1. Introduction

These are the formal definitions of an Extended Relational Algebra in a style similar to Algebra A in Appendix A of DTATRM by D&D. The operators of this ERA are used to construct shorthands, intended to form the basis of a relational programming language. It should be noted that Appendix A takes the reduction path from language shorthand to algebra operation. Here the path is taken in the opposite direction, defining algebraic operations from which the shorthands of a language may be constructed.

These operators do not provide or require any particular type system. Any use of the algebra is dependent on an external provider of a type system and functions over those types. Shorthands such as `GROUP` and `UNGROUP`, `WRAP` and `UNWRAP` require that relations and tuples be provided by that type system.

The naming conventions are adapted from Appendix A. In what follows:

A is an attribute name
T is a type
v is a value
<A,T> is an attribute (member of a heading)
<A,T,v> is an attribute value (member of a tuple)
r is a relation
Hr is the heading of r
Br is the body of r
tr is a tuple of r

2. Basics from Appendix A

These are the basic 5 definitions adapted from Appendix A.

a) NOT

Given a relation `r` then `s = NOT(r)` is defined as follows.

```
Hs = Hr
Bs = { ts : ∃ tr ∉ Br, 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`.

The **`NOT`** operator is used in shorthands that involve negation, such as `ANTIJOIN`, `MINUS`.

b) REMOVE

Given a relation `r` and a type `T` such that `<A,T> ∈ Hr` then `s = REMOVE(A)` is defined as follows.

```
Hs = Hr minus { <A,T> }
Bs = { ts : ∃ tr ∉ Br, ∃ v ∈ T, <A,T,v> ∈ tr, 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`.

The **`REMOVE`** operator is used in shorthands that remove one or more attributes, such as `PROJECT`.

c) RENAME

Given a relation `r`, a type `T` such that `<A,T> ∈ Hr` and no type `T` such that `<B,T> ∈ Hr` then `s = RENAME(A,B)` is defined as follows.

```
Hs = ( Hr minus { <A,T> } ) union { <B,T> }
Bs = { ts : ∃ tr ∈ Br, ∃ v ∈ T, <A,T,v> ∈ tr,
   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>`.

The **RENAME** operator is used in shorthands that rename one or more attributes, such as `RENAME`.

d) AND

Given relations `r1` and `r2` such that `<A,T1> ∈ Hr1 and <A,T2> ∈ Hr2 implies T1 = T2` then `s = AND(r1,r2)` is defined as follows.

```
Hs = Hr1 union Hr2
Bs = { ts : ∃ tr1 ∈ Br1, ∃ tr2 ∈ Br2, ts = tr1 union tr2 }
```

The **AND** operator is relational conjunction, which is usually known as a natural join. 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.

The **AND** operator is used in join shorthands such as `JOIN`, `SEMIJOIN`, `ANTIJOIN`, `COMPOSE`, `TIMES`.

e) OR

Given relations `r1` and `r2` such that `<A,T1> ∈ Hr1 and <A,T2> ∈ Hr2 implies T1 = T2` then `s = OR(r1,r2)` is defined as follows.

```
Hs = Hr1 union Hr2
Bs = { ts : ∃ tr1, ∃ tr2, ( tr1 ∈ Br1 or tr2 ∈ Br2), ts = tr1 union tr2 }
```

The **OR** operator is relational disjunction, being a generalization of what is usually known 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`.

The **OR** operator is used in set shorthands such as `UNION`, `MINUS`, `SYMDIFF`, `INSERT`.

3. Relational Constant

This is the formalisation of a relational constant or 'relcon', and relies on some previously defined function `f`. Separate formalisations are provided for monadic and dyadic functions, in both predicate and value-returning forms. Extending them to n-adic functions is left as an exercise.

a) Monadic predicate

Given attribute names `X`, types `Tx` and function `f` with the type signature `Tx->Boolean` then `s = RELCON(X,f)` is defined as follows.

```
Hs = {<X,Tx>}
```

```
Bs = { ts : ∃ vx ∈ Tx, f(vx) = true, ts = {<X,Tx,vx>} }
```

b) Monadic value

Given attribute names `X,Y`, types `Tx,Ty` and function `f` with the type signature `Tx->Ty` then `s = RELCON(X,Y,f)` is defined as follows.

```
Hs = {<X,Tx>, <Y,Ty>}
Bs = { ts : ∃ vx ∈ Tx, ∃ vy ∈ Ty, f(vx) = vy,
   ts = {<X,Tx,vx>, <Y,Ty,vy>} }
```

c) Dyadic predicate

Given attribute names `X,Y`, types `Tx,Ty` and function `f` with the type signature `Tx->Ty->Boolean` then `s = RELCON(X,Y,f)` is defined as follows.

```
Hs = {<X,Tx>, <Y,Ty>}
Bs = { ts : ∃ vx ∈ Tx, ∃ vy ∈ Ty, f(vx,vy) = true),
   ts = {<X,Tx,vx>, <Y,Ty,vy} }
```

d) Dyadic value

Given attribute names `X,Y,Z`, types `Tx,Ty,Tz` and function `f` with the type signature `Tx->Ty->Tz` then `s = RELCON(X,Y,Z,f)` is defined as follows.

```
Hs = {<X,Tx>, <Y,Ty>, <Z,Tz>}
Bs = { ts : ∃ vx ∈ Tx, ∃ vy ∈ Ty, ∃ vz ∈ Tz, f(vx,vy) = vz),
   ts = {<X,Tx,vx>, <Y,Ty,vy, <Z,Tz,vz>} }
```

[Note: for the relcon PLUS in App-A, types `Tx`, `Ty` and `Tz` are all INTEGER, and the function `f` is the scalar operator "+".]

The **RELCON** operator predicate forms are used together with **AND** in shorthands such as `WHERE`/`RESTRICT`.

The **RELCON** operator value-returning forms are used together with **AND** in shorthands such as `EXTEND`, `SUMMARIZE`, `UPDATE` and `DELETE`.

4. Aggregation

This is the formalisation of simple aggregation, similar to SUM, MAX or MIN but allowing for a wider range of functions. It relies on some previously defined aggregating function `f`. An aggregating function has an argument which is a list of values (a 'bag' or 'multiset' rather than a 'set') and returns a single value. Extending this approach to more complex aggregations is left as an exercise.

Given a relation `r`, a type `T` such that `<A,T> ∈ Hr` and an aggregating function `f` with the type signature `T[]->Ta` then `s = AGGREGATE(r,A,f)` is defined as follows.

```
Hs = ( Hr minus {<A,T>} ) union {<A,Ta>}
Bs = { ts : ∃ v ∈ T, ∃ tr ∈ r, ∃ <A,T,v> ∈ tr, v ∈ v[],
   ts = tr minus {<A,T,v>} union {<A,Ta,f(v[])} }
```

Note: the term `v[]` should be read as: a list of the values of `v` for some value of `ts`.

The **AGGREGATION** operator is used in shorthands such as `SUMMARIZE`.

5. Transitive Closure

As transitive closure cannot be defined directly in set-builder notation, this formalisation defines it indirectly by means of a recurrence relation. This consists of a starting value (given) and a sequence of successors (defined by set-builder). The transitive closure is the fix-point union of that sequence.

The starting value ('seed') represents known edges in a directed graph; the end value is all the possible paths through the graph.

a) Simple Transitive Closure

Given a relation `r` with the heading `{<A,T>,<B,T>}` for some type `T` then the successor relation `r'` is defined as follows.

```
Hr' = Hr
Br' = { tr' : tr' ∈ Br or
   ∃ v1 ∈ T, ∃ v2 ∈ T, ∃ v3 ∈ T, ∃ v4 ∈ T,
   ∃ tr1 ∈ Br, tr1 = {<A,T,v1>, <B,T,v2>},
   ∃ tr2 ∈ Br, tr2 = {<A,T,v2>, <B,T,v3>},
   tr' = {<A,T,v1>, <B,T,v3>} }
```

Then the transitive closure `s = TCLOSE(r)` is defined as follows.

$S = r^1 \cup r^2 \cup r^3 \ldots r^\infty$

This is a linear recurrence, which can be shown to reach a fix-point termination.

The **TCLOSE** operator is used in a shorthand of the same name.

b) Extended Transitive Closure

In this case each tuple is associated with a value, and this definition relies on some previously defined dyadic function `f` that takes values of that type as its argument and return value. The value computed represents in some sense the cost of that path.

Given a relation `r` with heading `{<A,T>,<B,T>,<C,Tv>}` for some types `T` and `Tv`, and a dyadic function `f` with the type signature `Tv->Tv->Tv` then the successor relation `r'` is defined as follows.

```
Hr' = Hr
Br' = { tr' : tr' ∈ Br or
 ( ∃ v1 ∈ T, ∃ v2 ∈ T, ∃ v3 ∈ T, ∃ v4 ∈ T,
   ∃ w1 ∈ Tv, ∃ w2 ∈ Tv,
   ∃ tr1 ∈ Br, tr1 = {<A,T,v1>, <B,T,v2>, <C,Tv,w1>},
   ∃ tr2 ∈ Br, tr2 = {<A,T,v2>, <B,T,v3>, <C,Tv,w2>},
   tr' = {<A,T,v1>, <B,T,v3>, <C,Tv,f(w1,w2)} ) }
```

Then the extended transitive closure `s = ETCLOSE(r,f)` is defined as follows.

$s = r^1 \cup r^2 \cup r^3 \ldots r^\infty$

This is a linear recurrence, which can be shown to reach a fix-point termination.

The **ETCLOSE** operator is used in combination with **AGGREGATE** in a shorthand to perform Generalised Transitive Closure as defined by D&D (RM VSS 5).