Appendix A

A New Relational Algebra

INTRODUCTION

In this appendix, we describe a new relational algebra that we call **A**. The name **A** is a doubly recursive acronym: It stands for *ALGEBRA,* which in turn stands for *A Logical Genesis Explains Basic Relational Algebra*. As this expanded name suggests, **A** has been designed in such a way as to emphasize, perhaps more clearly than previous algebras have done, its close relationship to and solid foundation in the discipline of predicate logic. In addition, the abbreviated name **A** has pleasing connotations of beginning, basis, foundation, simplicity, and the like——not to mention that it is an obvious precursor to **D**.

The algebra **A** differs from Codd's original algebra [20-22] in four principal respects:

- Cartesian product (TIMES) is replaced by a natural join operator that, appealing to its counterpart in predicate logic, we call simply ◂AND▸. The original TIMES becomes merely a special case of ◂AND▸. *Note:* We adopt the convention of using solid arrowheads ◂ and ▸ to delimit **A** operator names, as in ◂AND▸, in order to distinguish those operators from operators with the same name in predicate logic or **Tutorial D** or both. Also, in the case of ◂AND▸ in particular, do not be misled by the name: The ◂AND▸ operator of **A** is, of course, a relational operator (it operates on relations and returns a relation), whereas its predicate logic and **Tutorial D** counterparts are logical operators (they operate on propositions or, more generally, predicates and return a truth value). Analogous remarks apply to the **A** operators ◂OR▸ and ◂NOT▸ as well (see the next two bullet items).

- UNION is replaced by a more general ◂OR▸ operator that does not require its operands to be of the same type. The original UNION becomes merely a special case of ◂OR▸.

- We include a relational complement operator, ◄NOT►. The availability of ◄NOT► allows us to drop the relational difference operator (MINUS), since that operator can easily be expressed in terms of ◄AND► and ◄NOT►.

- We are able to dispense with *restrict* (WHERE), EXTEND, and SUMMARIZE, since these operators all become further special cases of ◄AND►. *Note:* EXTEND and SUMMARIZE were not included in Codd's original algebra but were added subsequently [132].

In addition to ◄AND►, ◄OR►, and ◄NOT►, **A** includes three operators called ◄RENAME►, ◄REMOVE►, and ◄COMPOSE►, which are discussed in the next section but one. It also includes a transitive closure operator, ◄TCLOSE►; however, this operator is essentially identical to TCLOSE as discussed in Chapters 2 and 6, and we have little more to say about it in this appendix.

MOTIVATION AND JUSTIFICATION

In this section we explain our reasons for developing **A** and justify the departures from Codd's algebra identified in the previous section. Our explanations and notation are deliberately not too formal (formal definitions appear in a later section). In particular, we show tuples not as sets of <*A,T,V*> triples, as the formal apparatus of Chapter 4 would require, but as simple commalists of values enclosed in angle brackets. For example, we use the expression <EX,DX> to mean a 2-tuple in which EX denotes a certain employee and DX a certain department.

Since we often appeal in what follows to ideas from predicate logic, using natural language predicates as examples, a brief note on the terminology we use in that connection is in order:

- First, where our examples include operands that are relvar references, the predicates for the corresponding relvars are relvar predicates as described in Chapter 2. For example, the predicate "Employee E works in department D" might be the relvar predicate for a relvar called WORKS_IN.

- We refer to the parameters of a predicate as its **free variables.**[1] For example, in the predicate "Employee E works in department D," E and D are free variables.

- Second, we use Greek derivatives involving the suffix *-adic* when referring to the number of free variables in a predicate, but Latin ones involving the suffix *-ary* when referring to the degree of a relation. For example, the predicate "Employee E works in department D" is dyadic, while a relation corresponding to that predicate is binary.

[1]We apologize for this slight terminological inconsistency, but in fact——as we have discussed in detail elsewhere [78]——there seems to be almost no consensus on the use of terms in logic textbooks either.

The algebra **A** has been motivated by certain general objectives, the following among them:

- For psychological reasons, we sought a collection of operators with immediate counterparts in logic and with less reliance on set theory in their nomenclature. We feel that relational theory is better taught and understood this way; indeed, we have been dismayed at the widespread lack of appreciation in the database community at large of the logical foundations of relational theory, and we think it likely that this lack has contributed to the deficiencies we observe in available relational (or would-be relational) technology.

- Previous algebras have had more than one operator corresponding to logical AND. We thought this apparent redundancy worth looking into, with a view to eliminating it.

- We wanted all of the relational operators of **Tutorial D** to be mappable to expressions in **A**, for convenience and also for our own satisfaction (and we would strongly recommend that the same be true for any industrial strength **D** as well). Full details of the mappings in question are deferred to the final section of this appendix, but some idea of what is involved can be found in examples prior to that section.

We now proceed to justify the four principal respects in which **A** differs from previous algebras.

Dispensing with TIMES

In logic, when two predicates are connected by AND, attention must be paid to the names of the free variables. Any free variable name that appears in both predicates must be understood to stand for the same thing when it consequently appears more than once in the resulting predicate. For example, consider the natural language predicates "Employee E works in department D" and "Employee E works on project J." The AND of these two predicates yields a triadic predicate, not a tetradic one: namely, "Employee E works in department D and employee E works on project J." This latter predicate can perhaps be abbreviated to just "Employee E works in department D and on project J," to stress the fact that we cannot substitute some particular employee for the E that works in department D without at the same time substituting that very same employee for the E that works on project J. This observation regarding free variable names lies at the heart of the well-known natural join operator (the ◂AND▸ operator in **A**).

As for the classical TIMES operator, it is of course just a special case of natural join (which hereinafter we abbreviate to simply *join*). More precisely, TIMES corresponds to the AND of two predicates that have no free variables in common——for example, "Employee E works in department D and project J has budget B." TIMES as such can thus be discarded.

We return for a moment to the predicate "Employee E works in department D and on project J" to make another point. As already noted, that formulation of the predicate is really an abbreviation. Now, it might be abbreviated still further, to just "Employee E works in department D on project J." However, that further abbreviation could lead to the erroneous conclusion that project J is somehow "in" department D. In reference [21], Codd characterized this kind of error as *the connection trap,* but it has since become known, at least in some circles, as the *join* trap instead——rather unfairly, we feel, since it is not unique to join in particular, nor to relational operators in general. In fact, it was precisely Codd's point in reference [21] that the error is more likely to arise in a nonrelational context than it is in a relational one.

Dispensing with UNION

We can combine natural language predicates with OR as well as AND. Thus, there is a ternary relation corresponding to the triadic predicate "Employee E works in department D or employee E works on project J." If employee EX works in department DX, then <EX,DX,*j*> is a tuple in the body of this relation for all possible projects *j,* regardless of whether employee EX actually works on project *j* (and regardless of whether there is even a project *j* in the company at this time). Likewise, if employee EX works on project JX, then <EX,*d,*JX> is a tuple in the body of this relation for all possible departments *d,* regardless of whether employee EX actually works in department *d* (and regardless of whether there is even a department *d* in the company at this time).

Just as we introduce ◂AND▸ as the **A** counterpart of AND, therefore, we introduce ◂OR▸ as the **A** counterpart of OR. As for the classical UNION operator, it is of course just a special case of ◂OR▸. More precisely, UNION corresponds to the OR of two predicates that have exactly the same free variables——for example, "Employee E works in department D or employee E is on loan to department D." UNION as such can thus be discarded.

Note: We do not concern ourselves here with the computational difficulties that might arise from our generalization of Codd's UNION, because at this point we are only defining an algebra. Various safety mechanisms can be (and normally are) imposed in practice to circumvent such difficulties. For similar reasons, we also do not concern ourselves with the high degree of redundancy that most relations produced by ◂OR▸ will exhibit.

Dispensing with MINUS

Let WORKS_IN be a relation with attributes E and D, where E is an employee and D is a department, and let the corresponding predicate be "Employee E works in department D." Then the *logical complement* (◂NOT▸) of this relation has a body that consists of all possible tuples of the form <E,D> such that it is not the case that employee E works in department D. *Note:* Computational difficulties arise here as they did with ◂OR▸, but again we need not concern ourselves with them at this juncture.

To see that MINUS can now be discarded, consider the following example. Let WORKS_IN be as above; let WORKS_ON be a relation with attributes E and J, where J is a project; and let the predicate corresponding to WORKS_ON be "Employee E works on project J." Now consider the unary relation corresponding to the monadic predicate "Employee E works in some department but works on no project at all." In Codd's algebra, we could obtain this relation by projecting both WORKS_IN and WORKS_ON over their E attributes and then taking the appropriate difference. In **A**, we first project WORKS_ON over E (see the next section for a discussion of projection), and then we take the ◂NOT▸ of that projection; the corresponding predicate is "There does not exist a project such that employee E works on that project." This relation can then be joined ("◂AND▸ed") with WORKS_IN, and the result projected over E, to obtain the desired final result.

Dispensing with *restrict* (WHERE), EXTEND, and SUMMARIZE

Restrict (WHERE), EXTEND, and SUMMARIZE all require certain operators to be invoked as part of their execution. In the case of *restrict,* the operators in question return values (truth values, to be precise) that are used to disqualify certain tuples from appearing in the result relation; in the case of EXTEND and SUMMARIZE, they return values that are used as the basis for defining certain attributes in the result relation.

It occurred to us that it made sense, and could possibly be useful, **to treat such operators as relations**. Consider an operator *Op* that is in fact a scalar function (a scalar function is an operator for which every valid invocation returns exactly one result and that result is a scalar value). Suppose *Op* has *n* parameters. Then *Op* can be treated as a relation with *n*+1 attributes, one for each parameter and one for the result. The attributes corresponding to the parameters clearly form a key for this relation; however, that key is not necessarily the only one. For example, let PLUS be a relation with attributes X, Y, and Z, each of type INTEGER, corresponding to the scalar function "+" of integer arithmetic and to the predicate "X + Y = Z." Then each of {X,Y}, {Y,Z}, and {Z,X} is a key for relation PLUS; further, that relation contains exactly one 3-tuple <*x,y,z*> for every possible combination of values *x, y,* and *z* that satisfies the predicate (i.e., such that *x* + *y* = *z*).

Note, incidentally, that the relation PLUS can be regarded as an example of what in Chapters 5 and 6 we referred to as a "relcon" or relation constant: It is named, like a relvar, but unlike a relvar it has a value that does not change over time. Of course, analgous remarks apply to the relational representation of any function; the keys discussed in the previous paragraph are thus keys for a "relcon," not a relvar.

Let us take a closer look at what is going on here. A scalar function is a special case of a relation, of course, as the PLUS example illustrates. In fact, any relation can always be regarded as an operator that maps from some subset of its attributes to the rest; and, if the mapping in question is a functional (i.e., many-to-one) mapping specifically, then the relation can be regarded as a function. In fact, since a set of *n* elements has 2^n subsets, a relation of degree *n* can be regarded as representing 2^n different operators, some of which will be functions and some not (in general). For example, PLUS can be regarded, among other things, as an operator that maps from Z to X and Y——but of course that particular mapping is not a functional one (the functional dependencies $Z \rightarrow X$ and $Z \rightarrow Y$ do not hold), and the corresponding operator is thus not a function.

We now claim that, given the fact that operators can be treated as relations, and given also the availability of the **A** operators ◄AND►, ◄REMOVE►, and ◄RENAME► (the latter two still to be discussed), it is indeed the case that we can dispense with *restrict,* EXTEND, and SUMMARIZE. We will justify this claim in the next section but one.

◄REMOVE►, ◄RENAME►, AND ◄COMPOSE►

◄REMOVE►

◄REMOVE► is the **A** counterpart to the existential quantifier of predicate logic. It corresponds to Codd's *project*. However, it differs from *project* in that it specifies, not an attribute (or attributes, plural) to be projected over, but rather an attribute to be "projected away"; it is equivalent to projecting the relation in question over all of its attributes except the one specified. Our motivation for this inversion, so to speak, with respect to Codd's *project* is a psychological one——projecting a relation with (say) attributes X and Y over attribute X is equivalent to existentially quantifying over attribute Y. For example, the projection of WORKS_IN over E corresponds to the natural language predicate "There exists some department D such that employee E works in department D." We thus feel that ◄REMOVE► is psychologically closer to our foundation in logic than *project* is. *The Third Manifesto,* however, explicitly requires the language **D** not to arbitrate in this matter; rather, projection over specified attributes and projection over all except specified attributes are required to be equally easy to express.

◂RENAME▸

The purpose of ◂RENAME▸ is, loosely, to rename some attribute of some relation. More precisely, the ◂RENAME▸ operator takes a given relation and returns another that is identical to the given one except that one of its attributes has a different name. Such an operator is required[2] in any concrete syntax for relational expressions in which attributes are distinguished by name, as they are in **A** (and **D**).

◂COMPOSE▸

In addition to the operators discussed so far——◂AND▸, ◂OR▸, ◂NOT▸, ◂REMOVE▸, and ◂RENAME▸——we have allowed ourselves the luxury (some might think) of including a "macro" operator called ◂COMPOSE▸. ◂COMPOSE▸ is a combination of ◂AND▸ and ◂REMOVE▸, in which attributes common to the "◂AND▸ed" relations are subsequently "◂REMOVE▸d." The name ◂COMPOSE▸ is meant to be suggestive of the fact that relational composition is a natural generalization of functional composition. (In case you are not familiar with this latter notion, the composition of two functions *f*(...) and *g*(...), in that order, is the function *f*(*g*(...)).) *Note:* Codd did in fact include a relational composition operator in his earliest papers [20-21] but for some reason subsequently discarded it; we find it useful in support of our desire to treat operators as relations. To be specific, it turns out that a certain degenerate form of composition can be used to simulate the expression of operator invocations, as will be seen in the next section.

Closing Remarks

It should be obvious that **A** is relationally complete [22]. Previous algebras have needed six operators for this purpose (typically RENAME, *restrict, project,* TIMES, UNION, and MINUS); we have reduced that number to five. Moreover, thanks to our observation that operators can be treated as relations, we have also avoided the need for EXTEND and SUMMARIZE; indeed, these operators might have been added needlessly in the past, simply for lack of that observation. Points arising:

- As a matter of fact **A** is "more than" relationally complete, in the sense that its unconstrained ◂OR▸ and ◂NOT▸ operators permit the definition of relations that cannot be defined in previous algebras. The point is purely academic, of course, since as already noted the ◂OR▸ and ◂NOT▸ operations will not be totally unconstrained in practice, in order to avoid certain computational problems that would otherwise arise.

[2] Or highly desirable, at any rate. As the next section shows, ◂RENAME▸ is in fact not a primitive operation.

- We do not actually need both ◂AND▸ and ◂OR▸ in order to achieve relational completeness, thanks to De Morgan's Laws. For example, *A* ◂AND▸ *B* is identically equal to ◂NOT▸ ((◂NOT▸ *A*) ◂OR▸ (◂NOT▸ *B*)), so we could dispense with ◂AND▸ if we included both ◂NOT▸ and ◂OR▸. We could even collapse ◂NOT▸ and ◂OR▸ into a single operator, ◂NOR▸ ("neither *A* nor *B*"; equivalently, "not *A* and not *B*"). Equally well, of course, we could dispense with ◂OR▸ and collapse ◂AND▸ and ◂NOT▸ into a single operator, ◂NAND▸ ("not *A* or not *B*"). Overall, therefore, we could if desired reduce our algebra to just three operators: ◂RENAME▸, ◂REMOVE▸, and either ◂NOR▸ or ◂NAND▸ (plus ◂TCLOSE▸).

- In fact, we will show in the next section that we do not really need ◂RENAME▸ either; thus, we could in fact reduce our algebra still further to just the two operators ◂REMOVE▸ and either ◂NOR▸ or ◂NAND▸ (plus ◂TCLOSE▸).

Of course, we are not suggesting that all of the various operators that we claim can be dispensed with should in fact *be* dispensed with in the concrete syntax of **D**——they are useful and convenient shorthands, generally speaking, and as a matter of fact RM Prescription 18 of our *Manifesto* expressly requires that they ("or some logical equivalent thereof") all be supported. But we do suggest that such operators be explicitly defined *as* shorthands, for reasons of clarity and simplicity among others [31].

TREATING OPERATORS AS RELATIONS

In this section we elaborate on our idea of treating operators as relations. Consider the relation PLUS once again, with attributes X, Y, and Z, each of type INTEGER, corresponding to the predicate "X + Y = Z." Let TWO_AND_TWO be that relation whose body consists of just the single 2-tuple

```
{ < X, INTEGER, 2 >, < Y, INTEGER, 2 > }
```

(we now revert to something closer to the formal notation for tuples——i.e., as sets of <*A,T,v*> triples——introduced in Chapter 2). Then the expression

```
TWO_AND_TWO ◂COMPOSE▸ PLUS
```

yields a relation whose body consists of the single 1-tuple

```
{ < Z, INTEGER, 4 > }
```

Observe, therefore, that we have effectively invoked the "+" operator with arguments X = 2 and Y = 2 and obtained the result Z = 4.[3] Of course, that result is still embedded as an attribute value inside a tuple inside a relation (like all **A** operators, ◂COMPOSE▸

[3] Note that the result has a name, Z. We are still considering the implications of this fact for the language **D**.

returns a relation); if we want to extract that result as a pure scalar value, we will have to go beyond **A** *per se* and make use of the operators (required by RM Prescriptions 7 and 6, respectively) for (a) extracting a specified tuple from a specified relation (necessarily of cardinality one) and then (b) extracting a specified attribute value from a specified tuple. In **Tutorial D** terms, for example, these extractions can be performed as follows:

```
    Z FROM ( TUPLE FROM ( result ) )
```

where *result* denotes the result of evaluating the **A** expression TWO_AND_TWO ◂COMPOSE▸ PLUS.

In other words, while it is certainly true that any given operator can be treated as a relation, it will still be necessary to step outside the confines of the algebra *per se* in order to obtain the actual result of some invocation of that operator. For present purposes, however, we are interested only in treating operators as relations *within a pure relational context;* such a treatment allows us to explain the classical relational operation EXTEND, for example, in a purely relational way (i.e., without having to leave the relational context at all), as we now proceed to demonstrate.

Consider the expression

```
    TWO_AND_TWO ◂AND▸ PLUS
```

(this expression is the same as before, except that we have replaced ◂COMPOSE▸ by ◂AND▸). The result is a relation whose body consists of just the single 3-tuple

```
    { < X, INTEGER, 2 >, < Y, INTEGER, 2 >, < Z, INTEGER, 4 > }
```

It should be clear, therefore, that the original **A** expression is logically equivalent to the following **Tutorial D** *extension:*

```
    EXTEND TWO_AND_TWO ADD ( X + Y AS Z )
```

This example should thus be sufficient to suggest how we can indeed dispense with EXTEND, as claimed.

Moreover, that very same expression TWO_AND_TWO ◂AND▸ PLUS is logically equivalent to the following **Tutorial D** *restriction:*

```
    PLUS WHERE X = 2 AND Y = 2
```

This same example should thus also be sufficient to suggest how we can dispense with *restrict,* again as claimed.

As an aside, we remark that if we were to rename attributes X and Z of PLUS as Z and X, respectively, then the expression TWO_AND_TWO ◂AND▸ PLUS would yield a relation whose body consists of just the single 3-tuple

```
    { < Z, INTEGER, 2 >, < Y, INTEGER, 2 >, < X, INTEGER, 0 > }
```

In other words, MINUS would be just a good a name for our "relcon" as PLUS is, psychologically speaking.

As for SUMMARIZE, it is well known that any given summarization can be expressed in terms of EXTEND instead of SUMMARIZE *per se* (though the details are a little complicated and we omit them here; see the final section of this appendix for further explanation). It follows that we can dispense with SUMMARIZE as well.

Now consider the following **Tutorial D** expression:

```
R RENAME ( X AS Y )
```

(we assume here that R denotes a relation with an attribute called X and no attribute called Y). Then the **Tutorial D** expression

```
( EXTEND R ADD ( X AS Y ) ) { ALL BUT X }
```

is semantically equivalent to the original RENAME expression. Thus, it should be clear that ◂RENAME▸ can be expressed in terms of EXTEND (which as we already know is basically just ◂AND▸) and ◂REMOVE▸, and hence is not primitive.

Before we leave this section, we would like to stress the point that it is not just operators that are scalar functions specifically that can be treated as relations. Consider the following examples:

- An example of an operator that is scalar but not a function is SQRT——"square root"——which, given a positive numeric argument, returns two scalar results (at least, we will assume so for the sake of this discussion). For example, SQRT(4.0) returns both +2.0 and -2.0.

- An example of an operator that is a function but not scalar is ADDR_OF ("address of"), which, given an employee E, returns the address of that employee as a collection——more precisely, a tuple——involving four scalar values (STREET, CITY, STATE, and ZIP).

Again we take a closer look. First, SQRT. SQRT can obviously be treated as a relation with attributes X and Y, say, each of type RATIONAL ($X \geq 0$). However, that relation is not a function because the functional dependency $X \rightarrow Y$ does not hold: for example, the tuples (4.0,+2.0) and (4.0,-2.0) both appear. (By contrast, the functional dependency $Y \rightarrow X$ does hold; SQRT can be regarded as a function if it is looked at in the inverse——i.e., "square of"——direction.) Observe that the relation contains:

- For $x = 0$, exactly one tuple with X = x

- For $x > 0$, exactly two tuples with X = x

- For $x < 0$, no tuples at all with X = x

It follows from the foregoing that the expression

```
    SQRT ◄COMPOSE► { { < X, RATIONAL, 4.0 > } }
```

effectively represents an invocation of the SQRT operator, but——in contrast to the situation in conventional programming languages——the invocation in question returns two results. More precisely, it produces a (unary) relation with the following body:

```
    { { < Y, RATIONAL, +2.0 > }, { < Y, RATIONAL, -2.0 > } }
```

(If desired, we could now go on to extract the individual scalar values +2.0 and -2.0 from this relation.) One implication of this example is that a relational language such as **D** might reasonably include an extended form of EXTEND that——unlike the traditional EXTEND——is not necessarily limited to producing exactly one output tuple from each input tuple.

By way of another example, consider the expression

```
    SQRT ◄COMPOSE► { { < X, RATIONAL, -4.0 > } }
```

This expression also represents an invocation of the SQRT operator, but——again in contrast to the situation in conventional programming languages——the invocation in question returns no result (more precisely, it produces a relation with heading {Y RATIONAL} and body empty). In conventional programming languages the invocation SQRT(-4.0) would give rise to a run-time exception.

Now we turn to ADDR_OF. This operator too can obviously be treated as a relation, this one having attributes E, STREET, CITY, STATE, and ZIP, where {E} is a key. *Note:* The other four attributes might form a key as well, if no two employees ever live at the same address (in which case the ADDR_OF relation would correspond to the inverse function that also happened to apply). Of course, the name ADDR_OF would be a little questionable if such were the case; EMP_AT might be just as appropriate, EMP_ADDR perhaps more so. The issue is merely psychological, of course.[4]

It follows from the foregoing that the expression

    { { < E, EMPLOYEE, *e* > } } ◄COMPOSE► ADDR_OF

(where *e* denotes some employee) effectively represents an invocation of the ADDR_OF operator, but——in contrast to the situation in conventional programming languages——the invocation in question returns *a nonscalar result*. One implication of this example is that a relational language such as **D** might reasonably include an extended form of EXTEND that (unlike the traditional EXTEND) is not necessarily limited to producing just one additional attribute.[5]

[4] Actually, analogous remarks apply to the SQRT example, where the name is again not very appropriate if the relation is looked at in the inverse ("square of") direction.

[5] **Tutorial D** does support such an operator. What is more, the additional attributes can be scalar-, tuple-, or relation-valued, or any combination.

FORMAL DEFINITIONS

We now proceed to give formal definitions for the **A** operators discussed up to this point.[6] First we explain our notation (which is based on that introduced in Chapter 2, of course, and——unlike that of previous sections——is now meant to be completely precise). Let *r* be a relation, let *A* be the name of an attribute of *r,* let *T* be the name of the corresponding type (i.e., the type of attribute *A*), and let *v* be a value of type *T*. Then:

- The heading H*r* of *r* is a set of attributes (i.e., ordered pairs of the form <*A,T*>). By definition, no two attributes in that set contain the same attribute name *A*.
- Let *tr* be a tuple that conforms to H*r;* i.e., *tr* is a set of ordered triples of the form <*A,T,v*>, one such triple for each attribute in H*r*.
- The body B*r* of *r* is a set of such tuples *tr*. Note that (in general) there will be some such tuples *tr* that conform to H*r* but do not appear in B*r*.

  The rest of our notation is meant to be self-explanatory.

Observe that a heading is a set, a body is a set, and a tuple is a set (and we remind you from Chapter 2 that every subset of a heading is a heading, every subset of a body is a body, and every subset of a tuple is a tuple). A member of a heading is an attribute (i.e., an ordered pair of the form <*A,T*>); a member of a body is a tuple; and a member of a tuple is an ordered triple of the form <*A,T,v*>.

Now we can define the operators *per se*. Each of the definitions that follow consists of (a) a formal specification of the rules, if any, that apply to the operands of the operator in question, (b) a formal specification of the heading of the result of that operator, and (c) a formal specification of the body of that result, followed by (d) an informal discussion of the formal specifications.

- Let *s* be ◄NOT► *r*.

  H*s* = H*r*

  B*s* = { *ts* : *exists tr* (*tr* ∉ B*r and ts* = *tr*) }

  The ◄NOT► operator yields the complement *s* of a given relation *r*. The heading of *s* is the heading of *r*. The body of *s* contains every tuple with that heading that is not in the body of *r*.

[6] Greaves [97] gives definitions of all of the operators of **A** in terms of the formal specification language Z.

- Let *s* be *r* ◂REMOVE▸ *A*. It is required that there exist some type *T* such that <*A*,*T*> ∈ H*r*.

  ```
  Hs = Hr minus { <A,T> }

  Bs = { ts : exists tr exists v
              ( tr ∈ Br and v ∈ T and <A,T,v> ∈ tr and
                                  ts = tr minus { <A,T,v> } ) }
  ```

  The ◂REMOVE▸ operator yields a relation *s* formed by removing a given attribute *A* from a given relation *r*. The operation is equivalent to taking the projection of *r* over all of its attributes except *A*. The heading of *s* is the heading of *r* minus the ordered pair <*A*,*T*>. The body of *s* contains every tuple that conforms to the heading of *s* and is a subset of some tuple of *r*.

- Let *s* be *r* ◂RENAME▸ (*A*,*B*). It is required that there exist some type *T* such that <*A*,*T*> ∈ H*r* and that there exist no type *T* such that <*B*,*T*> ∈ H*r*.

  ```
  Hs = ( Hr minus { <A,T> } ) union { <B,T> }

  Bs = { ts : exists tr exists v
              ( tr ∈ Br and v ∈ T and <A,T,v> ∈ tr and
                                  ts = ( tr minus { <A,T,v> } )
                                            union { <B,T,v> } ) }
  ```

  The ◂RENAME▸ operator yields a relation *s* that differs from a given relation *r* only in the name of one of its attributes, which is changed from *A* to *B*. The heading of *s* is the heading of *r* except that the ordered pair <*A*,*T*> is replaced by the ordered pair <*B*,*T*>. The body of *s* consists of every tuple of the body of *r*, except that in each such tuple the triple <*A*,*T*,*v*> is replaced by the triple <*B*,*T*,*v*>.

- Let *s* be *r1* ◂AND▸ *r2*. It is required that if <*A*,*T1*> ∈ H*r1* and <*A*,*T2*> ∈ H*r2*, then *T1* = *T2*.

  ```
  Hs = Hr1 union Hr2

  Bs = { ts : exists tr1 exists tr2
                 ( ( tr1 ∈ Br1 and tr2 ∈ Br2 ) and
                                ts = tr1 union tr2 ) }
  ```

  The ◂AND▸ operator is relational *conjunction*, yielding a relation *s* that in previous literature has been referred to as the (natural) join of the two given relations *r1* and *r2*. The heading of *s* is the union of the headings of *r1* and *r2*. The body of *s* contains every tuple that conforms to the heading of *s* and is a superset of both some tuple in the body of *r1* and some tuple in the body of *r2*. We remark that the ◂AND▸ operator might logically be called the *conjoin*.

- Let *s* be *r1* ◂OR▸ *r2*. It is required that if <*A,T1*> $\in$ H*r1* and <*A,T2*> $\in$ H*r2,* then *T1* = *T2*.

  H*s* = H*r1 union* H*r2*

  ```
  Bs = { ts : exists tr1 exists tr2
              ( ( tr1 ∈ Br1 or tr2 ∈ Br2 ) and
                                 ts = tr1 union tr2 ) }
  ```

  The ◂OR▸ operator is relational *disjunction,* being a generalization of what in previous literature has been referred to as union (in the special case where the given relations *r1* and *r2* have the same heading, the result *s* is in fact the union of those two relations in the traditional sense). The heading of *s* is the union of the headings of *r1* and *r2*. The body of *s* contains every tuple that conforms to the heading of *s* and is a superset of either some tuple in the body of *r1* or some tuple in the body of *r2*. We remark that the ◂OR▸ operator might logically be called the *disjoin*.

We also define the "macro" operator ◂COMPOSE▸. Let *s* be *r1* ◂COMPOSE▸ *r2* (where *r1* and *r2* are as for ◂AND▸). Let the attributes common to *r1* and *r2* be *A1, A2, ..., An* (*n* $\geq$ 0). Then *s* is the result of the expression

    (*r1* ◂AND▸ *r2*) ◂REMOVE▸ *An* ... ◂REMOVE▸ *A2* ◂REMOVE▸ *A1*

Note that when *n* = 0, *r1* ◂COMPOSE▸ *r2* is the same as *r1* ◂AND▸ *r2,* which is in turn the same as *r1* TIMES *r2* in Codd's algebra.

Finally, we remind you that the **A** operator ◂TCLOSE▸ is essentially identical to the TCLOSE operator already discussed in Chapters 2 and 6.

HOW Tutorial D BUILDS ON A

As noted in the body of the book, many——in fact, almost all——of the built-in relational operators in **Tutorial D** are really just shorthands. In this section we justify this remark by showing how the operators in question[7] map to those of the relational algebra **A** defined in earlier sections. The notation is intended to be self-explanatory, for the most part.

Transitive closure: The **Tutorial D** <*tclose*>

    TCLOSE *r*

is semantically equivalent to the **A** expression

    ◂TCLOSE▸ *r*

[7] Or most of them, at any rate. For simplicity, however, we ignore operators like <*n-adic join*> that are clearly equivalent to certain combinations of other operators.

Renaming: The **Tutorial D** <*rename*>

    r RENAME (*A* AS *B*)

is semantically equivalent to the **A** expression

    r ◄RENAME► (*A*,*B*)

Other **Tutorial D** <*rename*> formats are just shorthand for repeated application of the format shown above.

Projection: The **Tutorial D** <*project*>

    r { ALL BUT *A* }

is semantically equivalent to the **A** expression

    r ◄REMOVE► *A*

Other **Tutorial D** <*project*> formats are readily defined in terms of the format shown above.

Join: The **Tutorial D** <*dyadic join*>

    r1 JOIN *r2*

is semantically equivalent to the **A** expression

    r1 ◄AND► *r2*

The **Tutorial D** <*dyadic intersect*> *r1* INTERSECT *r2* is just that special case of *r1* JOIN *r2* in which *r1* and *r2* have the same heading, so the ◄AND► operator of **A** takes care of INTERSECT as well.

Compose: The **Tutorial D** <*compose*>

    r1 COMPOSE *r2*

is semantically equivalent to the **A** expression

    r1 ◄COMPOSE► *r2*

Union: The **Tutorial D** <*dyadic union*>

    r1 UNION *r2*

(where *r1* and *r2* have the same heading) is semantically equivalent to the **A** expression

    r1 ◄OR► *r2*

Minus: The **Tutorial D** <*minus*>

    r1 MINUS *r2*

(where *r1* and *r2* have the same heading) is semantically equivalent to the **A** expression

    r1 ◄AND► (◄NOT► *r2*)

Semijoin: The **Tutorial D** <*semijoin*>

    r1 SEMIJOIN *r2*   *or*   *r1* MATCHING *r2* is semantically equivalent to the **Tutorial D** expression

```
    ( r1 JOIN r2 ) { A, B, ..., C }
```

(where *A, B, ..., C* are all of the attributes of *r1*), and can therefore be expressed in **A**. Since this latter expression involves only operators that have already been shown to be expressible in **A,** it follows that semijoin can also be expressed in **A**. *Note:* A remark similar to the foregoing sentence applies to many of the operators still to be discussed. We will let that one sentence do duty for all.

Semidifference: The **Tutorial D** <*semiminus*>

```
    r1 SEMIMINUS r2   or   r1 NOT MATCHING r2
```

is semantically equivalent to the **Tutorial D** expression

```
    r1 MINUS ( r1 SEMIJOIN r2 )
```

Division: The **Tutorial D** <*divide*>

```
    r1 DIVIDEBY r2 PER ( r3 )
```

(a Small Divide) is shorthand for the **Tutorial D** expression

```
    r1 { A1 }
       MINUS ( ( r1 { A1 } JOIN r2 { A2 } )
                    MINUS r3 { A1, A2 } ) { A1 }
```

(where *A1* is the set of attributes common to *r1* and *r3* and *A2* is the set of attributes common to *r2* and *r3*), and can therefore be expressed in **A**. Likewise, the **Tutorial D** <*divide*>

```
    r1 DIVIDEBY r2 PER ( r3, r4 )
```

(a Great Divide) is shorthand for the **Tutorial D** expression

```
    ( r1 { A1 } JOIN r2 { A2 } )
        MINUS ( ( r1 { A1 } JOIN r4 { A2, A3 } )
                      MINUS ( r3 { A1, A3 }
                            JOIN r4 { A2, A3 } ) ) { A1, A2 }
```

(where *A1* is the set of attributes common to *r1* and *r3, A2* is the set of attributes common to *r2* and *r4,* and *A3* is the set of attributes common to *r3* and *r4*), and can therefore be expressed in **A**.

Extension: Let PLUS be a relation constant with heading

```
    { X INTEGER, Y INTEGER, Z INTEGER }
```

and with body consisting of all tuples such that the Z value is equal to the sum of the X and Y values. Then the **Tutorial D** <*extend*>

```
    EXTEND r ADD ( A + B AS C )
```

(where we assume without loss of generality that *A* and *B* are attributes of *r*[8] of type INTEGER) is semantically equivalent to the **Tutorial D** expression

```
r JOIN ( PLUS RENAME ( X AS A, Y AS B, Z AS C ) )
```

and can therefore be expressed in **A**. Analogous equivalents can be provided for all other forms of <*extend*> (including forms in which the <*exp*> in the <*extend add*> is tuple- or relation-valued). For example, the **Tutorial D** expression

```
EXTEND r ADD ( TUPLE { A A, B B } AS C )
```

is semantically equivalent to the **Tutorial D** expression

```
r JOIN s
```

where *s* is a relation with heading

```
{ A TA, B TB, C TUPLE { A TA, B TB } }
```

containing exactly one tuple for each possible combination of *A* and *B* values, in which the *C* value is exactly the corresponding <*A,B*> tuple (*TA* and *TB* here being the types of attributes *A* and *B,* respectively).

Restriction: Let ONE be a relation constant with heading

```
{ X INTEGER }
```

and with body consisting of a single tuple, with X value one. Then the **Tutorial D** <*where*>

```
r WHERE A = 1
```

(where *A* is an attribute of *r* of type INTEGER) is semantically equivalent to the **Tutorial D** expression

```
r JOIN ( ONE RENAME ( X AS A ) )
```

and can therefore be expressed in **A**.

Now let GT be a relation constant with heading

```
{ X INTEGER, Y INTEGER }
```

and with body consisting of all tuples such that the X value is greater than the Y value. Then the **Tutorial D** <*where*>

```
r WHERE A > B
```

(where *A* and *B* are attributes of *r* of type INTEGER) is semantically equivalent to the **Tutorial D** expression

```
r JOIN ( GT RENAME ( X AS A, Y AS B ) )
```

and can therefore be expressed in **A**.

[8] If they are not, we can effectively make them so by means of appropriate joins. An analogous remark applies to many of our examples; for brevity, we will not make it every time.

More generally, consider the **Tutorial D** <*where*> *r* WHERE *x*, where *x* is an arbitrarily complex <*bool exp*>. Let *A, B, ..., C* be all of the attributes of *r* mentioned in *x*. Let *rx* be a relation whose body consists of all tuples of the form <*A,B,...,C*> that satisfy *x*. Then *r* WHERE *x* is equivalent to *r* JOIN *rx*.

Summarization: The **Tutorial D** <*summarize*>

```
    SUMMARIZE r1 PER ( r2 ) ADD ( ss ( exp ) AS Z )
```

(where *r2* has attributes *A, B, ..., C; r1* has the same attributes and possibly more; and *ss* is any <*summary spec*> except COUNT, EXACTLY, or EXACTLYD) is semantically equivalent to the **Tutorial D** expression

```
    ( EXTEND r2 ADD
    ( r1 JOIN RELATION { TUPLE { A A, B B, ..., C C } } AS Y,
      agg ( ( EXTEND Y ADD ( exp AS X ) ) { X }, X ) AS Z ) )
    { ALL BUT Y }
```

(where *agg* is identical to *ss* unless *ss* is COUNTD, SUMD, or AVGD, in which case *agg* is COUNT, SUM, or AVG, respectively, and where the projection over *X* in the third line is included only if *ss* is COUNTD, SUMD, or AVGD), and can therefore be expressed in **A**.

Similarly,

```
    SUMMARIZE r1 PER ( r2 ) ADD ( ss ( exp1, exp2 ) AS Z )
```

(where *ss* is therefore EXACTLY or EXACTLYD) is semantically equivalent to the **Tutorial D** expression

```
    ( EXTEND r2 ADD
    ( r1 JOIN RELATION { TUPLE { A A, B B, ..., C C } } AS Y,
      EXACTLY ( exp1,
                 ( EXTEND Y ADD ( exp2 AS X ) ) { X }, X ) AS Z ) )
    { ALL BUT Y }
```

(where the projection over *X* in the fourth line is included only if *ss* is EXACTLYD), and can therefore be expressed in **A**.

Similarly,

```
    SUMMARIZE r1 PER ( r2 ) ADD ( COUNT ( ) AS Z )
```

is semantically equivalent to the **Tutorial D** expression

```
    ( EXTEND r2 ADD
    ( r1 JOIN RELATION { TUPLE { A A, B B, ..., C C } } AS Y,
      COUNT ( Y ) AS Z ) )
    { ALL BUT Y }
```

and can therefore be expressed in **A**.

Note: Each of the foregoing SUMMARIZE expansions involves an invocation of some aggregate operator (COUNT, for example, in the third case). But we have already seen, in the section "Treating Operators as Relations," that read-only operators can be implemented in **A,** and our conclusion that SUMMARIZE can also be expressed in **A** is thus not undermined by our reliance on those aggregate operators in the expansions.

Wrapping and unwrapping: The **Tutorial D** <*wrap*>

```
    r WRAP ( { A, B, ..., C } AS X )
```

is shorthand for the **Tutorial D** expression

```
    ( EXTEND r ADD ( TUPLE { A A, B B, ..., C C } AS X ) )
                                           { ALL BUT A, B, ..., C }
```

and can therefore be expressed in **A**. Likewise, the **Tutorial D** <*unwrap*>

```
    r UNWRAP ( X )
```

is shorthand for the **Tutorial D** expression

```
    ( EXTEND r ADD ( A FROM X AS A,
                     B FROM X AS B,
                         ......
                     C FROM X AS C ) ) { ALL BUT X }
```

(where *A, B, ..., C* are all the attributes of *X*), and can therefore also be expressed in **A**.

Grouping and ungrouping: Let relation *r* have attributes *A, B, ..., C, D, E, ..., F*. Then the **Tutorial D** <*group*>

```
    r GROUP ( { D, E, ..., F } AS X )
```

is shorthand for the **Tutorial D** expression

```
    ( EXTEND r
      ADD ( r AS RR ,
            RELATION { TUPLE { A A, B B, ..., C C } } AS TX ,
            RR COMPOSE TX AS X ) )
    { A, B, ..., C, X }
```

(where *RR* and *TX* are attribute names not already appearing in *r*), and can therefore be expressed in **A**. Likewise, the **Tutorial D** <*ungroup*>

```
    r UNGROUP ( X )
```

(where *r* has attributes *A, B, ..., C,* and *X,* and *X* in turn is a relation-valued attribute with attributes *D, E, ..., F*) is shorthand for the **Tutorial D** expression

```
    ( EXTEND ( r COMPOSE s )
      ADD ( D FROM Y AS D, E FROM Y AS E, ..., F FROM Y AS F ) )
    { A, B, ..., C, D, E, ..., F }
```

where *s* is a relation with heading

```
    { X RELATION { D, E, ..., F }, Y TUPLE { D, E, ..., F } }
```

and with body containing every possible tuple such that the *Y* value (a tuple) is a member of the body of the *X* value (a relation). It follows that the original <*ungroup*> can be expressed in **A**.

***** End of Appendix A *****