ and $\neg T(x', y')$ hold at some stage n, then $d(x, y) \leq n$ and $d(x', y') > n$ and *closer*(x, y, x', y') is inferred.

The formal syntax and semantics of datalog$^\neg$ are straightforward extensions of those for datalog. A datalog$^\neg$ *rule* is an expression of the form

$$A \leftarrow L_1, \ldots, L_n,$$

where A is an atom and each L_i is either an atom B_i (in which case it is called *positive*) or a negated atom $\neg B_i$ (in which case it is called *negative*). (In this chapter we use an active domain semantics for evaluating datalog$^\neg$ and so do not require that the rules be range restricted; see Exercise 14.13.)

A datalog$^\neg$ *program* is a nonempty finite set of datalog$^\neg$ rules. As for datalog programs, $sch(P)$ denotes the database schema consisting of all relations involved in the program P; the relations occurring in heads of rules are the *idb* relations of P, and the others are the *edb* relations of P.

The semantics of datalog$^\neg$ that we present in this chapter is an extension of the fixpoint semantics of datalog. Let $\mathbf{K}$ be an instance over $sch(P)$. Recall that an (active domain) *instantiation* of a rule $A \leftarrow L_1, \ldots, L_n$ is a rule $\nu(A) \leftarrow \nu(L_1), \ldots, \nu(L_n)$, where ν is a valuation that maps each variable into $adom(P, \mathbf{K})$. A fact A' is an *immediate consequence* for $\mathbf{K}$ and P if $A' \in \mathbf{K}(R)$ for some *edb* relation R, or $A' \leftarrow L'_1, \ldots, L'_n$ is an instantiation of a rule in P and each positive L'_i is a fact in $\mathbf{K}$, and for each negative $L'_i = \neg A'_i$, $A'_i \notin \mathbf{K}$. The *immediate consequence operator* of P, denoted Γ_P, is now defined as follows. For each $\mathbf{K}$ over $sch(P)$,

$$\Gamma_P(\mathbf{K}) = \mathbf{K} \cup \{A \mid A \text{ is an immediate consequence for } \mathbf{K} \text{ and } P\}.$$

Given an instance $\mathbf{I}$ over $edb(P)$, one can compute $\Gamma_P(\mathbf{I})$, $\Gamma_P^2(\mathbf{I})$, $\Gamma_P^3(\mathbf{I})$, etc. As suggested in Example 14.3.1, each application of Γ_P is called a *stage* in the evaluation. From the

definition of Γ_P, it follows that

$$\Gamma_P(\mathbf{I}) \subseteq \Gamma_P^2(\mathbf{I}) \subseteq \Gamma_P^3(\mathbf{I}) \subseteq \ldots.$$

As for datalog, the sequence reaches a fixpoint, denoted $\Gamma_P^\infty(\mathbf{I})$, after a finite number of steps. The restriction of this to the *idb* relations (or some subset thereof) is called the *image* (or *answer*) of P on $\mathbf{I}$.

An important difference with datalog is that $\Gamma_P^\infty(\mathbf{I})$ is no longer guaranteed to be a minimal model of P containing $\mathbf{I}$, as illustrated next.

Example 14.3.2 Let P be the program

$$R(0) \leftarrow Q(0), \neg R(1)$$
$$R(1) \leftarrow Q(0), \neg R(0).$$

Let $\mathbf{I} = \{Q(0)\}$. Then $P(\mathbf{I}) = \{Q(0), R(0), R(1)\}$. Although $P(\mathbf{I})$ is a model of P, it is not minimal. The minimal models containing $\mathbf{I}$ are $\{Q(0), R(0)\}$ and $\{Q(0), R(1)\}$.

As discussed in Chapter 12, the operational semantics of datalog based on the immediate consequence operator is equivalent to the natural semantics based on minimal models. As shown in the preceding example, there may not be a unique minimal model for a datalog$^\neg$ program, and the semantics given for datalog$^\neg$ may not yield any of the minimal models. The development of a natural model-theoretic semantics for datalog$^\neg$ thus calls for selecting a natural model from among several possible candidates. Inevitably, such choices are open to debate; Chapter 15 presents several alternatives.

Noninflationary Semantics

The language datalog$^\neg$ has inflationary semantics because the set of facts inferred through the consecutive firings of the rules is increasing. To obtain a noninflationary variant, there are several possibilities. One could keep the syntax of datalog$^\neg$ but make the semantics noninflationary by retaining, at each stage, only the newly inferred facts (see Exercise 14.16). Another possibility is to allow explicit retraction of a previously inferred fact. Syntactically, this can be done using negations in heads of rules, interpreted as deletions of facts. We adopt this solution here, in part because it brings our language closer to some practical languages that use so-called (production) rules in the sense of expert and active database systems. The resulting language is denoted by datalog$^{\neg\neg}$, to indicate that negations are allowed in both heads and bodies of rules.

Example 14.3.3 (Add-Remove Visited Again) The following datalog$^{\neg\neg}$ program computes in T the *Add-Remove* query of Example 14.1.2, given as input a graph G.

$$
\begin{aligned}
T(x, y) &\leftarrow G(x, y), \neg \mathit{off}(1) \\
\mathit{off}(1) &\leftarrow \\
\neg T(x, y) &\leftarrow T(x, z), T(z, y), \mathit{off}(1) \\
T(x, y) &\leftarrow \neg T(x, z), \neg T(z, x), \neg T(y, z), \neg T(z, y), \mathit{off}(1)
\end{aligned}
$$

Relation *off* is used to inhibit the first rule (initializing T to G) after the first step.

The immediate consequence operator Γ_P and semantics of a datalog$^{\neg\neg}$ program are analogous to those for datalog$^{\neg}$, with the following important proviso. If a negative literal $\neg A$ is inferred, the fact A is removed, unless A is also inferred in the same firing of the rules. This gives priority to inference of positive over negative facts and is somewhat arbitrary. Other possibilities are as follows: (1) Give priority to negative facts; (2) interpret the simultaneous inference of A and $\neg A$ as a "no-op" (i.e., including A in the new instance only if it is there in the old one); and (3) interpret the simultaneous inference of A and $\neg A$ as a contradiction that makes the result undefined. The chosen semantics has the advantage over possibility (3) that the semantics is always defined. In any case, the choice of semantics is not crucial: They yield equivalent languages (see Exercise 14.15).

With the semantics chosen previously, termination is no longer guaranteed. For instance, the program

$$
\begin{aligned}
T(0) &\leftarrow T(1) \\
\neg T(1) &\leftarrow T(1) \\
T(1) &\leftarrow T(0) \\
\neg T(0) &\leftarrow T(0)
\end{aligned}
$$

never terminates on input $T(0)$. The value of T flip-flops between $\{\langle 0\rangle\}$ and $\{\langle 1\rangle\}$, so no fixpoint is reached.

Datalog$^{\neg\neg}$ and Datalog$^{\neg}$ as Fragments of CALC+μ and CALC+μ^+

Consider datalog$^{\neg\neg}$. It can be viewed as a subset of CALC+μ in the following manner. Suppose that P is a datalog$^{\neg\neg}$ program. The *idb* relations defined by rules can alternately be defined by simultaneous induction using formulas that correspond to the rules. Each firing of the rules corresponds to one step in the simultaneous inductive definition. For instance, the simultaneous induction definition corresponding to the program in Example 14.3.3 is the one in Example 14.2.4. Because simultaneous induction can be simulated in CALC+μ (see Lemma 14.2.5), datalog$^{\neg\neg}$ can be simulated in CALC+μ. Moreover, notice that only a single application of the fixpoint operator is used in the simulation. Similar remarks apply to datalog$^{\neg}$ and CALC+μ^+. Furthermore, in the inflationary case it is easy to see that the formula can be chosen to be *existential* (i.e., its prenex normal form[3] uses only existential

[3] A CALC formula in prenex normal form is a formula $Q_1 x_1 \ldots Q_k x_k \varphi$ where Q_i, $1 \leq i \leq k$ are quantifiers and φ is quantifier free.

quantifiers). The same can be shown in the noninflationary case, although the proof is more subtle. In summary (see Exercise 14.18), the following applies:

Lemma 14.3.4 Each datalog$^{\neg\neg}$ (datalog$^{\neg}$) query is equivalent to a CALC+μ (CALC+μ^+) query of the form

$$\{\vec{x} \mid \mu_T^{(+)}(\varphi(T))(\vec{t})\},$$

where

(a) φ is an existential 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 $\vec{t}$.

The Rule Algebra

The examples of datalog$^{\neg}$ programs shown in this chapter make it clear that the semantics of such programs is not always easy to understand. There is a simple mechanism that facilitates the specification by the user of various customized semantics. This is done by means of the *rule algebra*, which allows specification of an order of firing of the rules as well as firing up to a fixpoint in an inflationary or noninflationary manner. For the inflationary version RA^+ of the rule algebra, the base expressions are individual datalog$^{\neg}$ rules; the semantics associated with a rule is to apply its immediate consequence operator once in a cumulative fashion. Union ($\cup$) can be used to specify simultaneous application of a pair of rules or more complex programs. The expression $P; Q$ specifies the composition of P and Q; its semantics is to execute P once and then Q once. Inflationary iteration of program P is called for by $(P)^+$. The noninflationary version of the rule algebra, denoted RA, starts with datalog$^{\neg}$ rules, but now with a noninflationary, destructive semantics, as defined in Exercise 14.16. Union and composition are generalized in the natural fashion, and the noninflationary iterator, denoted *, is used.

Example 14.3.5 Let P be the set of rules

$$T(x, y) \leftarrow G(x, y)$$
$$T(x, y) \leftarrow T(x, z), G(z, y)$$

and let Q consist of the rule

$$CT(x, y) \leftarrow \neg T(x, y).$$

The RA^+ program $(P)^+; Q$ computes in CT the complement of the transitive closure of G.

It follows easily from the results of Section 14.4 that RA^+ is equivalent to datalog$^{\neg}$, and RA is equivalent to noninflationary datalog$^{\neg}$ and hence to datalog$^{\neg\neg}$ (Exercise 14.23). Thus an RA^+ program can be compiled into a (possibly much more complicated) datalog$^{\neg}$

program. For instance, the RA^+ program in Example 14.3.5 is equivalent to the datalog$^\neg$ program in Example 14.4.2. The advantage of the rule algebra is the ease of expressing various semantics. In particular, RA^+ can be used easily to specify the stratified and well-founded semantics for datalog$^\neg$ introduced in Chapter 15.

14.4 Equivalence

The previous sections introduced inflationary and noninflationary recursive languages with negation in the algebraic, logic, and deductive paradigms. This section shows that the inflationary languages in the three paradigms, *while*$^+$, CALC+μ^+, and datalog$^\neg$, are equivalent and that the same holds for the noninflationary languages *while*, CALC+μ, and datalog$^{\neg\neg}$. This yields two classes of queries that are central in the theory of query languages: the *fixpoint* queries (expressed by the inflationary languages) and the *while* queries (expressed by the noninflationary languages). This is summarized in Fig. 14.2, at the end of the chapter.

We begin with the equivalence of the inflationary languages because it is the more difficult to show. The equivalence of CALC+μ^+ and *while*$^+$ is easy because the languages have similar capabilities: Program composition in *while*$^+$ corresponds closely to formula composition in CALC+μ^+, and the *while change* loop of *while*$^+$ is close to the inflationary fixpoint operator of CALC+μ^+. More difficult and surprising is the equivalence of these languages with datalog$^\neg$, because this much simpler language has no explicit constructs for program composition or nested recursion.

Lemma 14.4.1 CALC+μ^+ and *while*$^+$ are equivalent.

Proof We consider first the simulation of CALC+μ^+ queries by *while*$^+$. Let $\{\langle x_1,\ldots,x_m\rangle \mid \xi(x_1,\ldots,x_m)\}$ be a CALC+μ^+ query over an input database with schema $\mathbf{R}$. It suffices to show that there exists a *while*$^+$ program P_ξ that defines the same result as $\xi(x_1, \ldots, x_m)$ in some m-ary relation R_ξ. The proof is by induction on the depth of nesting of the fixpoint operator in ξ, denoted $d(\xi)$. If $d(\xi) = 0$ (i.e., ξ does not contain a fixpoint operator), then ξ is in CALC and P_ξ is

$$R_\xi \ += \ E_\xi,$$

where E_ξ is the relational algebra expression corresponding to ξ. Now suppose the statement is true for formulas with depth of nesting of the fixpoint operator less than $d (d > 0)$. Let ξ be a formula with $d(\xi) = d$.

If $\xi = \mu_Q(\varphi(Q))(f_1, \ldots, f_k)$, then P_ξ is

$Q \mathrel{+}= \emptyset$;
while change do
  begin
  E_φ;
  $Q \mathrel{+}= R_\varphi$
  end;
  $R_\xi \mathrel{+}= \pi(\sigma(Q))$,

where $\pi(\sigma(Q))$ denotes the selection and projection corresponding to $f_1, \ldots, f_k$.

Suppose now that ξ is obtained by first-order operations from k formulas $\xi_1, \ldots, \xi_k$, each having μ^+ as root. Let $E_\xi(R_{\xi_1}, \ldots, R_{\xi_k})$ be the relational algebra expression corresponding to ξ, where each subformula $\xi_i = \mu_Q(\varphi(Q))(e^i_1, \ldots, e^i_{n_i})$ is replaced by R_{ξ_i}. For each i, let P_{ξ_i} be a program that produces the value of $\mu_Q(\varphi(Q))(e^i_1, \ldots, e^i_{n_i})$ and places it into R_{ξ_i}. Then P_ξ is

$$P_{\xi_1}; \ldots; \; P_{\xi_k};$$
$$R_\xi \mathrel{+}= E_\xi(R_{\xi_1}, \ldots, R_{\xi_k}).$$

This completes the induction and the proof that CALC+μ^+ can be simulated by *while*$^+$. The converse simulation is similar (Exercise 14.20). ■

We now turn to the equivalence of CALC+μ^+ and datalog$^\neg$. Lemma 14.3.4 yields the subsumption of datalog$^\neg$ by CALC+μ^+. For the other direction, we simulate CALC+μ^+ queries using datalog$^\neg$. This simulation presents two main difficulties.

The first involves delaying the firing of a rule until after the completion of a fixpoint by another set of rules. Intuitively, this is hard because checking that the fixpoint has been reached involves checking the *nonexistence* rather than the existence of some valuation, and datalog$^\neg$ is more naturally geared toward checking the *existence* of valuations. The solution to this difficulty is illustrated in the following example.

Example 14.4.2 The following datalog$^\neg$ program computes the complement of the transitive closure of a graph G. The example illustrates the technique used to delay the firing of a rule (computing the complement) until the fixpoint of a set of rules (computing the transitive closure) has been reached (i.e., until the application of the transitivity rule yields no new tuples). To monitor this, the relations *old-T*, *old-T-except-final* are used. *old-T* follows the computation of T but is one step behind it. The relation *old-T-except-final* is identical to *old-T* but the rule defining it includes a clause that prevents it from firing when T has reached its last iteration. Thus *old-T* and *old-T-except-final* differ only in the iteration after the transitive closure T reaches its final value. In the subsequent iteration, the program recognizes that the fixpoint has been reached and fires the rule computing the complement in relation CT. The program is

$$\begin{aligned}
&T(x, y) && \leftarrow G(x, y)\\
&T(x, y) && \leftarrow G(x, z), T(z, y)\\
&\textit{old-T}(x, y) && \leftarrow T(x, y)\\
&\textit{old-T-except-final}(x, y) && \leftarrow T(x, y), T(x', z'), T(z', y'), \neg T(x', y')\\
&CT(x, y) && \leftarrow \neg T(x, y), \textit{old-T}(x', y'),\\
& && \qquad \neg \textit{old-T-except-final}(x', y')
\end{aligned}$$

(It is assumed that G is not empty; see Exercise 14.3.)

The second difficulty concerns keeping track of iterations in the computation of a fixpoint. Given a formula $\mu^+_T(\varphi(T))$, the simulation of φ itself may involve numerous relations other than T, whose behavior may be "sabotaged" by an overly zealous application of iteration of the immediate consequence operator. To overcome this, we separate the internal computation of φ from the external iteration over T, as illustrated in the following example.

Example 14.4.3 Let G be a binary relation schema. Consider the CALC+μ^+ query $\mu^+_{good}(\phi(good))(x)$, where

$$\phi = \forall y\ (G(y, x) \rightarrow good(y)).$$

Note that the query computes the set of nodes in G that are not reachable from a cycle (in other words, the nodes such that the length of paths leading to them is bounded). One application of $\varphi(good)$ is achieved by the datalog$^\neg$ program P:

$$\begin{aligned} bad(x) &\leftarrow G(y, x), \neg good(y) \\ delay &\leftarrow \\ good(x) &\leftarrow delay, \neg bad(x) \end{aligned}$$

Simply iterating P does not yield the desired result. Intuitively, the relations *delay* and *bad*, which are used as "scratch paper" in the computation of a single iteration of μ^+, cannot be reinitialized and so cannot be reused to perform the computation of subsequent iterations.

To surmount this problem, we essentially create a version of P for each iteration of $\varphi(good)$. The versions are distinguished by using "timestamps." The nodes themselves serve as timestamps. The timestamps marking iteration i are the values newly introduced in relation *good* at iteration $i - 1$. Relations *delay* and *delay-stamped* are used to delay the derivation of new tuples in *good* until *bad* and *bad-stamped* (respectively) have been computed in the current iteration. The process continues until no new values are introduced in an iteration. The full program is the union of the three rules given earlier, which perform the first iteration, and the following rules, which perform the iteration with timestamp t:

$$\begin{aligned} bad\text{-}stamped(x, t) &\leftarrow G(y, x), \neg good(y), good(t) \\ delay\text{-}stamped(t) &\leftarrow good(t) \\ good(x) &\leftarrow delay\text{-}stamped(t), \neg bad\text{-}stamped(x, t). \end{aligned}$$

We now embark on the formal demonstration that datalog$^\neg$ can simulate CALC+μ^+. We first introduce some notation relating to the timestamping of a program in the simulation. Let $m \geq 1$. For each relation schema Q, let $\overline{Q}$ be a new relational schema with $\text{arity}(\overline{Q}) = arity(Q) + m$. If $(\neg)Q(e_1, \ldots, e_n)$ is a literal and $\vec{z}$ an m-tuple of distinct variables, then $(\neg)Q(e_1, \ldots, e_n)[\vec{z}]$ denotes the literal $(\neg)\overline{Q}(e_1, \ldots, e_n, z_1, \ldots, z_m)$. For each program P and tuple $\vec{z}$, $P[\vec{z}]$ denotes the program obtained from P by replacing each literal A by $A[\vec{z}]$. Let P be a program and $B_1, \ldots, B_q$ a list of literals. Then P // $B_1, \ldots, B_q$ is the program obtained by appending $B_1, \ldots, B_q$ to the bodies of all rules in P.

To illustrate the previous notation, consider the program P consisting of the following two rules:

$$S(x, y) \leftarrow R(x, y)$$
$$S(x, y) \leftarrow R(x, z), S(z, y).$$

Then $P[z] \,//\, \neg T(x, w, y)$ is

$$\overline{S}(x, y, z) \leftarrow \overline{R}(x, y, z), \neg T(x, w, y)$$
$$\overline{S}(x, y, z) \leftarrow \overline{R}(x, z, z), \overline{S}(z, y, z), \neg T(x, w, y).$$

Lemma 14.4.4 CALC+μ^+ and datalog$^\neg$ are equivalent.

Proof As seen in Lemma 14.3.4, datalog$^\neg$ is essentially a fragment of CALC+μ^+, so we just need to show the simulation of CALC+μ^+ by datalog$^\neg$. The proof is by structural induction on the CALC+μ^+ formula. The core of the proof involves a control mechanism that delays firing certain rules until other rules have been evaluated. Therefore the induction hypothesis involves the capability to simulate the CALC+μ^+ formula using a datalog$^\neg$ program as well as to produce concomitantly a predicate that only becomes true when the simulation has been completed. More precisely, we will prove by induction the following: For each CALC+μ^+ formula φ over a database schema $\mathbf{R}$, there exists a datalog$^\neg$ program $prog(\varphi)$ whose *edb* relations are the relations in $\mathbf{R}$, whose *idb* relations include $result_\varphi$ with arity equal to the number of free variables in φ and a 0-ary relation $done_\varphi$ such that for every instance $\mathbf{I}$ over $\mathbf{R}$,

(i) $[prog(\varphi)(\mathbf{I})](result_\varphi) = \varphi(\mathbf{I})$, and

(ii) the 0-ary predicate $done_\varphi$ becomes true at the last stage in the evaluation of $prog(\varphi)$ on $\mathbf{I}$.

We will assume, without loss of generality, that no variable of φ occurs free and bound, or bound to more than one quantifier, that φ contains no $\forall$ or $\vee$, and that the initial query has the form $\{x_1, \ldots, x_n \mid \xi\}$, where $x_1, \ldots, x_n$ are distinct variables. Note that the last assumption implies that (i) establishes the desired result.

Suppose now that φ is an atom $\mathrm{R}(\vec{e})$. Let $\vec{x}$ be the tuple of distinct variables occurring in $\vec{e}$. Then $prog(\varphi)$ consists of the rules

$$done_\varphi \quad \leftarrow$$
$$result_\varphi(\vec{x}) \leftarrow R(\vec{e}).$$

There are four cases to consider for the induction step.

1. $\varphi = \alpha \wedge \beta$. Without loss of generality, we assume that the *idb* relations of $prog(\alpha)$ and $prog(\beta)$ are disjoint. Thus there is no interference between $prog(\alpha)$ and $prog(\beta)$. Let $\vec{x}$ and $\vec{y}$ be the tuples of distinct free variables of α and β, respectively, and let $\vec{z}$ be the tuple of distinct free variables occurring in $\vec{x}$ or $\vec{y}$.

Then $prog(\varphi)$ consists of the following rules:

$$\begin{aligned}
&prog(\alpha)\\
&prog(\beta)\\
&result_\varphi(\vec{z}) \leftarrow done_\alpha, done_\beta, result_\alpha(\vec{x}), result_\beta(\vec{y})\\
&done_\varphi \quad \leftarrow done_\alpha, done_\beta.
\end{aligned}$$

2. $\varphi = \exists\, x(\psi)$. Let $\vec{y}$ be the tuple of distinct free variables of ψ, and let $\vec{z}$ be the tuple obtained from $\vec{y}$ by removing the variable x. Then $prog(\varphi)$ consists of the rules

$$\begin{aligned}
&prog(\psi)\\
&result_\varphi(\vec{z}) \leftarrow done_\psi, result_\psi(\vec{y})\\
&done_\varphi \quad \leftarrow done_\psi.
\end{aligned}$$

3. $\varphi = \neg(\psi)$. Let $\vec{x}$ be the tuple of distinct free variables occurring in ψ. Then $prog(\varphi)$ consists of

$$\begin{aligned}
&prog(\psi)\\
&result_\varphi(\vec{x}) \leftarrow done_\psi, \neg result_\psi(\vec{x})\\
&done_\varphi \quad \leftarrow done_\psi.
\end{aligned}$$

4. $\varphi = \mu_S(\psi(S))(\vec{e})$. This case is the most involved, because it requires keeping track of the iterations in the computation of the fixpoint as well as bookkeeping to control the value of the special predicate $done_\varphi$. Intuitively, each iteration is marked by timestamps. The current timestamps consist of the tuples newly inserted in the previous iteration. The program $prog(\varphi)$ uses the following new auxiliary relations:
   - Relation $fixpoint_\varphi$ contains $\mu_S(\psi(S))$ at the end of the computation, and $result_\varphi$ contains $\mu_S(\psi(S))(\vec{e})$.
   - Relation run_φ contains the timestamps.
   - Relation $used_\varphi$ contains the timestamps introduced in the previous stages of the iteration. The active timestamps are in $run_\varphi - used_\varphi$.
   - Relation $not\text{-}final_\varphi$ is used to detect the final iteration (i.e., the iteration that adds no new tuples to $fixpoint_\varphi$). The presence of a timestamp in $used_\varphi - not\text{-}final_\varphi$ indicates that the final iteration has been completed.
   - Relations $delay_\varphi$ and $not\text{-}empty_\varphi$ are used for timing and to detect an empty result.

In the following, $\vec{y}$ and $\vec{t}$ are tuples of distinct variables with the same arity as S. We first have particular rules to perform the first iteration and to handle the special case of an empty result:

$$
\begin{aligned}
&\mathit{prog}(\psi)\\
&\mathit{fixpoint}_\varphi(\vec{y}) \leftarrow \mathit{result}_\psi(\vec{y}), \mathit{done}_\psi\\
&\mathit{delay}_\varphi \leftarrow \mathit{done}_\psi\\
&\mathit{not\text{-}empty}_\varphi \leftarrow \mathit{result}_\psi(\vec{y})\\
&\mathit{done}_\varphi \leftarrow \mathit{delay}_\varphi, \neg \mathit{not\text{-}empty}_\varphi.
\end{aligned}
$$

The remainder of the program contains the following rules:

- Stamping of the database and starting an iteration: For each R in ψ different from S and a tuple $\vec{x}$ of distinct variables with same arity as R,

$$
\begin{aligned}
\overline{R}(\vec{x}, \vec{t}) &\leftarrow R(\vec{x}), \mathit{fixpoint}_\varphi(\vec{t})\\
\mathit{run}_\varphi(\vec{t}) &\leftarrow \mathit{fixpoint}_\varphi(\vec{t})\\
\overline{S}(\vec{y}, \vec{t}) &\leftarrow \mathit{fixpoint}_\varphi(\vec{y}), \mathit{fixpoint}_\varphi(\vec{t}).
\end{aligned}
$$

- Timestamped iteration:

$$\mathit{prog}(\psi)[\vec{t}]//\mathit{run}_\varphi(\vec{t}), \neg \mathit{used}_\varphi(\vec{t})$$

- Maintain $\mathit{fixpoint}_\varphi$, $\mathit{not\text{-}last}_\varphi$, and used_φ:

$$
\begin{aligned}
\mathit{fixpoint}_\varphi(\vec{y}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t}), \overline{\mathit{result}}_\psi(\vec{y}, \vec{t}), \neg \mathit{used}_\varphi(\vec{t})\\
\mathit{not\text{-}final}_\varphi(\vec{t}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t}), \overline{\mathit{result}}_\psi(\vec{y}, \vec{t}), \neg \mathit{fixpoint}_\varphi(\vec{y})\\
\mathit{used}_\varphi(\vec{t}) &\leftarrow \overline{\mathit{done}}_\psi(\vec{t})
\end{aligned}
$$

- Produce the result and detect termination:

$$\mathit{result}_\varphi(\vec{z}) \leftarrow \mathit{fixpoint}_\varphi(\vec{e})$$

  where $\vec{z}$ is the tuple of distinct variables in $\vec{e}$,

$$\mathit{done}_\varphi \leftarrow \mathit{used}_\varphi(\vec{t}), \neg \mathit{not\text{-}final}_\varphi(\vec{t}).$$

It is easily verified by inspection that $\mathit{prog}(\varphi)$ satisfies (i) and (ii) under the induction hypothesis for cases (1) through (3). To see that (i) and (ii) hold in case (4), we carefully consider the stages in the evaluation of prog_φ. Let $\mathbf{I}$ be an instance over the relations in ψ other than S; let $J_0 = \emptyset$ be over S; and let $J_i = J_{i-1} \cup \psi(J_{i-1})$ for each $i > 0$. Then $\mu_S(\psi(S))(\mathbf{I}) = J_n$ for some n such that $J_n = J_{n-1}$. The program prog_φ simulates the consecutive iterations of this process. The first iteration is simulated using prog_ψ directly, whereas the subsequent iterations are simulated by prog_ψ timestamped with the tuples added at the previous iteration. (We omit consideration of the case in which the fixpoint is $\emptyset$; this is taken care of by the rules involving delay_φ and $\mathit{not\text{-}empty}_\varphi$.)

We focus on the stages in the evaluation of $prog_\varphi$ corresponding to the end of the simulation of each iteration of ψ. The stage in which the simulation of the first iteration is completed immediately follows the stage in which $done_\psi$ becomes true. The subsequent iterations are completed immediately following the stages in which

$$\exists \vec{t}(\overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t}))$$

becomes true. Thus let k_1 be the stage in which $done_\psi$ becomes true, and let k_i $(2 < i \leq n)$ be the successive stages in which

$$\exists \vec{t}(\overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t}))$$

is true. First note that

- at stage k_1

$$\{\vec{y} \mid result_\psi(\vec{y})\} = \psi(J_0);$$

- at stage $k_1 + 1$

$$fixpoint_\varphi = J_1.$$

For $i > 1$ it can be shown by induction on i that

- at stage k_i $(i \leq n)$

$$\{\vec{t} \mid \overline{done}_\psi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \psi(J_{i-2}) - J_{i-2} = J_{i-1} - J_{i-2}$$
$$\{\vec{y} \mid \overline{done}_\psi(\vec{t}) \wedge \overline{result}_\psi(\vec{y}, \vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \psi(J_{i-1});$$
$$\{\vec{t} \mid \overline{done}_\psi(\vec{t}) \wedge \overline{result}_\psi(\vec{y}, \vec{t}) \wedge \neg fixpoint_\varphi(\vec{y})\} = \psi(J_{i-1}) - J_{i-1} = J_i - J_{i-1};$$

- at stage $k_i + 1$ $(i < n)$

$$fixpoint_\varphi = J_{i-1} \cup \psi(J_{i-1}) = J_i,$$
$$used_\varphi = not\text{-}last_\varphi = \overline{done}_\psi = J_{i-1};$$

- at stage $k_i + 2$ $(i < n)$

$$\{\vec{t} \mid run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = J_i - J_{i-1},$$
$$\{\vec{x} \mid \overline{R}(\vec{x}, \vec{t}) \wedge run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = \mathbf{I}(R),$$
$$\{\vec{x} \mid \overline{S}(\vec{x}, \vec{t}) \wedge run_\varphi(\vec{t}) \wedge \neg used_\varphi(\vec{t})\} = J_i.$$

Finally, at stage $k_n + 1$

$$used_\varphi = J_{n-1},$$

$$\textit{not-last}_\varphi = J_{n-2},$$
$$\textit{fixpoint}_\varphi = J_n = \mu_S(\psi(S))(\mathbf{I}),$$

and at stage $k_n + 2$

$$\textit{result}_\varphi = \mu_S(\psi(S))(\vec{z})(\mathbf{I}),$$
$$\textit{done}_\varphi = \mathbf{true}.$$

Thus (i) and (ii) hold for $\textit{prog}_\varphi$ in case (4), which concludes the induction. ■

Lemmas 14.4.1 and 14.4.4 now yield the following:

THEOREM 14.4.5 *while*$^+$, CALC+μ^+, and datalog$^\neg$ are equivalent.

The set of queries expressible in *while*$^+$, CALC+μ^+, and datalog$^\neg$ is called the *fixpoint queries*. An analogous equivalence result can be proven for the noninflationary languages *while*, CALC+μ, and datalog$^{\neg\neg}$. The proof of the equivalence of CALC+μ and datalog$^{\neg\neg}$ is easier than in the inflationary case because the ability to perform deletions in datalog$^{\neg\neg}$ facilitates the task of simulating explicit control (see Exercise 14.21). Thus we can prove the following:

THEOREM 14.4.6 *while*, CALC+μ, and datalog$^{\neg\neg}$ are equivalent.

The set of queries expressible in *while*, CALC+μ, and datalog$^{\neg\neg}$ is called the *while queries*. We will look at the *fixpoint* queries and the *while* queries from a complexity and expressiveness standpoint in Chapter 17. Although the spirit of our discussion in this chapter suggested that *fixpoint* and *while* are distinct classes of queries, this is far from obvious. In fact, the question remains open: As shown in Chapter 17, *fixpoint* and *while* are equivalent iff PTIME = PSPACE (Theorem 17.4.3).

The equivalences among languages discussed in this chapter are summarized in Fig. 14.2.

Normal Forms

The two equivalence theorems just presented have interesting consequences for the underlying extensions of datalog and logic. First they show that these languages are closed under composition and complementation. For instance, if two mappings f, g, respectively, from a schema S to a schema S' and from S' to a schema S'' are expressible in datalog$^{\neg(\neg)}$, then $f \circ g$ and $\neg f$ are also expressible in datalog$^{\neg(\neg)}$. Analogous results are true for CALC+$\mu^{(+)}$.

A more dramatic consequence concerns the nesting of recursion in the calculus and algebra. Consider first CALC+μ^+. By the equivalence theorems, this is equivalent to datalog$^\neg$, which, in turn (by Lemma 14.3.4), is essentially a fragment of CALC+μ^+. This yields a normal form for CALC+μ^+ queries and implies that a single application of

Languages		*Class of queries*
inflationary	*while*$^+$ CALC $+\mu^+$ *datalog*$^\neg$	*fixpoint*
noninflationary	*while* CALC $+\mu$ *datalog*$^{\neg\neg}$	*while*

Figure 14.2: Summary of language equivalence results

the inflationary fixpoint operator is all that is needed. Similar remarks apply to CALC+μ queries. In summary, the following applies:

THEOREM 14.4.7 Each CALC+$\mu^{(+)}$ query is equivalent to a CALC+$\mu^{(+)}$ query of the form

$$\{\, \vec{x} \mid \mu_T^{(+)}(\varphi(T))(\vec{t})\},$$

where φ is an existential CALC formula.

Analogous normal forms can be shown for *while*$^{(+)}$ (Exercise 14.22) and for $RA^{(+)}$ (Exercise 14.24).

14.5 Recursion in Practical Languages

To date, there are numerous prototypes (but no commercial product) that provide query and update languages with recursion. Many of these languages provide semantics for recursion in the spirit of the procedural semantics described in this chapter. Prototypes implementing the deductive paradigm are discussed in Chapter 15.

SQL 2-3 (a norm provided by ISO/ANSII) allows **select** statements that define a table used recursively in the **from** and **where** clauses. Such recursion is also allowed in Starburst. The semantics of the recursion is inflationary, although noninflationary semantics can be achieved using deletion. An extension of SQL 2-3 is ESQL (Extended SQL). To illustrate the flavor of the syntax (which is typical for this category of languages), the following is an ESQL program defining a table SPARTS (subparts), the transitive closure of the table PARTS. This is done using a view creation mechanism.

create view *SPARTS* **as**
select *
from *PARTS*
union

select *P1.PART, P2.COMPONENT*
from *SPARTS P1, PARTS P2*
where *P1.COMPONENT* = *P2.PART*;

This is in the spirit of CALC+μ^+. With deletion, one can simulate CALC+μ. The system Postgres also provides similar iteration up to a fixpoint in its query language POSTQUEL.

A form of recursion closer to *while* and *while*$^+$ is provided by SQL embedded in full programming languages, such as C+SQL, which allows SQL statements coupled with C programs. The recursion is provided by while loops in the host language.

The recursion provided by datalog$^\neg$ and datalog$^{\neg\neg}$ is close in spirit to *production-rule* systems. Speaking loosely, a production rule has the form

if ⟨condition⟩ then ⟨action⟩.

Production rules permit the specification of database updates, whereas deductive rules usually support only database queries (with some notable exceptions). Note that the deletion in datalog$^{\neg\neg}$ can be viewed as providing an update capability. The production-rule approach has been studied widely in connection with expert systems in artificial intelligence; OPS5 is a well-known system that uses this approach.

A feature similar to recursive rules is found in the emerging field of *active* databases. In active databases, the rule condition is often broken into two pieces; one piece, called the *trigger*, is usually closely tied to the database (e.g., based on insertions to or deletions from relations) and can be implemented deep in the system.

In active database systems, rules are recursively fired when conditions become true in the database. Speaking in broad terms, the noninflationary languages studied in this chapter can be viewed as an abstraction of this behavior. For example, the database language RDL1 is close in spirit to the language datalog$^{\neg\neg}$. (See also Chapter 22 for a discussion of active databases.)

The language Graphlog, a visual language for queries on graphs developed at the University of Toronto, emphasizes queries involving paths and provides recursion specified using regular expressions that describe the shape of desired paths.

Bibliographic Notes

The *while* language was first introduced as RQ in [CH82] and as LE in [Cha81a]. The other noninflationary languages, CALC+μ and datalog$^{\neg\neg}$, were defined in [AV91a]. The equivalence of the noninflationary languages was also shown there.

The fixpoint languages have a long history. Logics with fixpoints have been considered by logicians in the general case where infinite structures (corresponding to infinite database instances) are allowed [Mos74]. In the finite case, which is relevant in this book, the fixpoint queries were first defined using the partial fixpoint operator μ_T applied only to formulas positive in T [CH82]. The language allowing applications of μ_T to formulas monotonic, but not necessarily positive, in T was further studied in [Gur84]. An interesting difference between unrestricted and finite models arises here: Every CALC formula monotone in some predicate R is equivalent for unrestricted structures to some CALC formula positive in R (Lyndon's lemma), whereas this is not the case for finite structures [AG87]. Monotonicity is undecidable for both cases [Gur84].

The languages (1) with fixpoint over positive formulas, (2) with fixpoint over monotone formulas, and (3) with inflationary fixpoint over arbitrary formulas were shown equivalent in [GS86]. As a side-effect, it was shown in [GS86] that the nesting of μ (or μ^+) provides no additional power. This fact had been proven earlier for the first language in [Imm86]. Moreover, a new alternative proof of the sufficiency of a single application of the fixpoint in CALC+μ^+ is provided in [Lei90]. The simultaneous induction lemma (Lemma 14.2.5) was also proven in [GS86], extending an analogous result of [Mos74] for infinite structures. Of the other inflationary languages, *while*$^+$ was defined in [AV90] and datalog$^\neg$ with fixpoint semantics was first defined in [AV88c, KP88].

The equivalence of datalog$^\neg$ with CALC+μ^+ and *while*$^+$ was shown in [AV91a]. The relationship between the *while* and *fixpoint* queries was investigated in [AV91b], where it was shown that they are equivalent iff PTIME = PSPACE. The issues of complexity and expressivity of *fixpoint* and *while* queries will be considered in detail in Chapter 17.

The rule algebra for logic programs was introduced in [IN88].

The game of life is described in detail in [Gar70]. The normal forms discussed in this chapter can be viewed as variations of well-known folk theorems, described in [Har80].

SQL 2-3 is described in an ISO/ANSII norm [57391, 69392]). Starburst is presented in [HCL$^+$90]. ESQL (Extended SQL) is described in [GV92]. The example ESQL program in Section 14.5 is from [GV92]. The query language of Postgres, POSTQUEL, is presented in [SR86]. OPS5 is described in [For81].

The area of *active* databases is the subject of numerous works, including [Mor83, Coh89, KDM88, SJGP90, MD89, WF90, HJ91a]. Early work on database triggers includes [Esw76, BC79]. The language RDL1 is presented in [dMS88].

The visual graph language Graphlog, developed at the University of Toronto, is described in [CM90, CM93a, CM93b].

Exercises

Exercise 14.1 (Game of life) Consider the two rules informally described in Example 14.1.

(a) Express the corresponding queries in datalog$^{\neg(\neg)}$, *while*$^{(+)}$, and CALC+$\mu^{(+)}$.

(b) Find an input for which a vertex keeps changing color forever under the second rule.

Exercise 14.2 Prove that the termination problem for a *while* program is undecidable (i.e., that it is undecidable, given a *while* query, whether it terminates on all inputs). *Hint:* Use a reduction of the containment problem for algebra queries.

Exercise 14.3 Recall the datalog$^{\neg\neg}$ program of Example 14.4.2.

(a) After how many stages does the program complete for an input graph of diameter n?

(b) Modify the program so that it also handles the case of empty graphs.

(c) Modify the program so that it terminates in order of $\log(n)$ stages for an input graph of diameter n.

Exercise 14.4 Recall the definition of $\mu_T(\varphi(T))$.

(a) Exhibit a formula φ such that $\varphi(T)$ has a unique minimal fixpoint on all inputs, and $\mu_T(\varphi(T))$ terminates on all inputs but does not evaluate to the minimal fixpoint on any of them.

(b) Exhibit a formula φ such that $\mu_T(\varphi(T))$ terminates on all inputs but φ does not have a unique minimal fixpoint on any input.

Exercise 14.5

(a) Give a *while* program with explicit looping condition for the query in Example 14.1.2.

(b) Prove that $while^{(+)}$ with looping conditions of the form $E = \emptyset$, $E \neq \emptyset$, $E = E'$, and $E \neq E'$, where E, E' are algebra expressions, is equivalent to $while^{(+)}$ with the *change* conditions.

Exercise 14.6 Consider the problem of finding, given two graphs G, G' over the same vertex set, the minimum set X of vertexes satisfying the following conditions: (1) For each vertex v, if all vertexes v' such that there is a G-edge from v' to v are in X, then v is in X; and (2) the analogue for G'-edges. Exhibit a *while* program and a *fixpoint* query that compute this set.

Exercise 14.7 Recall the CALC+μ^+ query of Example 14.4.3.

(a) Run the query on the input graph G:
$\{\langle a, b\rangle, \langle c, b\rangle, \langle b, d\rangle, \langle d, e\rangle, \langle e, f\rangle, \langle f, g\rangle, \langle g, d\rangle, \langle e, h\rangle, \langle i, j\rangle, \langle j, h\rangle\}$.

(b) Exhibit a $while^+$ program that computes *good*.

(c) Write a program in your favorite conventional programming language (e.g., C or LISP) that computes the good vertexes of a graph G. Compare it with the database queries developed in this chapter.

(d) Show that a vertex a is *good* if there is no path from a vertex belonging to a cycle to a. Using this as a starting point, propose an alternative algorithm for computing the good vertexes. Is your algorithm expressible in *while*? In *fixpoint*?

★ **Exercise 14.8** Suppose that the input consists of a graph G together with a successor relation on the vertexes of G [i.e., a binary relation *succ* such that (1) each element has exactly one successor, except for one that has none; and (2) each element in the binary relation G occurs in *succ*].

(a) Give a *fixpoint* query that tests whether the input satisfies (1) and (2).

(b) Sketch a *while* program computing the set of pairs $\langle a, b\rangle$ such that the shortest path from a to b is a prime number.

(c) Do (b) using a $while^+$ query.

Exercise 14.9 (Simultaneous induction) Prove Lemma 14.2.5.

♠ **Exercise 14.10** (Fixpoint over positive formulas) Let $\varphi(T)$ be a formula positive in T (i.e., each occurrence of T is under an even number of negations in the syntax tree of φ). Let $\mathbf{R}$ be the set of relations other than T occurring in $\varphi(T)$.

(a) Show that $\varphi(T)$ is monotonic in T. That is, for all instances $\mathbf{I}$ and $\mathbf{J}$ over $\mathbf{R} \cup \{T\}$ such that $\mathbf{I}(\mathbf{R}) = \mathbf{J}(\mathbf{R})$ and $\mathbf{I}(T) \subseteq \mathbf{J}(T)$,

$$\varphi(\mathbf{I}) \subseteq \varphi(\mathbf{J}).$$

(b) Show that $\mu_T(\varphi(T))$ is defined on every input instance.

(c) [GS86] Show that the family of CALC+μ queries with fixpoints only over positive formulas is equivalent to the CALC+μ^+ queries.

★ **Exercise 14.11** Suppose CALC+μ^+ is modified so that free variables are allowed under fixpoint operators. More precisely, let

$$\varphi(T, x_1, \ldots, x_n, y_1, \ldots, y_m)$$

be a formula where T has arity n and the x_i and y_j are free in φ. Then

$$\mu_{T,x_1,\ldots,x_n}(\varphi(T, x_1, \ldots, x_n, y_1, \ldots, y_m))(e_1, \ldots, e_n)$$

is a correct formula, whose free variables are the y_j and those occurring among the e_i. The fixpoint is defined with respect to a given valuation of the y_j. For instance,

$$\exists z \exists w (P(z) \wedge \mu_{T,x,y}(\varphi(T, x, y, z))(u, w))$$

is a well-formed formula. Give a precise definition of the semantics for queries using this operator. Show that this extension does not yield increased expressive power over CALC+μ^+. Do the same for CALC+μ.

Exercise 14.12 Let G be a graph. Give a *fixpoint* query in each of the three paradigms that computes the pairs of vertexes such that the shortest path between them is of even length.

Exercise 14.13 Let datalog$_{rr}^{\neg(\neg)}$ denote the family of datalog$^{\neg(\neg)}$ programs that are *range restricted*, in the sense that for each rule r and each variable x occurring in r, x occurs in a positive literal in the body of r. Prove that datalog$_{rr}^{\neg} \equiv$ datalog$^{\neg}$ and datalog$_{rr}^{\neg\neg} \equiv$ datalog$^{\neg\neg}$.

Exercise 14.14 Show that negations in bodies of rules are redundant in datalog$^{\neg\neg}$ (i.e., for each datalog$^{\neg\neg}$ program P there exists an equivalent datalog$^{\neg\neg}$ program Q that uses no negations in bodies of rules). *Hint:* Maintain the complement of each relation R in a new relation R', using deletions.

♠ **Exercise 14.15** Consider the following semantics for negations in heads of datalog$^{\neg\neg}$ rules:

(α) the semantics giving priority to positive over negative facts inferred simultaneously (adopted in this chapter),

(β) the semantics giving priority to negative over positive facts inferred simultaneously,

(γ) the semantics in which simultaneous inference of A and $\neg A$ leads to a "no-op" (i.e., including A in the new instance only if it is there in the old one), and

(δ) the semantics prohibiting the simultaneous inference of a fact and its negation by making the result undefined in such circumstances.

For a datalog$^{\neg\neg}$ program P, let P_ξ, denote the program P with semantics $\xi \in \{\alpha, \beta, \gamma, \delta\}$.

(a) Give an example of a program P for which P_α, P_β, P_γ, and P_δ define distinct queries.

(b) Show that it is undecidable, for a given program P, whether P_δ never simultaneously infers a positive fact and its negation for any input.

(c) Let datalog$_\xi^{\neg\neg}$ denote the family of queries P_ξ for $\xi \in \{\alpha, \beta, \gamma\}$. Prove that datalog$_\alpha^{\neg\neg} \equiv$ datalog$_\beta^{\neg\neg} \equiv$ datalog$_\gamma^{\neg\neg}$.

(d) Give a syntactic condition on datalog$^{\neg\neg}$ programs such that under the δ semantics they never simultaneously infer a positve fact and its negation, and such that the resulting query language is equivalent to datalog$_\alpha^{\neg\neg}$.

Exercise 14.16 (Noninflationary datalog$^\neg$) The semantics of datalog$^\neg$ can be made noninflationary by defining the immediate consequence operator to be destructive in the sense that only the newly inferred facts are kept after each firing of the rules. Show that, with this semantics, datalog$^\neg$ is equivalent to datalog$^{\neg\neg}$.

★ **Exercise 14.17** (Multiple versus single carriers)

(a) Consider a datalog$^\neg$ program P producing the answer to a query in an *idb* relation S. Prove that there exists a program Q with the same *edb* relations as P and just one idb relation T such that, for each *edb* instance $\mathbf{I}$,

$$[P(\mathbf{I})](S) = \pi(\sigma([Q(\mathbf{I})](T))),$$

where σ denotes a selection and π a projection.

(b) Show that the projection π and selection σ in part (a) are indispensable. *Hint:* Suppose there is a datalog$^\neg$ program with a single *edb* relation computing the complement of transitive closure of a graph. Reach a contradiction by showing in this case that connectivity of a graph is expressible in relational calculus. (It is shown in Chapter 17 that connectivity is not expressible in the calculus.)

(c) Show that the projection and selection used in Lemma 14.2.5 are also indispensable.

★ **Exercise 14.18**

(a) Prove Lemma 14.3.4 for the inflationary case.

(b) Prove Lemma 14.3.4 for the noninflationary case. *Hint:* For datalog$^{\neg\neg}$, the straightforward simulation yields a formula $\mu_T(\varphi(T))(\vec{x})$, where φ may contain negations over existential quantifiers to simulate the semantics of deletions in heads of rules of the datalog$^{\neg\neg}$ program. Use instead the noninflationary version of datalog$^\neg$ described in Exercise 14.16.

Exercise 14.19 Prove that the simulation in Example 14.4.3 works.

Exercise 14.20 Complete the proof of Lemma 14.4.1 (i.e., prove that each *while*$^+$ program can be simulated by a CALC+μ^+ program).

★ **Exercise 14.21** Prove the noninflationary analogue of Lemma 14.4.4 (i.e., that datalog$^{\neg\neg}$ can simulate CALC+μ). *Hint:* Simplify the simulation in Lemma 14.4.4 by taking advantage of the ability to delete in datalog$^{\neg\neg}$. For instance, rules can be inhibited using "switches," which can be turned on and off. Furthermore, no timestamping is needed.

Exercise 14.22 Formulate and prove a normal form for *while*$^+$ and *while*, analogous to the normal forms stated for CALC+μ^+ and CALC+μ.

Exercise 14.23 Prove that RA^+ is equivalent to datalog$^\neg$ and RA is equivalent to noninflationary datalog$^\neg$, and hence to datalog$^{\neg\neg}$. *Hint:* Use Theorems 14.4.5 and 14.4.6 and Exercise 14.16.

Exercise 14.24 Let the *star height* of an RA program be the maximum number of occurrences of * and $^+$ on a path in the syntax tree of the program. Show that each RA program is equivalent to an RA program of star height one.