20 Complex Values

Alice: *Complex values?*

Riccardo: *We could have used a different title: nested relations, complex objects, structured objects . . .*

Vittorio: *. . . N1NF, NFNF, NF^2, NF2, V-relation . . . I have seen all these names and others as well.*

Sergio: *In a nutshell, relations are nested within relations; something like Matriochka relations.*

Alice: *Oh, yes. I love Matriochkas.*

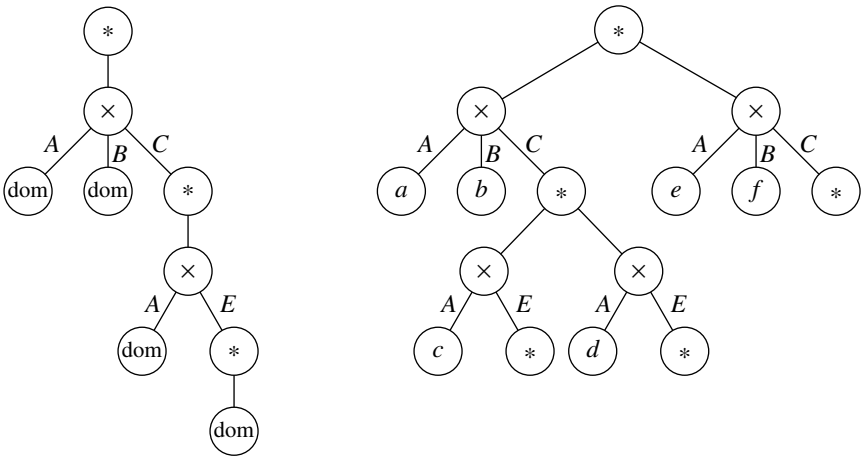
Although we praised the simplicity of the data structure in the relational model, this simplicity becomes a severe limitation when designing many practical database applications. To overcome this problem, the complex value model has been proposed as a significant extension of the relational one. This extension is the topic of this chapter.

Intuitively, complex values are relations in which the entries are not required to be atomic (as in the relational model) but are allowed to be themselves relations. The data structure in the relational model (the relation) can be viewed as the result of applying to atomic values two constructors: a *tuple constructor* to make tuples and a *set constructor* to make sets of tuples (relations). Complex values allow the application of the tuple and set constructor recursively. Thus they can be viewed as finite trees whose internal nodes indicate the use of the tuple and finite set constructors. Clearly, a relation is a special kind of complex value: a set of tuples of atomic values.

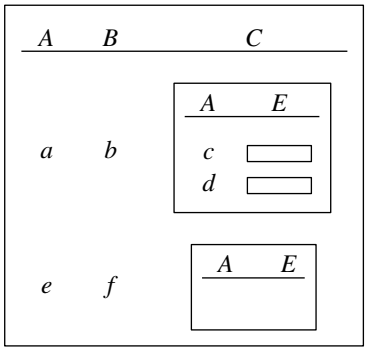
At the schema level, we will specify a set of complex *sorts* (or types). These indicate the structure of the data. At the instance level, sets of complex values corresponding to these sorts are provided. For example, we have the following:

Sort	*Complex Value*
dom	a
{**dom**}	$\{a, b, c\}$
$\langle A : \mathbf{dom}, B : \mathbf{dom}\rangle$	$\langle A : a, B : b\rangle$
$\{\langle A : \mathbf{dom}, B : \mathbf{dom}\rangle\}$	$\{\langle A : a, B : b\rangle, \langle A : b, B : a\rangle\}$
{{**dom**}}	$\{\{a, b\}, \{a\}, \{\,\}\}$

An example of a more involved complex value sort and of a value of that sort is shown in Fig. 20.1(a). The tuple constructor is denoted by $\times$ and the set constructor by $*$. An



(a) A sort and a value of that sort



(b) Another representation of the same value

Figure 20.1: Complex value

alternative representation more in the spirit of our representations of relations is shown in Fig. 20.1(b). Another complex value (for a **CINEMA** database) is shown in Fig. 20.2.

We will see that, whereas it is simple to add the tuple constructor to the traditional relational data model, the set constructor requires a number of interesting new ideas. There are similarities between this set construct and the set constructs used in general-purpose programming languages such as Setl.

In this chapter, we introduce complex values and present a many-sorted algebra and an equivalent calculus for complex values. The focus is on the use of the two constructors of complex values: tuples and (finite) sets. (Additional constructors, such as list, bags, and

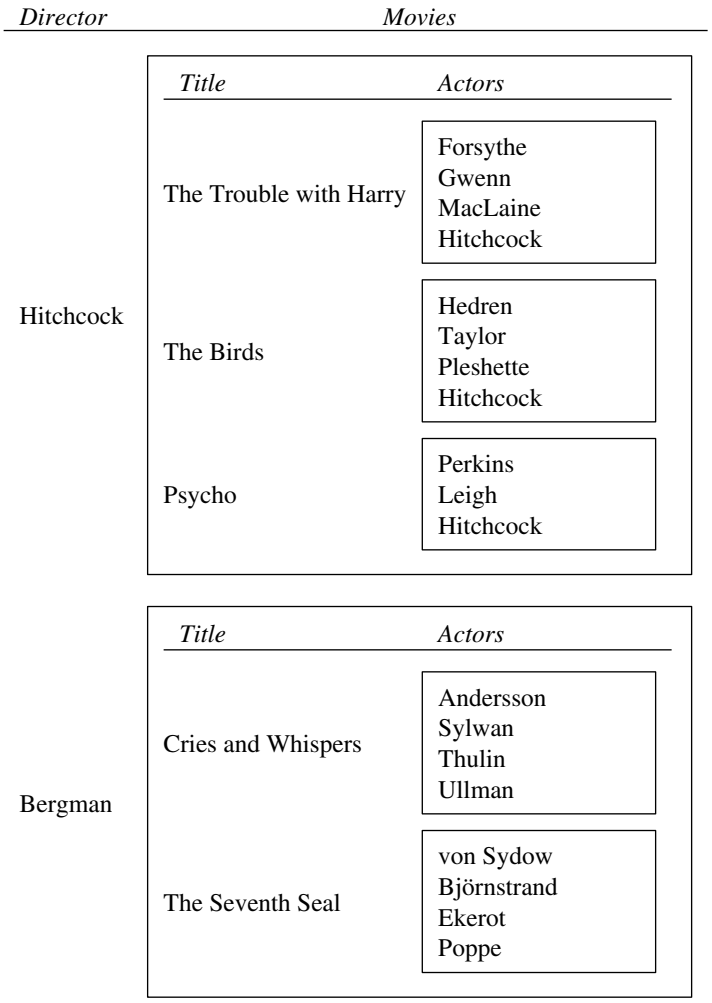


Figure 20.2: The **CINEMA** database revisited (with additional data shown)

union, have also been incorporated into complex values but are not studied here.) After introducing the algebra and calculus, we present examples of these interesting languages. We then comment on the issues of expressive power and complexity and describe equivalent languages with fixpoint operators, as well as languages in the deductive paradigm. Finally we briefly examine a subset of the commercial query language O_2SQL that provides an elegant SQL-style syntax for querying complex values.

The theory described in this chapter serves as a starting point for object-oriented databases, which are considered in Chapter 21. However, key features of the object-oriented paradigm, such as objects and inheritance, are still missing in the complex value framework and are left for Chapter 21.

20.1 Complex Value Databases

Like the relational model, we will use relation names in **relname**, attributes in **att**, and constants in **dom**. The sorts are more complex than for the relational model. Their abstract syntax is given by

$$\tau \;=\; \mathbf{dom} \;\mid\; \langle B_1 : \tau, \ldots, B_k : \tau \rangle \;\mid\; \{\tau\},$$

where $k \geq 0$ and $B_1, \ldots, B_k$ are distinct attributes. Intuitively, an element of **dom** is a constant; an element of $\langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle$ is a k-tuple with an element of sort τ_i in entry B_i for each i; and an element of sort $\{\tau\}$ is a finite set of elements of sort τ.

Formally, the set of values of sort τ (i.e., the interpretation of τ), denoted $[\![\tau]\!]$, is defined by

1. $[\![\mathbf{dom}]\!] = \mathbf{dom}$,
2. $[\![\{\tau\}]\!] = \{\{v_1, \ldots, v_j\} \mid j \geq 0, v_i \in [\![\tau]\!], i \in [1, j]\}$, and
3. $[\![\langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle]\!] = \{\langle B_1 : v_1, \ldots, B_k : v_k \rangle \mid v_j \in [\![\tau_j]\!], j \in [1, k]\}$.

An element of a sort is called a *complex value*. A complex value of the form $\langle B_1 : a_1, \ldots, B_k : a_k \rangle$ is said to be a *tuple*, whereas a complex value of the form $\{a_1, \ldots, a_j\}$ is a set.

Remark 20.1.1 For instance, consider the sort

$$\{\langle A : \mathbf{dom}, B : \mathbf{dom}, C : \{\langle A : \mathbf{dom}, E : \{\mathbf{dom}\}\rangle\}\rangle\}$$

and the value

$$\begin{array}{l} \{\ \langle A : a, B : b, C : \{\ \langle A : c, E : \{\}\rangle, \\ \qquad\qquad\qquad\qquad\ \ \langle A : d, E : \{\}\rangle\}\rangle, \\ \quad \langle A : e, B : f, C : \{\ \}\rangle\ \} \end{array}$$

of that sort. This is yet again the value of Fig. 20.1. It is customary to omit **dom** and for instance write this sort $\{\langle A, B, C : \{\langle A, E : \{\}\rangle\}\rangle\}$.

As mentioned earlier, each complex value and each sort can be viewed as a finite tree. Observe the tree representation. Outgoing edges from tuple vertexes are labeled; set vertexes have a single child in a sort and an arbitrary (but finite) number of children in a value.

Finally note that (because of the empty set) a complex value may belong to more than one sort. For instance, the value of Fig. 20.1 is also of sort

$$\{\langle A : \mathbf{dom}, B : \mathbf{dom}, C : \{\langle A : \mathbf{dom}, E : \{\{\mathbf{dom}\}\}\rangle\}\rangle\}.$$

Relational algebra deals with *sets* of tuples. Similarly, complex value algebra deals with sets of complex values. This motivates the following definition of sorted relation (this

definition is frequently a source of confusion):

A (complex value) *relation* of sort τ is a finite set of values of sort τ.

We use the term *relation* for complex value relation. When we consider the classical relational model, we sometimes use the phrase *flat relation* to distinguish it from complex value relation. It should be clear that the flat relations that we have studied are special cases of complex value relations.

We must be careful in distinguishing the sort of a complex value relation and the sort of the relation viewed as one complex value. For example, a complex value relation of sort $\langle A, B, C\rangle$ is a set of tuples over attributes ABC. At the same time, the entire relation can be viewed as one complex value of sort $\{\langle A, B, C\rangle\}$. There is no contradiction between these two ways of viewing a relation.

We now assume that the function *sort* (of Chapter 3) is from **relname** to the set of sorts. We also assume that for each sort, there is an infinite number of relations having that sort.

Note that the sort of a relation is not necessarily a tuple sort (it can be a set sort). Thus relations do not always have attributes at the top level. Such relations whose sort is a set are essentially unary relations without attribute names.

A *(complex value) schema* is a relation name; and a *(complex value) database schema* is a finite set of relation names. A *(complex value) relation* over relation name R is a finite set of values of sort $sort(R)$—that is, a finite subset of $[\![sort(R)]\!]$. A *(complex value database) instance* $\mathbf{I}$ of a schema $\mathbf{R}$ is a function from $\mathbf{R}$ such that for each R in $\mathbf{R}$, $\mathbf{I}(R)$ is a relation over R.

Example 20.1.2 To illustrate this definition, an instance $\mathbf{J}$ of $\{R_1, R_2, R_3\}$ where

$$sort(R_1) = sort(R_3) = \langle A : \mathbf{dom}, B : \{\langle A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle\}\rangle \text{ and}$$
$$sort(R_2) = \langle A : \mathbf{dom}, A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle$$

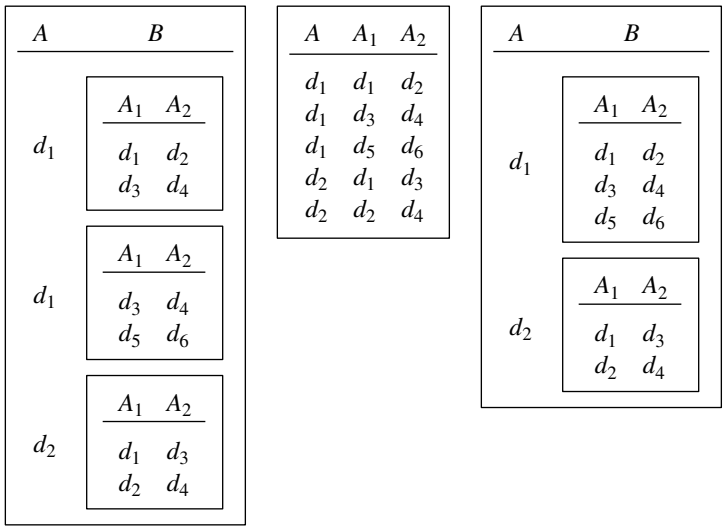
is shown in Fig. 20.3.

Variations

To conclude this section, we briefly mention some variations of the complex value model. The principal one that has been considered is the *nested relation model*. For nested relations, set and tuple constructors are required to alternate (i.e., set of sets and tuple with a tuple component are prohibited). For instance,

$$\tau_1 = \langle A, B, C : \{\langle D, E : \{\langle F, G\rangle\}\rangle\}\rangle \quad \text{and}$$
$$\tau_2 = \langle A, B, C : \{\langle E : \{\langle F, G\rangle\}\rangle\}\rangle$$

are nested relation sorts whereas



$\mathbf{J}(R_1)$ $\mathbf{J}(R_2)$ $\mathbf{J}(R_3)$

Figure 20.3: A database instance

$$\tau_3 = \langle A, B, C : \langle D, E : \{\langle F, G\rangle\}\rangle\rangle \quad \text{and}$$
$$\tau_4 = \langle A, B, C : \{\{\langle F, G\rangle\}\}\rangle$$

are not. (For τ_3, observe two adjacent tuple constructors; there are two set constructors for τ_4.)

The restriction imposed on the structure of nested relations is mostly cosmetic. A more fundamental constraint is imposed in so-called Verso-relations (V-relations).

As with nested relations, set and tuple constructors in V-relations are required to alternate. A relation is defined recursively to be a set of tuples, such that each component may itself be a relation but at least one of them must be atomic. The foregoing sort τ_1 would be acceptable for a V-relation whereas sort τ_2 would not because of the sort of tuples in the C component.

A further (more radical) assumption for V-relations is that for each set of tuples, the atomic attributes form a key. Observe that as a consequence, the cardinality of each set in a V-relation is bounded by a polynomial in the number of atomic elements occurring in the V-relation. This bound certainly does not apply for a relation of sort {**dom**} (a set of sets) or for a nested relation of sort

$$\langle A : \{\langle B : \mathbf{dom}\rangle\}\rangle,$$

which is also essentially a set of sets. The V-relations are therefore much more limited data structures. (See Exercise 20.1.) They can be viewed essentially as flat relational instances.

20.2 The Algebra

We now define a many-sorted algebra, denoted ALG^{cv} (for *c*omplex *v*alues). Like relational algebra, ALG^{cv} is a functional language based on a small set of operations. This section first presents a family of core operators of the algebra and then an extended family of operators that can be simulated by them. At the end of the section we introduce an important subset of ALG^{cv}, denoted ALG^{cv-}.

The Core of ALG^{cv}

Let $I, I_1, I_2, \ldots$ be relations of sort $\tau, \tau_1, \tau_2, \ldots$ respectively. It is important to keep in mind that a relation of sort τ is a *set* of values of sort τ.

Basic set operations: If $\tau_1 = \tau_2$, then $I_1 \cap I_2$, $I_1 \cup I_2$, $I_1 - I_2$, are relations of sort τ_1, and their values are defined in the obvious manner.

Tuple operations: If I is a relation of sort $\tau = \langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle$, then

- $\sigma_\gamma(I)$ is a relation of sort τ.
  The selection condition γ is (with obvious restrictions on sorts) of the form $B_i = d$, $B_i = B_j$, $B_i \in B_j$ or $B_i = B_j.C$, where d is a constant, and it is required in the last case that τ_j be a tuple sort with a C field. Then

$$\sigma_\gamma(I) = \{v \mid v \in I, v \models \gamma\},$$

  where $\models$ is defined by

  - $\langle \ldots, B_i : v_i, \ldots \rangle \models B_i = d$ if $v_i = d$,
  - $\langle \ldots, B_i : v_i, \ldots, B_j : v_j, \ldots \rangle \models B_i = B_j$ if $v_i = v_j$, and
  - $\langle \ldots, B_i : v_i, \ldots, B_j : v_j, \ldots \rangle \models B_i \in B_j$ if $v_i \in v_j$.
  - $\langle \ldots, B_i : v_i, \ldots, B_j : \langle \ldots, C : v_j, \ldots \rangle, \ldots \rangle \models B_i = B_j.C$ if $v_i = v_j$.

- $\pi_{B_1,\ldots,B_l}(I), l \leq k$ is a relation of sort $\langle B_1 : \tau_1, \ldots, B_l : \tau_l \rangle$ with

$$\pi_{B_1,\ldots,B_l}(I) = \{ \langle B_1 : v_1, \ldots, B_l : v_l \rangle \mid \exists v_{l+1}, \ldots, v_k (\langle B_1 : v_1, \ldots, B_k : v_k \rangle \in I)\}.$$

Constructive operations

- *powerset*(I) is a relation of sort $\{\tau\}$ and

$$powerset(I) = \{v \mid v \subseteq I\}.$$

- If $A_1, \ldots, A_n$ are distinct attributes, $tup\_create_{A_1 \ldots A_n}(I_1, \ldots, I_n)$ is of sort $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$, and

$$tup\_create_{A_1,\ldots,A_n}(I_1, \ldots, I_n) = \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle \mid \forall i \ (v_i \in I_i)\}.$$

- *set_create*(I) is of sort $\{\tau\}$, and *set_create*$(I) = \{I\}$.

Destructive operations

- If $\tau = \{\tau'\}$, then *set_destroy*(I) is a relation of sort τ' and

$$set\_destroy(I) = \cup I = \{w \mid \exists v \in I, w \in v\}.$$

- If I is of sort $\langle A : \tau' \rangle$, *tup_destroy*$(I)$ is a relation of sort τ', and

$$tup\_destroy(I) = \{v \mid \langle A : v \rangle \in I\}.$$

We are now prepared to define the (core of the) language ALG^{cv}. Let $\mathbf{R}$ be a database schema. A query returns a set of values of the same sort. By analogy with relations, a query of sort τ returns a *set* of values of sort τ. ALG^{cv} queries and their answers are defined as follows. There are two base cases:

Base values: For each relation name R in $\mathbf{R}$, R is an algebraic query of sort *sort*(R). The answer to query R is $\mathbf{I}(R)$.

Constant values: For each element a, $\{a\}$ is a (constant) algebraic query of sort **dom**. The answer to query $\{a\}$ is simply $\{a\}$.

Other queries of ALG^{cv} are obtained as follows. If $q_1, q_2, \ldots$ are queries, γ is a selection condition, and $A_1, \ldots$ are attributes,

$q_1 \cap q_2$,	$q_1 \cup q_2$,	$q_1 - q_2$,
$\sigma_\gamma(q_1)$,	$\pi_{A_1,\ldots,A_k}(q_1)$,	$tup\_create_{A_1,\ldots,A_k}(q_1, \ldots, q_k)$,
$powerset(q_1)$,	$tup\_destroy(q_1)$,	$set\_destroy(q_1)$,
$set\_create(q_1)$		

are queries if the appropriate restrictions on the sorts apply. (Note that because of the sorting constraints, *tup_destroy* and *set_destroy* cannot both be applicable to a given q_1.) The sort of a query and its answer are defined in a straightforward manner.

To illustrate these definitions, we present two examples. We then consider other algebraic operators that are expressible in the algebra. In Section 20.4 we provide several more examples of algebraic queries.

Example 20.2.1 Consider the instance $\mathbf{J}$ of Fig. 20.3. Then one can find in Fig. 20.4

$$\begin{array}{ll} J_1 = [\sigma_{A=d_2}(R_1)](\mathbf{J}), & J_2 = \pi_B(J_1), \\ J_3 = tup\_destroy(J_2), & J_4 = set\_destroy(J_3), \\ J_5 = powerset(J_4), & J_6 = tup\_create_C(J_4). \end{array}$$

Also observe that

$$J_5 = [powerset(set\_destroy(tup\_destroy(\pi_B(\sigma_{A=d_2}(R_1)))))](\mathbf{J}).$$

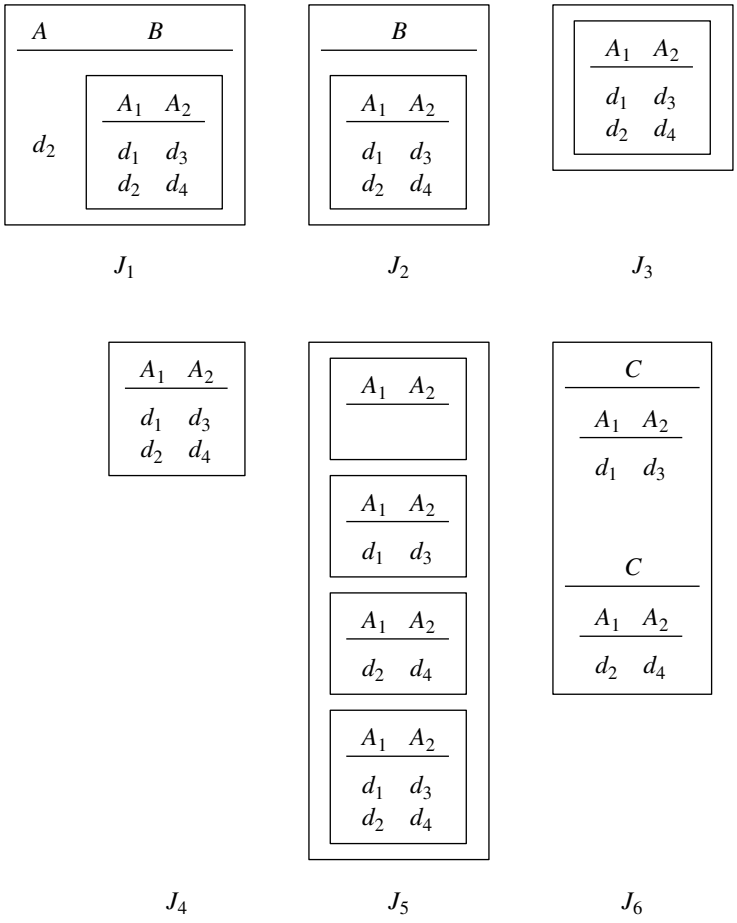


Figure 20.4: Algebraic operations

EXAMPLE 20.2.2 In this example, we illustrate the destruction and construction of a complex value. Consider the relation

$$I = \{\langle A : a, B : \{b, c\}, C : \langle A : d, B : \{e, f\}\rangle\rangle\}.$$

Then

$$\begin{aligned}
&[(\pi_A \circ \mathit{tup_destroy}) \\
&\quad \cup\ (\pi_B \circ \mathit{tup_destroy} \circ \mathit{set_destroy}) \\
&\quad \cup\ (\pi_C \circ \mathit{tup_destroy} \circ \pi_A \circ \mathit{tup_destroy}) \\
&\quad \cup\ (\pi_C \circ \mathit{tup_destroy} \circ \pi_B \circ \mathit{tup_destroy} \circ \mathit{set_destroy})](I) \\
&\quad = \{a, b, c, d, e, f\}.
\end{aligned}$$

We next reconstruct I from singleton sets:

$$I = tup\_create_{A,B,C}(\{a\}, set\_create(\{b\} \cup \{c\}),$$
$$tup\_create_{A,B}(\{d\}, set\_create(\{e\} \cup \{f\}))).$$

Additional Algebraic Operations

There are infinite possibilities in the choice of algebraic operations for complex values. We chose to incorporate in the core algebra only a few basic operations to simplify the formal presentation and the proof of the equivalence between the algebra and calculus. However, making the core *too* reduced would complicate that proof. (For example, the operator *set_create* can be expressed using the other operations but is convenient in the proof.) We now present several additional algebraic operations. It is important to note that all these operations can be expressed in complex value algebra. (In that sense, they can be viewed as macro operations.) Furthermore, all but the *nest* operator can be expressed without using the powerset operator.

We first generalize constant queries.

Complex constants: It is easy to see that the technique of Example 20.2.2 can be generalized. So instead of simply $\{a\}$ for a atomic, we use as constant queries arbitrary complex value sets.

We also generalize relational operations.

Renaming: Renaming can be computed using the other operations, as illustrated in Section 20.4 (which presents examples of queries).

Cross-product: For i in [1,2], let I_i be a relation of sort

$$\tau_i = \langle B_1^i : \tau_1^i, \ldots, B_{j_i}^i : \tau_{j_i}^i \rangle$$

and let the attribute sets in τ_1, τ_2 be disjoint. Then $I_1 \times I_2$ is the relation defined by

$$sort(I_1 \times I_2) = \langle B_1^1 : \tau_1^1, \ldots, B_{j_1}^1 : \tau_{j_1}^1, B_1^2 : \tau_1^2, \ldots, B_{j_2}^2 : \tau_{j_2}^2 \rangle$$

and

$$I_1 \times I_2 = \{ \langle B_1^1 : x_1^1, \ldots, B_{j_1}^1 : x_{j_1}^1, B_1^2 : x_1^2, \ldots, B_{j_2}^2 : x_{j_2}^2 \rangle \mid$$
$$\langle B_1^i : x_1^i, \ldots, B_{j_i}^i : x_{j_i}^i \rangle \in I_i \text{ for } i \in [1, 2] \}.$$

It is easy to simulate cross-product using the operations of the algebra. This is also illustrated in Section 20.4.

Join: This can be defined in the natural manner and can be simulated using cross-product, renaming, and selection.

It should now be clear that complex value algebra subsumes relational algebra when applied to flat relations. We also have new set-oriented operations.

N-ary set_create: We introduced *tup_create* as an n-ary operation. We also allow n-ary *set_create* with the meaning that

$$set\_create(I_1, \ldots, I_n) \equiv set\_create(I_1) \cup \cdots \cup set\_create(I_n).$$

Singleton: This operator transforms a set of values $\{a_1, \ldots, a_n\}$ into a set $\{\{a_1\}, \ldots, \{a_n\}\}$ of singletons.

Nest, unnest: Less primitive interesting operations such as *nest, unnest* can be considered. For example, for $\mathbf{J}$ of Fig. 20.3 we have

$$unnest_B(\mathbf{J}(R_1)) = \mathbf{J}(R_2) \qquad \text{and}$$
$$nest_{B=(A_1A_2)}(\mathbf{J}(R_2)) = \mathbf{J}(R_3).$$

More formally, suppose that we have R and S with sorts

$$sort(R) = \langle A_1 : \tau_1, \ldots, A_k : \tau_k, B : \{\langle A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n\rangle\}\rangle$$
$$sort(S) = \langle A_1 : \tau_1, \ldots, A_k : \tau_k, A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n\rangle.$$

Then for instances I of R and J of S, we have

$$unnest_B(I) = \{\langle A_1 : x_1, \ldots, A_n : x_n\rangle \mid \exists y$$
$$\langle A_1 : x_1, \ldots, A_k : x_k, B : y\rangle \in I \text{ and } \langle A_{k+1} : x_{k+1}, \ldots, A_n : x_n\rangle \in y\}$$
$$nest_{B=(A_{k+1},\ldots,A_n)}(J) = \{\langle A_1 : x_1, \ldots, A_k : x_k, B : y\rangle \mid$$
$$\emptyset \neq y = \{\langle A_{k+1} : x_{k+1}, \ldots, A_n : x_n\rangle \mid \langle A_1 : x_1, \ldots, A_n : x_n\rangle \in J\}\}.$$

Observe that

$$unnest_B(nest_{B=(A_1A_2)}(\mathbf{J}(R_2))) = \mathbf{J}(R_2).$$
$$nest_{B=(A_1A_2)}(unnest_B(\mathbf{J}(R_1))) \neq \mathbf{J}(R_1).$$

This is indeed not an isolated phenomenon. *Unnest* is in general the right inverse of *nest* ($nest_{B=\alpha} \circ unnest_B$ is the identity), whereas *unnest* is in general not information preserving (one-to-one) and so has no right inverse (see Exercise 20.8).

Relational projection and selection were filtering operations in the sense that intuitively they scan a set and keep only certain elements, possibly modifying them in a uniform way. The filters in complex value algebra are more general. Of course, we shall allow Boolean expressions in selection conditions. More interestingly, we also allow set comparators in addition to $\in$, such as $\ni$, $\subset$, $\subseteq$, $\supset$, $\supseteq$ and negations of these comparators (e.g., $\notin$). The inclusion comparator $\subseteq$ plays a special role in the calculus. We will see in Section 20.4 how to simulate selection with $\subseteq$.

Selection is a *predicative filter* in the sense that a predicate allows us to select some elements, leaving them unchanged. Other filters, such as projection, are *map filters*. They transform the elements. Clearly, one can combine both aspects and furthermore allow more complicated selection conditions or restructuring specifications. For instance, suppose I is

a set of tuples of sort

$$\langle A : \mathbf{dom}, B : \langle C : \langle E : \{\mathbf{dom}\}, E' : \mathbf{dom}\rangle, C' : \{\mathbf{dom}\}\rangle\rangle.$$

We could use an operation that first filters all the values matching the pattern

$$\langle A : x, B : \langle C : \langle E : y, E' : z\rangle, C' : \{x\}\rangle\rangle;$$

and then transforms them into

$$\langle A : (y \cup \{x\}), B : y, C : z\rangle.$$

This style of operations is standard in functional languages (e.g., *apply-to-all* in fp).

Remark 20.2.3 As mentioned earlier, all of the operations just introduced are expressible in ALG^{cv}. We might also consider an operation to *iterate* over the elements of a set in some order. Such an operation can be found in several systems. As we shall see in Section 20.6, iteration is essentially expressible within ALG^{cv}. On the other hand, an iteration that depends on a specific ordering of the underlying domain of elements cannot be simulated using ALG^{cv} unless the ordering is presented as part of the input. ■

In the following sections, we (informally) call *extended algebra* the algebra consisting of the operations of ALG^{cv} and allowing complex constants, renaming, cross-product, join, n-ary *set_create*, singleton, *nest*, and *unnest*.

An important subset of ALG^{cv}, denoted ALG^{cv-}, is formed from the core operators of ALG^{cv} by removing the *powerset* operator and adding the *nest* operator. As will be seen in Section 20.7, although the *nest* operator has the ability to construct sets, it is much weaker than *powerset*. When restricted to nested relations, the language ALG^{cv-} is usually called *nested relation algebra*.

20.3 The Calculus

The calculus is modeled after a standard, first-order, many-sorted calculus. However, as we shall see, calculus variables may denote sets, so the calculus will permit quantification over sets (something normally considered to be a second-order feature). For complex value calculus, the separation between first and second order (and higher order as well) is somewhat blurred. As with the algebra, we first present a core calculus and then extend it. The issues of domain independence and safety are also addressed.

For each sort, we assume the existence of a countably infinite set of variables of that sort. A variable is *atomic* if it ranges over the sort **dom**. Let **R** be a schema. A *term* is an atomic element, a variable, or an expression $x.A$, where x is a tuple variable and A is an attribute of x. We do not consider (yet) fancier terms. A *positive literal* is an expression of the form

$$R(t), \quad t = t', \quad t \in t', \quad \text{or} \quad t \subseteq t',$$

where $R \in \mathbf{R}$, t, t' are terms and the appropriate sort restrictions apply.[1] *Formulas* are defined from atomic formulas using the standard connectives and quantifiers: $\wedge, \vee, \neg, \forall, \exists$. A *query* is an expression $\{x \mid \varphi\}$, where formula φ has exactly one free variable (i.e. x). We sometimes denote it by $\varphi(x)$. The calculus is denoted CALC^{cv}.

The following example illustrates this calculus.

Example 20.3.1 Consider the schema and the instance of Fig. 20.3. We can verify that $\mathbf{J}(R_2)$ is the answer on instance $\mathbf{J}$ to the query

$$\begin{aligned}\{x \mid \exists y, z, z', u, v, w \;\; & (R_1(y) \wedge y.A = u \wedge y.B = z \\ & \wedge z' \in z \wedge z'.A_1 = v \wedge z'.A_2 = w \\ & \wedge x.A = u \wedge x.A_1 = v \wedge x.A_2 = w) \;\},\end{aligned}$$

where the sorts of the variables are as follows:

$$\begin{aligned} &sort(x) = \langle A, A_1, A_2\rangle, && sort(y) = \langle A, B : \{\langle A_1, A_2\rangle\}\rangle, \\ &sort(u) = sort(v) = sort(w) = \mathbf{dom}, && sort(z') = \langle A_1, A_2\rangle, \\ &sort(z) = \{\langle A_1, A_2\rangle\}. \end{aligned}$$

We could also have used an unsorted alphabet of variables and sorted them inside the formula, as in

$$\begin{aligned}\{x : \langle A, A_1, A_2\rangle \;\; \mid \;\; & \exists y : \langle A, B : \{\langle A_1, A_2\rangle\}\rangle, \\ & z : \{\langle A_1, A_2\rangle\}, z' : \langle A_1, A_2\rangle, \\ & u : \mathbf{dom}, v : \mathbf{dom}, w : \mathbf{dom} \\ & (R_1(y) \wedge y.A = u \wedge y.B = z \\ & \wedge z' \in z \wedge z'.A_1 = v \wedge z'.A_2 = w \\ & \wedge x.A = u \wedge x.A_1 = v \wedge x.A_2 = w) \;\}.\end{aligned}$$

The key difference with relational calculus is the presence of the predicates $\in$ and $\subseteq$, which are interpreted as the standard set membership and inclusion. Another difference (of a more cosmetic nature) is that we allow only one free variable in relation atoms and in query formulas. This comes from the stronger sorts: A variable may represent an n-tuple.

The *answer* to a query q on an instance $\mathbf{I}$, denoted $q(\mathbf{I})$, is defined as for the relational model. As in the relational case, we may define various interpretations, depending on the underlying domain of base values used. As with relational calculus, the basis for defining the semantics is the notion

$$\mathbf{I} \text{ satisfies } \varphi \text{ for } \nu \text{ relative to } \mathbf{d}.$$

---

[1] Strictly speaking, the symbols $=, \subseteq$ and $\in$ are also many sorted.

[Recall that ν is a valuation of the free variables of φ and $\mathbf{d}$ is an arbitrary set of elements containing $adom(\varphi, \mathbf{I})$.]

Consider the definition of this notion in Section 5.3. Cases (a) through (g) remain valid for the complex object calculus. We have to consider two supplementary cases. Recall that for equality, we had case (b):

(b) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s = s')$ and $\nu(s) = \nu(s')$.

In the same spirit, we add

(h-1) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s \in s')$ and $\nu(s) \in \nu(s')$

(h-2) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s \subseteq s')$ and $\nu(s) \subseteq \nu(s')$.

This formally states that $\in$ is interpreted as set membership and $\subseteq$ as set inclusion (in the same sense that as $=$ is interpreted as equality).

The issues surrounding domain independence for relational calculus also arise with CALC^{cv}. We develop a syntactic condition ensuring domain independence, but we also occasionally use an active domain interpretation.

Extensions

As in the case of the algebra, we now consider extensions of the calculus that can be simulated by the core syntax just given.

The standard abbreviations used for relational calculus, such as the logical connectives $\rightarrow$, $\leftarrow$, $\leftrightarrow$, can be incorporated into CALC^{cv}. Using these connectives, it is easy to see the nonminimality of the calculus: Each literal $x \subseteq y$ can be replaced by $\forall z(z \in x \rightarrow z \in y)$, where z is a fresh variable.

Arity In the core calculus, only relation atoms of the form $R(t)$ are permitted. Suppose that the sort of R is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ for some n. Then $R(u_1, \ldots, u_n)$ is a shorthand for

$$\exists y(R(y) \wedge y.A_1 = u_1 \wedge \cdots \wedge y.A_n = u_n),$$

where y is a new variable. In particular, if R_0 is a relation of sort $\langle\ \rangle$ $(n = 0)$, observe that the only value of that sort is the empty tuple. Thus a variable y of that sort has only one possible value, namely $\langle\ \rangle$. Thus for such y, we can use the following expression:

$$R_0(\,) \quad \text{for} \quad \exists y(R_0(y)).$$

Constructed Terms Next we allow constructed terms in the calculus such as

$$\{x, b\}, \quad x.A.C, \quad \langle B_1 : a, B_2 : y \rangle.$$

More formally, if $t_1, \ldots, t_k$ are terms and $B_1, \ldots, B_k$ are distinct attributes, then $\langle B_1 : t_1, \ldots, B_k : t_k \rangle$ is a term. Furthermore, if the t_i are of the same sort, $\{t_1, \ldots, t_k\}$ is a term;

and if t_1 is a tuple term with attribute C, then $t_1.C$ is a term. The sorts of terms are defined in the obvious way. Note that a term may have several sorts because of the empty set. (We ignore this issue here.)

The use of constructed terms can be viewed as syntactic sugaring. For instance, suppose that the term $\{a, y\}$ occurs in a formula ψ. Then ψ is equivalent to

$$\exists x(\psi' \wedge \forall z(z \in x \leftrightarrow (z = a \vee z = y))),$$

where ψ' is obtained from ψ by replacing the term $\{a, y\}$ by x (a fresh variable).

Complex Terms We can also view relations as terms. For instance, if R is a relation of sort $\langle A, B\rangle$, then R can be used in the language as a term of sort $\{\langle A, B\rangle\}$. We may then consider literals such as $x \in R$, which is equivalent to $R(x)$; or more complex ones such as $S \in T$, which essentially means

$$\exists y(T(y) \wedge \forall x(x \in y \leftrightarrow S(x))).$$

The previous extension is based on the fact that a relation (in our context) can be viewed as a complex value. This is again due to the stronger sort system. Now the answer to a query q is also a complex value. This suggests considering the use of queries as terms of the language. We consider this now: A query $q \equiv \{y \mid \psi(y)\}$ is a legal term that can be used in the calculus like any other term. More generally, we allow terms of the form

$$\{y \mid \psi(y, y_1, \ldots, y_n)\},$$

where the free variables of ψ are $y, y_1, \ldots, y_n$. Intuitively, we obtain queries by providing bindings for $y_1, \ldots, y_n$. We will call such an expression a *parameterized query* and denote it $q(y_1, \ldots, y_n)$ (where $y_1, \ldots, y_n$ are the parameters).

For instance, suppose that a formula $liked(x, y)$ computes the films y that person x liked; and another one $saw(x, y)$ computes those that x has seen. The set of persons who liked all the films that they saw is given by

$$\{\, x \mid \{y \mid liked(x, y)\} \subseteq \{y \mid saw(x, y)\} \,\}.$$

The following form of literals will play a particular role when we study safety for this calculus:

$$\begin{aligned} x &= \{y \mid \psi(y, y_1, \ldots, y_n)\}, \\ x' &\in \{y \mid \psi(y, y_1, \ldots, y_n)\}, \text{ and} \\ x'' &\subseteq \{y \mid \psi(y, y_1, \ldots, y_n)\}, \end{aligned}$$

where y is a free variable of ψ. Like the previous extensions, the parameterized queries can be viewed simply as syntactic sugaring. For instance, the three last formulas are, respectively, equivalent to

$$\forall y(y \in x \leftrightarrow \psi),$$
$$\exists y(x' = y \wedge \psi), \text{ and}$$
$$\forall y(y \in x'' \rightarrow \psi).$$

In the following sections, we (informally) call *extended calculus* the calculus consisting of CALC^{cv} extended with the abbreviations described earlier (such as constructed and complex terms and, notably, parameterized queries).

20.4 Examples

We illustrate the previous two sections with a series of examples. The queries in the examples apply to schema $\{R, S\}$ with

$$sort(R) = \langle A : \mathbf{dom}, A' : \mathbf{dom}\rangle,$$
$$sort(S) = \langle B : \mathbf{dom}, B' : \{\mathbf{dom}\}\rangle.$$

For each query, we give an algebraic and a calculus expression.

Example 20.4.1 The union of R and a set of two constant tuples is given by

$$\{r \mid R(r) \vee r = \langle A : 3, A' : 5\rangle \vee r = \langle A : 0, A' : 0\rangle\}$$

or

$$R \cup \{\langle A : 3, A' : 5\rangle, \langle A : 0, A' : 0\rangle\}.$$

Example 20.4.2 The selection of the tuples from S, where the first component is a member of the second component, is obtained with

$$\{s \mid S(s) \wedge s.B \in s.B'\} \quad \text{or} \quad \sigma_{B \in B'}(S).$$

Example 20.4.3 The (classical) cross-product of R and S is the result of

$$\{t \mid \exists r, s(R(r) \wedge S(s) \wedge t = \langle A : r.A, A' : r.A', B : s.B, B' : s.B'\rangle)\}$$

or

$$\pi_{AA'BB'}(\sigma_{A=A''.A}(\sigma_{A'=A''.A'}(\sigma_{B=B''.B}(\sigma_{B'=B''.B'}(q))))),$$

where q is

$$tup\_create_{AA'BB'A''B''}(tup\_destroy(\pi_A(R)),\ tup\_destroy(\pi_{A'}(R)),\ tup\_destroy(\pi_B(S)),\ tup\_destroy(\pi_{B'}(S)),\ R,\ S).$$

Example 20.4.4 The join of R and S on $A = B$. This query is the composition of the cross-product of Example 20.4.3, with a selection. In Example 20.4.3, let the formula describing the cross-product be φ_3 and let $(R \times S)$ be the algebraic expression. Then the $(A = B)$ join of R and S is expressed by

$$\{t \mid \varphi_3(t) \wedge t.A = t.B\} \quad \text{or} \quad \sigma_{A=B}(R \times S).$$

Example 20.4.5 The renaming of the attributes of R to A_1, A_2 is obtained in the calculus by

$$\{t \mid \exists r(R(r) \wedge t.A_1 = r.A \wedge t.A_2 = r.A')\}$$

with t of sort $\langle A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle$. In the algebra, it is given by

$$\pi_{A_1A_2}(\sigma_{A_0.A=A_1}(\sigma_{A_0.A'=A_2}(tup\_create_{A_0A_1A_2}(R, tup\_destroy(\pi_A(R)), tup\_destroy(\pi_{A'}(R)))))).$$

Example 20.4.6 Flattening S means producing a set of flat tuples, each of which contains the first component of a tuple of S and one of the elements of the second component. This is the unnest operation $unnest_{B'}(\cdot)$ in the extended algebra, or in the calculus

$$\{t \mid \exists s(S(s) \wedge t.B = s.B \wedge t.C \in s.B')\},$$

where t is of sort $\langle B, C\rangle$. In the core algebra, this is slightly more complicated. We first obtain the set of values occurring in the B' sets using

$$E_1 = tup\_create_C(set\_destroy(tup\_destroy(\pi_{B'}(S)))).$$

We can next compute $(E_1 \times S)$ (using the same technique as in Example 20.4.3). Then the desired query is given by

$$\pi_{BC}(\sigma_{C \in B'}(E_1 \times S)).$$

Flattening can be extended to sorts with arbitrary nesting depth.

Example 20.4.7 The next example is a selection using $\subseteq$. Consider a relation T of sort $\langle C : \{\mathbf{dom}\}, C' : \{\mathbf{dom}\}\rangle$. We want to express the query

$$\{t \mid T(t) \wedge t.C \subseteq t.C'\}$$

in the algebra. We do this in stages:

$$F_1 = \sigma_{C'' \in C}(T \times tup\_create_{C''}(set\_destroy(tup\_destroy(\pi_C(T))))),$$
$$F_2 = \sigma_{C'' \in C'}(F_1),$$
$$F_3 = F_1 - F_2,$$
$$F_4 = T - \pi_{CC'}(F_3).$$

Observe that

1. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_1 if $\langle C : U, C' : V\rangle$ is in T and u is in U.
2. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_2 if $\langle C : U, C' : V\rangle$ is in T and u is in U and V.
3. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_3 if $\langle C : U, C' : V\rangle$ is in T and u is in $U - V$.
4. A tuple $\langle C : U, C' : V\rangle$ is in F_4 if it is in T and there is no u in $U - V$ (i.e., $U \subseteq V$).

Example 20.4.8 This example illustrates the use of nesting and of sets. Consider the algebraic query

$$nest_{C=(A)} \circ nest_{C'=(A')} \circ \sigma_{C=C'} \circ unnest_C \circ unnest_{C'}(R).$$

It is expressed in the calculus by

$$\{\langle x, y\rangle \mid \exists u(x \in u \wedge y \in u \wedge u = \{x' \mid R(x', y)\} \wedge u = \{y' \mid \{x' \mid R(x', y')\} = u\})\}.$$

A consequence of Theorem 20.7.2 is that this query is expressible in relational calculus or algebra. It is a nontrivial exercise to obtain a relational query for it. (See Exercise 20.24.)

Example 20.4.9 Our last example highlights an important difference between the flat relational calculus and CALC^{cv}. As shown in Proposition 17.2.3, the flat calculus cannot express the transitive closure of a binary relation. In contrast, the following CALC^{cv} query does:

$$\{y \mid \forall x(closed(x) \wedge contains\_R(x) \rightarrow y \in x)\},$$

where

- $closed(x) \equiv$

$$\forall u, v, w(\langle A : u, A' : v\rangle \in x \wedge \langle A : v, A' : w\rangle \in x \rightarrow \langle A : u, A' : w\rangle \in x);$$

- $contains\_R(x) \equiv \forall z(R(z) \rightarrow z \in x)$;
- $sort(x) = \{sort(R)\}$, $sort(y) = sort(z) = sort(R)$; and $sort(u) = sort(v) = sort(w) = \mathbf{dom}$.

Intuitively, the formula specifies the set of pairs y such that y belongs to each binary relation x containing R and transitively closed. This construction will be revisited in Section 20.6.

20.5 Equivalence Theorems

This section presents three results that compare the complex value algebra and calculus. First we establish the equivalence of the algebra and the domain-independent calculus. Next we develop a syntactic safeness condition for the calculus and show that it does not reduce expressive power. Finally we develop a natural syntactic condition on CALC^{cv} that yields a subset equivalent to ALG^{cv-}.

Our first result is as follows:

THEOREM 20.5.1 The algebra and the domain independent calculus for complex values are equivalent.

In the sketch of the proof, we present a simulation of the core algebra by the extended calculus and the analogous simulation in the opposite direction. An important component of this proof—namely, that the extended algebra (calculus) is no stronger than the core algebra (calculus)—is left for the reader (see Exercises 20.6, 20.7, 20.8, 20.10, and 20.11).

From Algebra to Calculus

We now show that for each algebra query, there is a domain-independent calculus query equivalent to it.

Let q be a named algebra query. We construct a domain-independent query $\{x \mid \varphi_q\}$ equivalent to q. The formula φ_q is constructed by induction on subexpressions of q. For a subexpression E of q, we define φ_E as follows:

(a) E is R for some $R \in \mathbf{R}$: φ_E is $R(x)$.

(b) E is $\{a\}$: φ_E is $x = a$.

(c) E is $\sigma_\gamma(E_1)$: φ_E is $\varphi_{E_1}(x) \wedge \Gamma$, where Γ is

$$x.A_i = x.A_j \text{ if } \gamma \equiv A_i = A_j; \quad x.A_i = a \text{ if } \gamma \equiv A_i = a;$$
$$x.A_i \in x.A_j \text{ if } \gamma \equiv A_i \in A_j; \quad x.A_i = x.A_j.C \text{ if } \gamma \equiv A_i = A_j.C.$$

(d) E is $\pi_{A_{i_1},\ldots,A_{i_k}}(E_1)$: φ_E is

$$\exists y(x = \langle A_{i_1} : y.A_{i_1}, \ldots, A_{i_k} : y.A_{i_k}\rangle \wedge \varphi_{E_1}(y)).$$

(e) For the basic set operations, we have

$$\varphi_{E_1 \cap E_2}(x) = \varphi_{E_1}(x) \wedge \varphi_{E_2}(x),$$
$$\varphi_{E_1 \cup E_2}(x) = \varphi_{E_1}(x) \vee \varphi_{E_2}(x),$$
$$\varphi_{E_1 - E_2}(x) = \varphi_{E_1}(x) \wedge \neg\varphi_{E_2}(x).$$

(f) E is $powerset(E_1)$: φ_E is $x \subseteq \{y \mid \varphi_{E_1}(y)\}$.

(g) E is $set\_destroy(E_1)$: φ_E is $\exists y(x \in y \wedge \varphi_{E_1}(y))$.

(h) E is $tup\_destroy(E_1)$: φ_E is $\exists y(\langle A : x\rangle = y \wedge \varphi_{E_1}(y))$, where A is the name of the field (of y).

(i) E is $tup\_create_{A_1,\ldots,A_n}(E_1, \ldots, E_n)$: φ_E is

$$\exists y_1, \ldots, y_n(x = \langle A_1 : y_1, \ldots, A_n : y_n\rangle \wedge \varphi_{E_1}(y_1) \wedge \cdots \wedge \varphi_{E_n}(y_n)).$$

(j) E is $set\_create(E_1)$: $x = \{y \mid \varphi_{E_1}(y)\}$.

We leave the verification of this construction to the reader (see Exercise 20.13). The domain independence of the obtained calculus query follows from the fact that algebra queries are domain independent.

From Calculus to Algebra

We now show that for each domain-independent query, there is a named algebra query equivalent to it.

Let $q = \{x \mid \varphi\}$ be a domain-independent query over $\mathbf{R}$. As in the flat relational case, we assume without loss of generality that associated with each variable x occurring in q (and also variables used in the following proof) is a unique, distinct attribute A_x in **att**. We use the active domain interpretation for the query, denoted as before with a subscript *adom*.

The crux of the proof is to construct, for each subformula ψ of φ, an algebra formula E_ψ that has the property that for each input $\mathbf{I}$,

$$E_\psi(\mathbf{I}) = \{y \mid \exists x_1, \ldots, x_n(y = \langle A_{x_1} : x_1, \ldots, A_{x_n} : x_n\rangle \wedge \psi(x_1, \ldots, x_n))\}_{adom}(\mathbf{I}),$$

where $x_1, \ldots, x_n$ is a listing of $free(\psi)$.

This construction is accomplished in three stages.

Computing the Active Domain The first step is to construct an algebra query E_{adom} having sort **dom** such that on input instance $\mathbf{I}$, $E_{adom}(\mathbf{I}) = adom(q, \mathbf{I})$. The construction of E_{adom} is slightly more intricate than the similar construction for the relational case. We prove by induction that for each sort τ, there exists an algebra operation F_τ that maps a set I of values of sort τ to $adom(I)$. This induction was not necessary in the flat case because the base relations had fixed depth. For the base case (i.e., $\tau = \mathbf{dom}$), it suffices to use for F_τ an identity operation (e.g., $tup\_create_A \circ tup\_destroy$). For the induction, the following cases occur:

1. τ is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ for $n \geq 2$. Then F_τ is

$$F_{\langle A_1:\tau_1 \rangle}(\pi_{A_1}) \cup \cdots \cup F_{\langle A_n:\tau_n \rangle}(\pi_{A_n}).$$

2. τ is $\langle A_1 : \tau_1 \rangle$. Then F_τ is $F_{\tau_1}(\mathit{tup\_destroy})$.
3. τ is $\{\tau_1\}$. Then F_τ is $F_{\tau_1}(\mathit{set\_destroy})$.

Now consider the schema $\mathbf{R}$. Then for each R in $\mathbf{R}$, $F_{sort(R)}$ maps a relation I over R to $adom(I)$. Thus $adom(q, \mathbf{I})$ can be computed with the query

$$E_{adom} = F_{sort(R_1)}(R_1) \cup \cdots \cup F_{sort(R_m)}(R_m) \cup \{a_1\} \cup \cdots \cup \{a_p\},$$

where $R_1, \ldots, R_m$ is the list of relations in $\mathbf{R}$ and $a_1, \ldots, a_p$ is the list of elements occurring in q.

Constructing Complex Values In the second stage, we prove by induction that for each sort τ, there exists an algebra query G_τ that constructs the set of values I of sort τ such that $adom(I) \subseteq adom(q, \mathbf{I})$. For $\tau = \mathbf{dom}$, we can use E_{adom}. For the induction, two cases occur:

1. τ is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$. Then G_τ is $\mathit{tup\_create}_{A_1,\ldots,A_n}(G_{\tau_1}, \ldots, G_{\tau_n})$.
2. τ is $\{\tau_1\}$. Then G_τ is $\mathit{powerset}(G_{\tau_1})$.

Last Stage We now describe the last stage, an inductive construction of the queries E_ψ for subformulas ψ of φ. We assume without loss of generality that the logical connectives $\vee$ and $\forall$ do not occur in φ. The proof is similar to the analogous proof for the flat case. We also assume that relation atoms in φ do not contain constants or repeated variables. We only present the new case (the standard cases are left as Exercise 20.13). Let ψ be $x \in y$. Suppose that x is of sort τ, so y is of sort $\{\tau\}$. The set of values of sort τ (or $\{\tau\}$) within the active domain is returned by query G_τ, or $G_{\{\tau\}}$. The query

$$\sigma_{A_x \in A_y}(\mathit{tup\_create}_{A_x,A_y}(G_\tau, G_{\{\tau\}}))$$

returns the desired result.

Observe that with this construction, E_φ returns a set of tuples with a single attribute A_x. The query q is equivalent to $\mathit{tup\_destroy}(E_\varphi)$.

As we did for the relational model, we can define a variety of syntactic restrictions of the calculus that yield domain-independent queries. We consider such restrictions next.

Safe Queries

We now turn to the development of syntactic conditions, called safe range, that ensure domain independence. These conditions are reminiscent of those presented for relational calculus in Chapter 5. As we shall see, a variant of safe range, called strongly safe range, will yield a subset of CALC^{cv}, denoted CALC^{cv-}, that is equivalent to ALG^{cv-}.

We could define safe range on the core calculus. However, such a definition would be cumbersome. A much more elegant definition can be given using the extended calculus. In particular, we consider here the calculus augmented with (1) constructed terms and (2) parameterized queries.

Recall that intuitively, if a formula is safe range, then each variable is bounded, in the sense that it is restricted by the formula to lie within the active domain of the query or the input. We now define the notions of safe formulas and safe terms. To give these definitions, we define the set of *safe-range variables* of a formula using the following procedure, which returns either the symbol $\perp$ (which indicates that some quantified variable is not bounded) or the set of free variables that are bounded. In this discussion, we consider only formulas in which universal quantifiers do not occur.

In the following procedure, if several rules are applicable, the one returning the largest set of safe-range variables (which always exists) is chosen.

procedure *safe-range* (sr)

input: a calculus formula φ

output: a subset of the free variables of φ or $\perp$. (In the following, for each Z, $\perp \cup Z = \perp \cap Z = \perp - Z = Z - \perp = \perp$.)

begin
(*pred* is a predicate in $\{=, \in, \subseteq\}$)
if for some parameterized query $\{x \mid \psi\}$ occurring as a term in φ, $x \notin sr(\psi)$ **then return** $\perp$
case φ **of**

$R(t)$	: $sr(\varphi) = free(t)$;
$(t\ pred\ t' \wedge \psi)$	: **if** ψ is safe and $free(t') \subseteq free(\psi)$ **then** $sr(\varphi) = free(t) \cup free(\psi)$;
$t\ pred\ t'$	: **if** $free(t') = sr(t')$ **then** $sr(\varphi) = free(t') \cup free(t)$; **else** $sr(\varphi) = \emptyset$;
$\varphi_1 \wedge \varphi_2$	: $sr(\varphi) = sr(\varphi_1) \cup sr(\varphi_2)$;
$\varphi_1 \vee \varphi_2$	: $sr(\varphi) = sr(\varphi_1) \cap sr(\varphi_2)$;
$\neg\varphi_1$	: $sr(\varphi) = \emptyset$;
$\exists x \varphi_1$	: **if** $x \in sr(\varphi_1)$ **then** $sr(\varphi) = sr(\varphi_1) - \{x\}$ **else return** $\perp$

end;

We say that a formula φ is *safe* if $sr(\varphi) = free(\varphi)$; and a query q is safe if its associated formula is safe.

It is important to understand how new sets are created in a safe manner. The next example illustrates two essential techniques for such creation.

EXAMPLE 20.5.2 Let R be a relation of sort $\langle A, B\rangle$. The powerset of R can be obtained in a safe manner with the query

$$\{x \mid x \subseteq \{y \mid R(y)\}\}.$$

For $\{y \mid R(y)\}$ is clearly a safe query (by the first case). Now letting $t \equiv x$, $t' \equiv \{y \mid R(y)\}$, the formula is safe (by the third case).

Now consider the nesting of the B column of R. It is achieved by the following query:

$$\{x \mid x = \langle z, \{y \mid R(z, y)\}\rangle \wedge \exists y'(R(z, y'))\}.$$

Let $t \equiv x$, $t' \equiv \langle z, \{y \mid R(z, y)\}\rangle$ and $\psi \equiv \exists y'(R(z, y'))$. First note that $sr(R(z, y))$ contains y, so the parameterized query $\{y \mid R(z, y)\}$ can be used safely. Next the formula ψ is safe. Finally the only free variable in t' is z, which is also free in ψ. Thus x is safe range (by the second case) and the query is safe.

As detailed in Section 20.7, the complex value algebra and calculus can express mappings with complexity corresponding to arbitrarily many nestings of exponentiation. In contrast, as discussed in that section, the nested relation algebra ALG^{cv-}, which uses the nest operator but not powerset, has complexity in PTIME. Interestingly, there is a minor variation of the safe-range condition that yields a subset of the calculus equivalent to ALG^{cv-}. Specifically, a formula is *strongly safe range* if it is safe range and the inclusion predicate does not occur in it. In the previous example, the nesting is strongly safe range whereas *powerset* is not.

We now have the following:

THEOREM 20.5.3

(a) The safe-range calculus, the domain-independent calculus, and ALG^{cv} coincide.

(b) The strongly safe-range calculus and ALG^{cv-} coincide.

Crux Consider (a). By inspection of the construction in the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$, each algebra query is equivalent to a safe-range calculus query. Clearly, each safe-range calculus query is a domain-independent calculus query. We have already shown that each domain-independent calculus query is an algebra query.

Now consider (b). Observe that in the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$, $\subseteq$ is used only for powerset. Thus each query in ALG^{cv-} is a strongly safe-range query. Now consider a strongly safe-range query; we construct an equivalent algebra query. We cannot use the construction from the proof of the equivalence theorem, because *powerset* is crucial for constructing complex domains. However, we can show that this can be avoided using the ranges of variables. (See Exercise 20.16.) More precisely, the brute force construction of the domain of variables using powerset is replaced by a careful construction based on the strongly safe-range restriction. The remainder of the proof stays unchanged. ■

Because of part (b) of the previous result, we denote the strongly safe-range calculus by CALC^{cv-}.

20.6 Fixpoint and Deduction

Example 20.4.9 suggests that the complex value algebra and calculus can simulate iteration. In this section, we examine iteration in the spirit of both fixpoint queries and datalog. In both cases, they do not increase the expressive power of the algebra or calculus. However, they allow us to express certain queries more efficiently.

Fixpoint for Complex Values

Languages with fixpoint semantics were considered in the context of the relational model to overcome limitations of relational algebra and calculus. In particular, we observed that transitive closure cannot be computed in relational calculus. However, as shown by Example 20.4.9, transitive closure can be expressed in the complex value algebra and calculus. Although transitive closure can be expressed in that manner, the use of *powerset* seems unnecessarily expensive. More precisely, it can be shown that *any* query in the complex value algebra and calculus that expresses transitive closure uses exponential space (assuming the straightforward evaluation of the query). In other words, the blowup caused by the *powerset* operator cannot be avoided. On the other hand, a fixpoint construct allows us to express transitive closure in polynomial space (and time). It is thus natural to develop fixpoint extensions of the calculus and algebra.

We can provide inflationary and noninflationary extensions of the calculus with recursion. As in the relational case, an *inflationary fixpoint operator* μ_T^+ allows the iteration of a CALC^{cv} formula $\varphi(T)$ up to a fixpoint. This essentially permits the inductive definition of relations, using calculus formulas. The calculus CALC^{cv} augmented with the inflationary fixpoint operator is defined similarly to the flat case (Chapter 14) and yields $\text{CALC}^{cv} + \mu^+$. We only consider the inflationary fixpoint operator. (Exercise 20.19 explores the noninflationary version.)

Theorem 20.6.1 $\text{CALC}^{cv} + \mu^+$ is equivalent to ALG^{cv} and CALC^{cv}.

The proof of this theorem is left for Exercise 20.18. It involves simulating a fixpoint in a manner similar to Example 20.4.9.

Before leaving the fixpoint extension, we show how powerset can be computed by iterating a ALG^{cv-} formula to a fixpoint. (We will see later that powerset cannot be computed in ALG^{cv-} alone.)

Example 20.6.2 Consider a relation R of sort **dom** (i.e., a set of atomic elements). The powerset of R is computed by $\{x \mid \mu_T(\varphi(T))(x)\}$, where T is of sort $\{\textbf{dom}\}$ and

$$\varphi(T)(y) \equiv [y = \emptyset \vee \exists x', y'(R(x') \wedge T(y') \wedge y = y' \cup \{x'\}.]$$

This formula is in fact equivalent to a query in ALG^{cv-}. (See Exercise 20.15.) For example, suppose that R contains $\{2, 3, 4\}$. The iteration of φ yields

$$\begin{array}{lll} J_0 & = \emptyset & \\ J_1 & = \varphi(\emptyset) & = \{\emptyset\} \\ J_2 & = \varphi(J_1) & = J_1 \cup \{\{2\}, \{3\}, \{4\}\} \\ J_3 & = \varphi(J_2) & = J_2 \cup \{\{2, 3\}, \{2, 4\}, \{3, 4\}\} \\ J_4 & = \varphi(J_3) & = J_3 \cup \{\{2, 3, 4\}\}, \end{array}$$

and J_4 is a fixpoint and coincides with $powerset(\{2, 3, 4\})$.

Datalog for Complex Values

We now briefly consider an extension of datalog to incorporate complex values. The basic result is that the extension is equivalent to the complex value algebra and calculus. We also consider a special grouping construct, which can be used for set construction in this context.

In the datalog extension considered here, the predicates $\subseteq$ and $\in$ are permitted. A rule is *safe range* if each variable that appears in the head also appears in the body, and the body is safe (i.e., the conjunction of the literals of the body is a safe formula). We assume henceforth that rules are safe. Stratified negation will be used. The language is illustrated in the following example.

EXAMPLE 20.6.3 The input is a relation R of sort $\langle A, B : \{\langle C, C' \rangle\}\rangle$. Consider the query defining an *idb* relation T, which contains the tuples of R, with the B-component replaced by its transitive closure. Let us assume that we have a ternary relation *ins*, where $ins(w, y, z)$ is interpreted as "z is obtained by inserting w into y." We show later how to define this relation in the language. The program consists of the following rules:

$$(r1) \qquad S(x, y) \leftarrow R(x, y)$$

$$(r2) \qquad S(x, z) \leftarrow S(x, y), u \in y, v \in y, u.C' = v.C, ins(\langle u.C, v.C' \rangle, y, z)$$

$$(r3) \qquad S'(x, z) \leftarrow S(x, z), S(x, z'), z \subseteq z', z \neq z'$$

$$(r4) \qquad T(x, z) \leftarrow S(x, z), \neg S'(x, z).$$

The first two rules compute in S pairs corresponding to pairs from R, such that the second component of a pair contains the corresponding component from the pair in R and possibly additional elements derived by transitivity. Obviously, for each pair $\langle x, y \rangle$ of R, there is a pair $\langle x, z \rangle$ in S, such that z is the transitive closure of y, but there are other tuples as well. To answer the query, we need to select for each x the unique tuple $\langle x, z \rangle$ of S, where z is maximal.[2] The third rule puts into S' tuples $\langle x, z \rangle$ such that z is not maximal for that x. The last rule then selects those that are maximal, using negation.

[2] We assume, for simplicity, that the first column of R is a key. It is easy to change the rules for the case when this does not hold.

We now show the program that defines *ins* for some given sort τ (the variables are of sort $\{\tau\}$ except for w, which is of sort τ):

$$\begin{aligned}
&\mathit{super}(w, y, z) && \leftarrow w \in z, y \subseteq z\\
&\mathit{not\text{-}min\text{-}super}(w, y, z) && \leftarrow \mathit{super}(w, y, z), \mathit{super}(w, y, z'), z' \subseteq z, z' \neq z\\
&\mathit{ins}(w, y, z) && \leftarrow \mathit{super}(w, y, z), \neg \mathit{not\text{-}min\text{-}super}(w, y, z)
\end{aligned}$$

Note that the program is sort specific only through its dependence on the sorts of the variables. The same program computes *ins* for another sort τ', if we assume that the sort of w is τ' and that of the other variables is $\{\tau'\}$. Note also that the preceding program is not safe. To make it safe, we would have to use derived relations to range restrict the various variables.

We note that although we used $\subseteq$ in the example as a built-in predicate, it can be expressed using membership and stratified negation.

The proof of the next result is omitted but can be reconstructed reasonably easily using the technique of Example 20.6.3.

THEOREM 20.6.4 A query is expressible in datalog^{cv} with stratified negation if and only if it is expressible in CALC^{cv}.

The preceding language relies heavily on negation to specify the new sets. We could consider more set-oriented constructs. An example is the *grouping* construct, which is closely related to the algebraic nest operation. For instance, in the language $\mathcal{LDL}$, the rule:

$$S(x, \langle y \rangle) \leftarrow R(x, y)$$

groups in S, for each x, all the y's related to it in R (i.e., S is the result of the nesting of R on the second coordinate).

The grouping construct can be used to simulate negation. Consider a query q whose input consists of two unary relations R, S not containing some particular element a and that computes $R - S$. Query q can be answered by the following $\mathcal{LDL}$ program:

$$\begin{aligned}
&\mathit{Temp}(x, a) \leftarrow R(x)\\
&\mathit{Temp}(x, x) \leftarrow S(x)\\
&T(x, \langle y \rangle) \leftarrow \mathit{Temp}(x, y)\\
&\mathit{Res}(x) \leftarrow T(x, \{a\})
\end{aligned}$$

Note that for an x in $R - S$, we derive $T(x, \{a\})$; but for x in $R \cap S$, we derive $T(x, \{x, a\}) \neq T(x, \{a\})$ because a is not in R.

From the previous example, it is clear that programs with grouping need not be monotone. This gives rise to semantic problems similar to those of negation. One possiblity, adopted in $\mathcal{LDL}$, is to define the semantics of programs with grouping analogously to stratification for negation.

20.7 Expressive Power and Complexity

This section presents two results. First the expressive power and complexity of ALG^{cv}/CALC^{cv} is established—it is the family of queries computable in hyperexponential time. Second, we consider the expressive power of ALG^{cv-}/CALC^{cv-} (i.e., in algebraic terms the expressive power of permitting the *nest* operator, but not *powerset*). Surprisingly, we show that the *nest* operator can be eliminated from ALG^{cv-} queries with flat input/ouput.

Complex Value Languages and Elementary Queries

We now characterize the queries in ALG^{cv} in terms of the set of computable queries in a certain complexity class. First the notion of computable query is extended to the complex value model in the straightforward manner. The complexity class of interest is the class of *elementary queries*, defined next.

The *hyperexponential* functions hyp_i for i in N are defined by

1. $hyp_0(m) = m$; and
2. $hyp_{i+1}(m) = 2^{hyp_i(m)}$ for $i \geq 0$.

A query is an *elementary query* if it is a computable query and has hyperexponential time data complexity[3] w.r.t. the database size. By database size we mean the amount of space it takes to write the content of the database using some natural encoding. Note that, for complex value databases, size can be very different from cardinality. For example, the database could consist of a single but very large complex value.

It turns out that a query is in ALG^{cv}/CALC^{cv} iff it is an elementary query.

THEOREM 20.7.1 A query is in ALG^{cv}/CALC^{cv} iff it is an elementary query.

Crux It is trivial to see that each query in ALG^{cv}/CALC^{cv} is elementary. All operations can be evaluated in polynomial time in the size of their arguments except for powerset, which takes exponential time.

Conversely, let q be of complexity hyp_n. We show how to compute it in CALC^{cv}.

Suppose first that an enumeration of $adom(\mathbf{I})$ is provided in some binary relation *succ*. (We explain later how this is done.) We prove that q can then be computed in CALC$^{cv}+\mu^+$. Let $X^0 = adom(I)$ and for each i, $X^i = powerset(X^{i-1})$. Observe that for each X^i, we can provide an enumeration as follows: First *succ* provides the enumeration for X^0; and for each i, we define $V <_i U$ for U, V in X^i if there exists x in $U - V$ such that each element larger than x (under $<_{i-1}$) is in both or neither of U, V. Clearly, there exists a query in CALC$^{cv}+\mu^+$ that constructs X^n and a binary relation representing $<_n$.

Now we view each element of X^n as an atomic element. The input instance together with X^n and the enumeration can be seen as an ordered database with size the order of hyp_n. Query q is now polynomial in this new (much larger) instance. Finally we can easily

[3] We are concerned exclusively with the *data* complexity. Observe that when considering the union of hyperexponential complexities, time and space coincide.

extend to complex values the result from the flat case that CALC+μ^+ can express QPTIME on ordered databases (Theorem 17.4.2). Thus CALC^{cv}+μ^+ can also express all QPTIME queries on ordered complex value databases, so q can be computed in CALC^{cv}+μ^+ using $<_n$ on X_n. By Theorem 20.6.1, CALC^{cv}+μ^+ is equivalent to CALC^{cv}, so there exists a CALC^{cv} query φ computing q if an (arbitrary) enumeration of the active domain is given in some binary relation *succ*.

To conclude the proof, it remains to remove the restriction on the existence of an enumeration of the active domain. Let φ' be the formula obtained from φ by replacing

1. *succ* by some fresh variable y (the sort of y is set of pairs); and
2. each literal $succ(t, t')$ by $\langle t, t' \rangle \in y$.

Then q can be computed by

$$\exists y(\varphi' \wedge \psi).$$

where ψ is the CALC^{cv} formula stating that y is the representation in a binary relation of an enumeration of the active domain. (Observe that it is easy to state in CALC^{cv} that the content of a binary relation is an enumeration.) ■

On the Power of the *nest* Operator

The *set-height* of a complex sort is the maximum number of set constructors in any branch of the sort. We can exhibit hierarchies of classes of queries in CALC^{cv} based on the set-height of the sorts of variables used in the query. For example, consider all queries that take as input a flat relational schema and produce as output a flat relation. Then for each $n > 0$, the family of CALC^{cv} queries using variables that have sorts with set-height $\leq n$ is strictly weaker than the family of CALC^{cv} queries using variables that have sorts with set-height $\leq n + 1$. A similar hierarchy exists for ALG^{cv}, based on the sorts of intermediate types used. Intuitively, these results follow from the use of the powerset operator, which essentially provides an additional exponential amount of scratch paper for each additional level of set nesting.

The bottom of this hierarchy is simply relational calculus. Recall that ALG^{cv-} can use the *nest* operator but not the powerset operator. It is thus natural to ask, Where do ALG^{cv-}/CALC^{cv-} (assuming flat input and output) lie relative to the relational calculus and the first level of the hierarchy? Rather surprisingly, it turns out that the *nest* operator alone does not increase expressive power. Specifically, we show now that with flat input and output, ALG^{cv-}/CALC^{cv-} is equivalent to relational calculus.

THEOREM 20.7.2 Let φ be a CALC^{cv-}/ALG^{cv-} query over a relational database schema $\mathbf{R}$ with output of relational sort S. Then there exists a relational calculus query φ' equivalent to φ.

Crux The basic intuition underlying the proof is that with a flat input in CALC^{cv-} or ALG^{cv-}, each set constructed at an intermediate stage can be identified by a tuple of atomic

values. In terms of ALG^{cv-}, the intuitive reason for this is that sets can be created only in two ways:

- by *nest*, which builds a relation whose nonnested coordinates form a key for the nested one, and
- by *set_create*, which can build only singleton sets.

Thus all created sets can be identified using some flat key of bounded length. The sets can then be simulated in the computation by their flat representations. The proof consists of

- providing a careful construction of the flat representation of the sets created in the computation, which reflects the history of their creation; and
- constructing a new query, equivalent to the original one, that uses only the flat representations of sets.

The details of the proof are omitted. ■

Observe that an immediate consequence of the previous result is that transitive closure or powerset are *not* expressible in ALG^{cv-}.

REMARK 20.7.3 The previous results focus on relational queries. The same technique can be used for nonflat inputs. An arbitrary input $\mathbf{I}$ can be represented by a flat database $\mathbf{I}_f$ of size polynomial in the size of the input. Now an arbitrary ALG^{cv-} query on $\mathbf{I}$ can be simulated by a relational query on $\mathbf{I}_f$ to yield a flat database representing the result. Finally the complex object result is constructed in polynomial time. This shows in particular that ALG^{cv-} is in PTIME. ■

20.8 A Practical Query Language for Complex Values

We conclude our discussion of languages for complex values with a brief survey of a fragment of the query language O_2SQL supported by the commercial object-oriented database system O_2 (see Chapter 21). This fragment provides an elegant syntax for accessing and constructing deeply nested complex values, and it has been incorporated into a recent industrial standard for object-oriented databases.

For the first example we recall the query

(4.3) What are the address and phone number of the Le Champo?

Using the **CINEMA** database (Fig. 3.1), this query can be expressed in O_2SQL as

```
element select tuple ( t.address, t.phone )
        from  t in Location
        where t.name = "Le Champo"
```

The *select-from-where* clause has semantics analogous to those for SQL. Unlike SQL, the *select* part can specify an essentially arbitrary complex value, not just tuples. A *select-from-where* clause returns a set[4]; the keyword **element** here is a desetting operator that returns a runtime error if the set does not have exactly one element.

The next example illustrates how O_2SQL can work inside nested structures. Recall the complex value shown in Fig. 20.2, which represents a portion of the **CINEMA** database. Let the full complex value be named *Films*. The following query returns all movies for which the director does not participate as an actor.

```
select  m.Title
from    f in Films
        m in f.Movies
where f.Director not in select a
                          from  a in m.Actors
```

O_2SQL also provides a mechanism for collapsing nested sets. Again using the complex value *Films* of Fig. 20.2, the following gives the set of all directors that have not acted in any Hitchcock film.

```
select  f.Director
from    f in Films
where f.Director not in flatten select m.Actors
                                 from  g in Films
                                       m in g.Movies
                                 where g.Director = "Hitchcock"
```

Here the inner *select-from-where* clause returns a set of sets of actors. The keyword **flatten** has the effect of forming the union of these sets to yield a set of actors.

We conclude with an illustration of how O_2SQL can be used to construct a deeply nested complex value. The following query builds, from the complex value *Films* of Fig. 20.2, a complex value of the same type that holds information about all movies for which the director does not serve as an actor.

```
select tuple ( Director: f.Director,
               Movies: select tuple ( Title: m.Title,
                                      Actors: select a
                                              from a in m.Actors )
                       from m in f.Movies
                       where f.Director not in m.Actors )
from f in Films
```

[4] In the full language O_2SQL, a list or bag might also be returned; we do not discuss that here. Furthermore, we do not include the keyword **unique** in our queries, although technically it should be included to remove duplicates from answer sets.

Bibliographic Notes

The original proposal for generalizing the relational model to allow entries in relations to be sets is often attributed to Makinouchi [Mak77]. Our presentation is strongly influenced by [AB88]. An extensive coverage of the field can be found in [Hul87]. The nested relation model is studied in [JS82, TF86, RKS88]. The V-relation model is studied in [BRS82, AB86, Ver89], and the essentially equivalent partition normal form (PNF) nested relation model is studied in [RKS88]. The connection of the PNF nested relations with dependencies has also been studied (e.g., in [TF86, OY87]). References [DM86a, DM92] develop a *while*-like language that expresses all computable queries (in the sense of [CH80b]) over "directories"; these are database structures that are essentially equivalent to nested relations.

There have been many proposals of algebras. In general, the earlier ones have essentially the power of ALG^{cv-} (due to obvious complexity considerations). The powerset operation was first proposed for the Logical Data Model of [KV84, KV93b].

The calculus presented in this chapter is based on Jacobs's calculus [Jac82]. This original proposal allowed noncomputable queries [Var83]. We use in this chapter a computable version of that calculus that is also used (with minor variations) in [KV84, KV93b, AB88, RKS88, Hul87].

Parameterized queries are close to the commonly used mathematical concept of *set comprehension*.

The equivalence of the algebra and the calculus has been shown in [AB88]. An equivalence result for a more general model had been previously given in [KV84, KV93b]. The equivalence result is preserved with oracles. In particular, it is shown in [AB88] that if the algebra and the calculus are extended with an identical set of oracles (i.e., sorted functions that are evaluated externally), the equivalence result still holds.

The strongly safe-range calculus, and the equivalence of ALG^{cv-} and CALC^{cv-}, are based on [AB88].

The fact that transitive closure can be computed in the calculus was noted in [AB88]. The result that any algebra query computing transitive closure requires exponential space (with the straightforward evaluation model) was shown in [SP94]. The equivalence between the calculus and various rule-based languages is from [AB88]. In the rule-based paradigm, nesting can be expressed in many ways. A main difference between various proposals of logic programming with a set construct is in their approach to nesting: grouping in $\mathcal{LDL}$ [BNR$^+$87], data functions in COL [AG91], and a form of universal quantification in [Kup87]. In [Kup88], equivalence of various rule-based languages is proved. In [GG88], it is shown that various programming primitives are interchangeable: powerset, fixpoint, various iterators.

The correspondence between ALG^{cv}/CALC^{cv} queries and elementary queries is studied in [HS93, KV93a]. Hierarchies of classes of queries based on the level of set nesting are considered in [HS93, KV93a]. Related work is presented in [Lie89a]. Exact complexity characterizations are obtained with fixpoint, which is no longer redundant when the level of set nesting is bounded [GV91].

Theorem 20.7.2 is from [PG88], which uses a proof based on a strongly safe calculus.

The proof of Theorem 20.7.2 outlined in this chapter suggests a strong connection between ALG^{cv-} and the V-relation model.

Reference [BTBW92] introduces a rich family of languages for complex objects, extended to include lists and bags, that is based on structural recursion. One language in this family corresponds to the nested algebra presented in this chapter. Using this, an elegant family of generalizations of Theorem 20.7.2 is developed in [Won93].

An extension of complex values, called *formats* [HY84], includes a marked union construct in addition to tuple and finitary set. Abstract notions of relative information capacity are developed there; for example, it can be shown that two complex value types have equivalent information capacity iff they are isomorphic.

Exercises

Exercise 20.1 (V-relations) Consider the schema R of sort

$$\langle A, B : \{\langle C, D\rangle\}\rangle.$$

Furthermore, we impose the fd $A \to B$ (more precisely, the generalization of a functional dependency). (a) Prove that for each instance I of R, the size of I is bounded by a polynomial in $adom(I)$. (b) Show how the same information can be naturally represented using two flat relations. (One suffices with some coding.) (c) Formalize the notion of V-relation of Section 20.1 and generalize the results of (a) and (b).

Exercise 20.2 Consider a (flat) relation R of sort

$$name\ age\ address\ car\ child\_name\ child\_age$$

and the multivalued dependency $name\ age\ address \twoheadrightarrow car$. Prove that the same information can be stored in a complex value relation of sort

$$\langle name, age, address, cars : \{\mathbf{dom}\}, children : \{\langle child\_name, child\_age\rangle\}\rangle$$

Discuss the advantages of this alternative representation. (In particular, show that for the same data, the size of the instance in the second representation is smaller. Also consider update anomalies.)

Exercise 20.3 Consider the value

$$\{\ \langle A : a, B : \langle A : \{a, b\}, B : \langle A : a\rangle\rangle, C : \langle\ \rangle\rangle,$$
$$\langle A : a, B : \langle A : \{\}, B : \langle A : a\rangle\rangle, C : \langle\ \rangle\rangle\ \}.$$

Show how to construct it in the core algebra from $\{a\}$ and $\{b\}$.

Exercise 20.4 Prove that for each complex value relation I, there exists a constant query in the core algebra returning I.

Exercise 20.5 Let **R** be a database schema consisting of a relation R of sort

$$\langle A : \mathbf{dom},\ B : \langle A : \{\mathbf{dom}\},\ B : \langle A : \mathbf{dom}\rangle\rangle,\ C : \langle\ \rangle\rangle;$$

and let $\tau = \{\langle A : \mathbf{dom},\ B : \{\{\mathbf{dom}\}\}\rangle\}$.

(a) Give a query computing for each $\mathbf{I}$ over $\mathbf{R}$, $adom(\mathbf{I})$.

(b) Give a query computing the set of values J of sort τ such that $adom(J) \subseteq adom(\mathbf{I})$.

Exercise 20.6 Prove that *set_create* can be expressed using the other operations of the core algebra. *Hint:* Use *powerset*.

Exercise 20.7 Formally define the following operations: (a) renaming, (b) singleton, (c) cross-product, and (d) join. In each case, prove that the operation is expressible in ALG^{cv}. Which of these can be expressed without powerset?

Exercise 20.8 (Nest,unnest)

(a) Show that *nest* is expressible in ALG^{cv}.

(b) Show that *unnest* is expressible in ALG^{cv} without using the powerset operator.

(c) Prove that $unnest_A$ is a right inverse of $nest_{A=(A_1 \ldots A_k)}$ and that $unnest_A$ has no right inverse.

Exercise 20.9 (Map) The operation $map_{C,q}$ is applicable to relations of sort τ where τ is of the form $\{\langle C : \{\tau'\}, \ldots\rangle\}$ and q is a query over relations of sort τ'. For instance, let

$$I = \{\langle C : I_1, C' : J_1\rangle, \langle C : I_2, C' : J_2\rangle, \langle C : I_3, C' : J_3\rangle\}.$$

Then

$$map_{C,q}(I) = \{\langle C : q(I_1), C' : J_1\rangle, \langle C : q(I_2), C' : J_2\rangle, \langle C : q(I_3), C' : J_3\rangle\}.$$

(a) Give an example of *map* and show how the query of this example can be expressed in ALG^{cv}.

(b) Give a formal definition of *map* and prove that the addition of *map* does not change the expressive power of the algebra.

Exercise 20.10 Show how to express

$$\{x \mid \{y \mid liked(x, y)\} = \{y \mid saw(x, y)\}\}$$

in the core calculus.

Exercise 20.11 The calculus is extended by allowing terms of the form $z \cup z'$ and $z - z'$ for each set term z, z' of identical sort. Prove that this does not modify the expressive power of the language. More generally, consider introducing in the calculus terms of the form $q(t_1, \ldots, t_n)$, where q is an n-ary algebraic operation and the t_i are set terms of appropriate sort.

Exercise 20.12 Give five queries on the **CINEMA** database expressed in ALG^{cv}. Give the same queries in CALC^{cv}.

Exercise 20.13 Complete the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$ for Theorem 20.5.1. Complete the proof of "Last Stage" for Theorem 20.5.1.

Exercise 20.14 This exercise elaborates the simulation of CALC^{cv} by ALG^{cv} presented in the proof of Theorem 20.5.1. In particular, give the details of

(a) the construction of E_{adom}

(b) the construction of G_τ for each τ

(c) the last stage of the construction.

Exercise 20.15 Show that the query in Example 20.6.2 is strongly safe range (e.g., give a query in ALG^{cv-} or CALC^{cv-} equivalent to it).

Exercise 20.16 Show that every strongly safe-range query is in ALG^{cv-} [one direction of (b) of Theorem 20.5.3].

Exercise 20.17 Sketch a program expressing the query *even* in $\text{CALC}^{cv}+\mu^+$.

Exercise 20.18 Prove that $\text{CALC}^{cv}+\mu^+ = \text{ALG}^{cv}$.

Exercise 20.19 Define a *while* language based on ALG^{cv}. Show that it does not have more power than ALG^{cv}.

Exercise 20.20 Consider a query q whose input consists of two relations *blue*, *red* of sort $\langle A, B\rangle$ (i.e., consists of two graphs). Query q returns a relation of sort $\langle A, B : \{\textbf{dom}\}\rangle$ with the following meaning. A tuple $\langle x, X\rangle$ is in the result if x is a vertex and X is the set of vertexes y such that there exists a path from x to y alternating blue and red edges. Prove in one line that q is expressible in ALG^{cv}. Show how to express q in some complex value language of this chapter.

Exercise 20.21 Generalize the construction of Example 20.6.2 to prove Theorem 20.6.1.

Exercise 20.22 Datalog with stratified negation was shown to be weaker than datalog with inflationary negation. Is the situation similar for datalog^{cv} with negation?

Exercise 20.23 Exhibit a query that is not expressible in CALC^{cv-} but is expressible in CALC^{cv}, and one that is not expressible in CALC^{cv}.

Exercise 20.24 Give a relational calculus formula or algebra expression for the query in Example 20.4.8.

★ **Exercise 20.25** Recall the language $while_N$ from Chapter 18. The language allows assignments of relational algebra expressions to relational variables, looping, and integer arithmetic. Let $while_N^{cv}$ be like $while_N$, except that the relational algebra expressions are in ALG^{cv}. Prove that $while_N^{cv}$ can express all queries from flat relations to flat relations.