EXTENDED DOMAIN TYPES AND SPECIFICATION OF USER DEFINED OPERATORS

by

Eugene Wong

Memorandum No. UCB/ERL M85/3

13 February 1985

EXTENDED DOMAIN TYPES AND SPECIFICATION OF USER DEFINED OPERATORS

by

Eugene Wong

Memorandum No. UCB/ERL M85/3

13 February 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Extended Domain Types and Specification of User Defined Operators[1]

Eugene Wong[2]

University of California, Berkeley

1. Introduction

Currently, the domains of the relations in INGRES, as in most relational systems, are restricted to be numbers or character strings (hencefoth referred to as "primitive" data types). This restriction substantially limits the power of data manipulation in many cases. In this paper we propose to extend domain types in two significant ways: (a) as user-defined data types, and (b) as "relations." In addition, a means for operators to be defined on data types is also provided. While other suggestions for supporting user defined operators [ROW79] and extended domains [STO84] have been made, our approach differs significantly in the closure that it achieves. For example, user defined operators are realized within the query language without appealing to any host language procedures. While some measure of generality is sacrificed in the process, one gains a much needed degree of control over what the users define. Since the semantics of user defined operators are now understood by the DBMS, unified optimization can be undertaken with potential for significant performance gains.

At first glance, one might think that operators definable within the query language would be exceedingly restricted. We do not think this is the case,, and illustrate the power of our approach by applying the syntax to an extensive list of examples from geometric data.

---

[1]Research supported by the National Science Foundation under Grant ECS-8300463.

[2]Dept. EECS, University of California, Berkeley, CA 94720.

2. Extended Domain Types

In existing INGRES, relations are defined by "create" commands [ING79] of the form:

create relname(domain=format{,domain=format})

where "format" is one of the currently recognized primitive data types: {i1,i2,i4,f4,f8,c1-c255}. In addition, we assume that "binary" is also an accepted format. There are at least three commonly occurring situations for which these do not suffice. The three are:

(a) Vectors: For this case, each element of the domain is an n-tuple, n being fixed, e.g., complex numbers or coordinates of points in the plane. In addition, the values may be constrained (e.g., to be positve). While such domains could be represented by multiple domains, as is done currently, we need to treat each vector as an atomic object both in data manipulation operations and for defining new operators.

(b) Entities: Here, each domain value is an entity, e.g., "dept" in an employee relation, or "manager" in a department relation. While we can usually get by with using a numerical identifier as a surrogate for the entity, doing so causes undesirable linguistic complications, both syntactically and semantically.

(c) Sets: Here, each domain value is a "set" of entities or values, the size of the sets varying from set to set. For example, "children of employee" in an employee relation is such a set-valued domain. This situation corresponds to that of "repeating groups" in traditional data processing.

We propose that "create" statements be modified as follows:

create relname (domname = dtype {, domname = dtype})

where "dtype" (domain name) can be one of the following cases:

(a) dtype = format, where format is one of the existing primitive data types.

(b) dtype = typename, where "typename" is the name of a user defined data type specified by a "define type" statement with syntax yet to be specified.

(c) dtype = relname, where relname is the name of a relation.

(d) dtype = relname using domname, where "domname" is the name of a domain in the relation referenced by "relname."

Case (a) is what exists now. Case (b) corresponds to the "vector" case. Cases (c) and (d) represent "relations as a domain type," and corresponds to "entities" and "repeating groups" respectively.

3. User Defined Data Types

A vector domain type is defined by using the following syntax:

define type typename(domain = typenamelformat {, domain = typenamelformat})
[where qualification]

For example, we can define "date" by:

define type date(year = i2, month = i1, day = i1)
where 1 ≤month ≤12 and 1 ≤day ≤31

and "position" on the globe by:

define type position(longitude = c_coord, lattitude = c_coord)
define type c_coord (hemisph = binary, angle = i1)
where angle <= 90

The syntax is designed to serve two purposes: (a) to allow multiple domains to be treated atomically, and (b) to allow constraints to be expressed on a domain. We note that a "type" so defined is a "pseudo relation" in the sense that it represents the cartesian product of its

domains when no qualification is given, and the subet of the cartesian product defined by the qualification when one is stated. In either case, no relation need be stored.

A relation with a vector-valued domain is always equivalent to one containing only primitive domains, together with a set of integrity constraints representing the the qualification. For example, a domain: ship-date = date can be expressed as (ship-year = i2, ship-month = i1, ship-day = i1) with integrities on ship-month and ship-day. However, to do so would mean a loss of atomicity of the domain ship-date. Further, since integrity constraints are expressed on relations, not domains, every time the type "date" is used, the integrity would have to be restated. The importance of atomicity lies not merely in liguistic economy. Our proposed means for supporting user-defined operators depends on it in a crucial way.

4. Relation as a Data Type

4.1. Entities

First, consider the case of "entities" as a domain type. This case is represented by : dtype = relname, where an identifier in the relation referenced by "relname" is presumed. For example, a pair of relations "employee" and "department" might be defined together as follows:

create emp(ID = eno = i2, ename = c20, salary = i4, works = dept)

create dept(ID = dno = i2, dname = c10, mgr = emp)

where we have introduced the keyword "ID" to denote an identifier.

We note that in our example, the two relations are mutually dependent in their definition. While it does not mean that recursion is involved, it does pose a minor problem in name recognition since whichever relation is first defined it would involve an unrecognized name. This can be dealt with in various ways, e.g., by requiring they be defined in a single

transaction.

4.2. Repeating Groups

The case of "repeating groups" is represented by: dtype = relname using domname. A value of of such a domain is the set of all tuples from the relation referenced by "relname" that have the same value in the "using clause" domain. The following "create" statements illustrate the use of this syntax:

```
create part(ID = pno = i2, pname = c10, colorcode = i1, size = i1)
create supplier(ID = sno = i2, sname = c10, city = c10, poh = inventory using S)
create inventory(S = supplier, P = part, qoh = i2)
```

The domain "poh" (parts on hand) is a repeating group domain. A value of this domain is the set of all tuples in "inventory" that share a common S value (which in turn is an entity "supplier"). As a second example, consider a domain "dep" (dependents) in an employee relation:

```
create emp(ID = eno = i2, ename = c20, salary = i4, works = dept,
        dep = edep using E)
create edep(E = emp, dname = c20, relationship = i1)
```

Here, a value of the "dep" domain in "emp" consists of the set of tuples of "edep" sharing the same E value (presumably, the same emp).

5. Retrieval

Using the extended domain types and a convention due to Zaniolo [ZAN83], we can gain a tremendous simplification in QUEL statements. The convention is that: if x is a tuple then x.D is its value in domain D, which may be again a tuple or a set of tuples. In that case, x.D.C

stands for the value (or set of values) of x.D in domain C, and so forth. The effect of the convention is to reduce the number of tuple-variables that need to be introduced and to eliminate the need for explicit join-clauses. For example, under the proposed extension the following query is a legal expression:

retrieve (emp.ename) where emp.salary > emp.works.mgr.salary

In current QUEL, it would probably be stated as:

range of e is emp
range of m is emp
range of d is dept
retrieve (e.ename) where e.dept = d.dno and d.mgr = m.eno
and e.salary > m.salary

As a second example, consider finding those suppliers in New York who stock widgets in quanties greater than 1000. This can now be stated as:

retrieve (supplier.sname) where city = "New York" and supplier.inventory.qoh > 1000 and supplier.inventory.part.pname = "widget"

A great advantage of our extension is that data at different levels of detail can be seen and quantified simultaneously in a natural way. It is particularly well suited for "forms" interfaces that allow the details of the different levels to be successively displayed quickly and easily.

6. Interpretation of Extended Types as Views

A relation with a non-primitive domain can be thought of as a "view" on base relations with only primitive domains as follows:

(a) For the "vector" case, the domain can be replaced by the multiple domains

specified the "define type" statement with the qualification term represented by integrity constraints. Thus for example, "number = cx8" becomes "number.real = f8, number.imag = f8."

(b) For the "entity" case, the domain can be replaced by the "identifier" of the entity. For example, "mgr = emp" becomes "mgr.eno = i2."

(c) For the case of repeating groups, we can replace the nonprimitive domain by the domain referenced in the "using clause," repeating the step if necessary until a primitive domain is obtained. For example, "poh = inventory using S" becomes "poh.S.sno = i2."

A major consequence of the view interpretation is that any query (retrieval or update) on a relation involving extended domain types can be translated into a query on base relations involving only primitive domain types. However, the resulting query need not be legal unless any operator defined on the extended domains is translatable into recognized operators on primitive domains. Our proposed facility for defining operators will be designed to ensure this.

7. Updates

Since with extended domains a relation can be unnormalized, updates are potentially troublesome. However, the interpretation of extended domains as views provide us with a fallback position. All we need to ensure is that every update involving an extended domain can be translated into a view update, which is then accepted or rejected according to the existing criteria on acceptable view updates. This is a reasonable but somewhat conservative position.

For "append" and "delete" the conservative position means that there would be no propagated updates. For example, it would not be possible to use "append to emp" to add to the "dep" domain.

Propagated replacements are possible. For example,

replace emp(emp.works.mgr.ename = "Jones") where emp.ename = "Smith"

will change the name of the manager of Smith's dept to "Jones" (probably an erroneous expression for "move Smith to Jones' dept"). We note that even when it is unambiguious a view update often has unintended sideeffects. For example,

replace (emp.member-of.mgr.eno=2031) where emp.ename="joe"

is unambiguious, but has side effects that the user may not be aware of. Not only is the manager for "joe" changed, but so is the manager for everyone in "joe's" department. On the other hand, the statement:

replace (emp.dep.dname="bobby") where emp.dep.dname="robert"

is unambiguious and has no side effects. It is possible that updates involving extended domains may lead to an increase in updates with unintended side effects. If that proves to be true, it may require a test to determine whether an update has side effects, and to require an over-ride before such updates are executed.

8. Defining Operators

An operator is a function, and the conventional mathematical notation for a function is:

$z = f(x,y,\cdots)$

where f denotes a function mapping variables x,y, etc. into a target variable z. The information that we need for each variable is (symbol,type), and for the function f we need to specify its name and a formula for computation. We propose to incorporate these items of information in a "define oper" statement as follows:

define oper opname(variable=type{,variable=type})

```
as tvar=type
where tvar.dom=formula [if condition
[else tvar.dom=formula [if condition] ] ]
```

The "formula" term is any expression involving elsewhere defined operators, including those defined on primitive types (e.g., arithmetical operator). The "condition" is any legal QUEL qualification, which may involve operators on user defined types. As an example, consider the operation of translating a "point" in the plane. The following statements define the type "point" and the operator "translation:"

```
define type point(xcoord=i4, ycoord=i4)
define oper translation(p=point, a=i4,b=i4)
as p1 = point
where p1.xcoord = p.coord+a and p1.ycoord = p.ycoord+b
```

As a second example, consider the following defiinition for complex numbers and their multiplication:

```
define type cf4(real=f4, imag=f4)
define oper cmult(u=cf4, v=cf4)
as z = cf4 where z.real = u.real*v.real - u.imag*v.imag
and z.imag = u.real*v.imag + u.imag*v.real
```

9. Some Geometric Types and Operators

Geometric data provide a rich source of examples for datatypes and operators that would supply a reasonable test for the power and flexibility of our proposed approach. In this section the definition facility that we have proposed is applied to a rather extensive list of geometric examples. In and of itself, the list may be useful in applications.

The types that we shall define are as follows:

point

line (infinite and oriented)

line-segment

triangles

rectangles

polygons

9.1. Points and Lines

"Point" and "line" are basic geometric types, and are defined as follows:

```
define type point (xcoord=f4, ycoord=f4)
define type line (phase=f4, dist=f4)
```

where "xcoord" and "ycoord" are self-explanatory, "phase" is the angle (0 to 360 degrees) that the line makes with the x-axis, and "dist" is the distance from the origin to the line. Of the numerous ways that a line can be parameterized, our choice enjoys some geometric claims of being the best. These claims will be tested by the simplicity with which operators are defined.

For the basic types "point" and "line" the following operators are natural ones:

norm (of point as vector)

```
define oper norm (p=point) as n=f4
where n=(p.xcoord**2+p.ycoord**2)**0.5
```

argument (of point in polar coord.)

```
define oper arg (p=point) as a=f4
where a=arctan(p.ycoord/p.xcoord) if p.xcoord>0 and p.ycoord>0
```

```
else a=arctan(p.ycoord/p.xcoord)+180 if p.xcoord<0
else a=arctan(p.ycoord/p.xcoord)+360 if p.xcoord>0 and p.ycoord<0

                                    translate (a point)
define oper translate (p=point,a=f4,b=f4) as p1=point
where p1.xcoord=p.xcoord+a and p1.ycoord=p.ycoord+b

                                    difference (between points)
define oper diff (p1=point,p2=point) as p
where p = translate(p1, p2.xcoord, p2.ycoord)

                                    move (parallel, of line)
define oper move (l=line,d=f4) as nl=line
where nl.phase=l.phase and nl.dist=l.dist+d

                                    rotate (a line)
define oper rotate (l=line,ph=f4) as nl=line
where nl.phase=l.phase+ph and nl.dist=l.dist

                                    intersect (whether two lines)
define oper intersect (l1=line,l2=line) as z=binary
where z=1 if l1.phase != l2.phase or l1.dist=l2.dist

                                    intersection (point of, between lines)
define oper intersection (l1=line,l2=line) as p=point
where p.xcoord=(l1.dist*cosine(l2.phase)-l2.dist*cosine(l1.phase))/
        sine(l1.phase-l2.phase)
and p.ycoord=(-l1.dist*sine(l2.phase)+l2.dist*sine(l1.phase))/
```

sine(l1.phase-l2.phase)

contain (point in line)

define oper **contain** (l=line,p=point) as binary

where z=1 if p.xcoord*sine(l.phase)+p.ycoord*cosine(l.phase)=l.dist

else z=0

side (which, lies the point)

define oper **side** (p=point,l=line) as s=binary

where s = 1 if 0<=arg(p)-l.phase<=180

else s = 0

distance (point from line)

define oper **distance** (p=point,l=line) as d=f4

where d=(p.xcoord*sine(l.phase)+p.ycoord*cosine(l.phase))-l.dist

separation (between points)

define oper **separ** (p1=point,p2=point) as d=f4

where d=norm(translate(p1,-p2.xcoord,-p2.ycoord))

colinear (line defined by two points)

define oper **colinear** (p1=point,p2=point) as z=line

where z.phase=arg(diff(p2,p1))

and z.dist=(p2.xcoord*p1.ycoord - p1.xcoord*p2.ycoord)/separ(p1,p2)

As an example, sonsider the following relation:

create city(cid = i2, cname = c10, state = c10, location = point)

To find the distance from Chicago to New York, we can use the query:

```
range of c is city
range of c1 is city
retrieve (d = distance(c1.location, c2.location)) where c.cname = "Chicago"
and c1.cname = "New York"
```

Line segments can be parameterized in several ways. One natural parameterization is by the initial and final points of each segment. Thus, segments can be defined as follows:

```
define type segment (first=point,last=point)
```

The following operators involve line segments:

```
define oper length (s=segment) as z=f4
where z=separ(s.first,s.last)

define within (s=segment) as z=line
where z=colinear(s.first,s.last)

define oper on (p=point,s=segment) as z=binary
where z=1 if separ(p,s.first)+separ(p,s.last)=length(s)
else z=0

define oper cross (s1=segment,s2=segment) as z=binary
where z=1 if intersect(s1.within,s2.within)=1
and on(intersection(within(s1),within(s2)),s1)=1
and on(intersection(within(s1),within(s2)),s2)=1

define oper connect (p1=point,p2=point) as s=segment
where s.first=p1 and s.last=p2
```

As an application of the type "segment," consider the relation "route" deined as follows:

```
create route (rid = i2, sid =i2, section = segment)
```

To find those routes that include a section linking "Denver" and "Omaha," we can write:

```
range of c is city
range of c1 is city
range of r is route
retrieve (r.rid) where r.section = connect(c.location,c1.location)
and c.cname = "Denver" and c1.cname = "Omaha"
```

9.2. Two Dimensional Shapes

The two-dimensional shapes that we shall consider are: *trainagles, rectangles, and polygons*. The first two are vectors, while polygons with variable number of sides must be considered as repeating groups. We define "triangles" as follows:

```
define type triangle (v1=point,v2=point,v3=point)
```

where vi's are the three vertices of the triangle. For a triangle with sides a, b, and c, its area is given by the formula:

$$A = \frac{1}{4}\sqrt{a^2(b^2+c^2-a^2)+b^2(a^2+c^2-b^2)+c^2(a^2+b^2-c^2)}$$

Since the three sides of a triangle are the distances between its vertices, we can easily (though tediously) express the area as an operator as follows:

```
define oper area (t = triangle) as A = f4
where A = .25 * ((
(dist(t.v1,t.v2)**2) * (dist(t.v2,t.v3)**2 + dist(t.v1,t.v3)**2 - dist(t.v1,t.v2)**2)
+ (dist(t.v2,t.v3)**2) * (dist(t.v1,t.v2)**2 + dist(t.v1,t.v3)**2 - dist(t.v2,t.v3)**2)
+ (dist(t.v1,t.v3)**2) * (dist(t.v1,t.v2)**2 + dist(t.v2,t.v3)**2 - dist(t.v1,t.v3)**2)
```

)**.5)

Another example of an operator involving a triangle is "enclose" defined as follows:

define oper **enclose** (t = traingle, p = point) as e = binary

where e = 1 if side(p, colinear(t.v1, t.v2)) and side(p, colinear(t.v2, t.v3))

and side(p, colinear(t.v1, t.v3)) =1

else e = 0

There are many ways of defining a rectangle, but most obvious ones are either redundant or require constraints. For example, it can be uniquely defined by three of its vertices, but the six parameters involved are one too many and the vertices are constrained to make a right triangle. One way of defining a rectangle that is free of both redundancy and constraints is by a triplet: (point, line, positive number). The rectangle in question can be constructed by projecting the point orthogonally to the line and then folowing the line in its orientation for a distance equal to the positive number. Thus, we have:

define type **rectangle** (V = point, L = line, D = f4)

and the area of a rectangle is defined by:

define oper **AREA** (r = rectangle) as A = f4 where A = distance(r.P, r.L)*D

Since polygons must be dealt with as repeating groups, the vehicle for doing so is via relations. An implication of this is that we cannot define all polygons at once as a type, but must represent the ones that are needed as a relation and store it. To represent a polygon, we again encounter the problem of doing it without redundancy or constraint. One way is as follows: We first represent a convex polygon by a set of lines, the polygon being the intersection of the half-spaces defined by the lines. Then a general polygon, convex or not, is represented as the algebraic sum of convex poygons. For example, we might write:

create polygon(pid=i2, side=line)

A map showing the states would then be represented by:

```
create map(state=c10,shape=polygon using pid)
```

or the map could be combined withother information as follows:

```
create states(sname = c10, population = i4, governor = c20,
        map = polygon using pid)
```

Though advantageous in some sense, the representation of a convex polygon as the intersection of half spaces also has disadvantages. For example, the computation of area is difficult to define. A alternative way is to represent a polygon, convex or not, as a special case of an algebraic sum of triangles, e.g.,

```
create polygon(pid=i2, sign=binary, component=triangle)

create states(sname=c10, population=i4, governor=c20, map=polygon using pid)
```

Then, to find the area of each state, we can use the query:

```
range of s is states
retrieve (s.sname, size=sum( area(s.map.component) by s.sname))
```

This might be viewed as the penultimate example, since it combines many of the features that make QUEL powerful: aggregate function, relation as a data type, user-defined operator, and nesting of these constructs.

10. Conclusion

In this paper we propose an extension to QUEL that consists of: (a) allowing vectors and relations to be domain types, and (b) allowing vectors to be the basis of a facility for user-defined operators. Vectors together with the operators defined on them provide a way of significantly extending the semantics of INGRES. "Relation as a data type" provides a means

for achieving great linguistic simplification and for allowing user expressions that more closely match user conceptualization of data.

A number of further generalizations are possible. First, we have avoided "recursion" in defining domain types. It may be a desirable feature to add especially since it is easily incorporated into the proposed syntax. For example, consider a relation:

create assembly(master = part, sub = assembly)

To do a "parts explosion," one need merely to print out the relation "assembly."

A second interesting direction of generalization is to generalize "aggregation" operators to include user-defined set operators that can be qualified. For example, "the minimum distance between 'New York' and 'Denver' via interstate routes" might be such an operator. While we believe that the building blocks of doing this are now available the details are not yet at hand.

A third area of generalization concerns geometric data. Polygons are an example of collection of elementary geometric elements. More generally, one can define "point groups," "line groups" and "segment groups." A polygonal path, for example, is a segment group with a constraint. In terms of the constructs that we have introduced in this paper, these are all repeating groups, and can be used as domain types. Finally, in addition to geometrical data, textual data also provide interesting examples of both "types' and "operators." Some of these are given in [WON83].

References

[ING79] Woodfill, J. et. al., "INGRES Reference Manual," Electronics Research Laboratory Memorandum M79/43, University of California, Berkeley, CA, May 1979.

[ROW79] Rowe, L. and Schoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference, Boston, MA, May 1979.

[STO84] Stonebraker, M. et. al., "QUEL as a Data Type," Proc. 1984 ACM-SIGMOD Conference, Boston, MA, June 1984.

[WON83] Wong, E., "Semantic Enhancement through Extended Relational Views," Proc. Second International Conf. on Databases (ICOD-2), Cambridge, England, September, 1983.

[ZAN83] Zaniolo, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference, San Jose, CA, May 1983.