TclRAL: A Relational Algebra for Tcl

Andrew Mangogna

13th Annual Tcl/Tk Conference
October 11-13, 2006

Copyright

Abstract

TclRAL is a "C" language extension to Tcl that provides commands to implement a formal relational algebra. This paper introduces TclRAL and describes the goals and background of the extension. Each command is described in detail and examples of a relational style of programming are given to help explain the operations that the extension provides. A comparison with other approaches is also provided.

Contents

List of Figures

1 Introduction

Throughout its evolution, Tcl has gained increasingly sophisticated data structures. Initially, only simple scalar variables, arrays and lists were supported in the language. Over time as ever larger applications were implemented in Tcl, more sophisticated data structures were added. Most recently, DICTIONARIES were added to the core. Additionally, TCLLIB now supports a wide array of data structures ranging from trees and graphs to matrices. These developments improve the productivity of writing larger applications in Tcl. It is in this context that TclRAL was developed.

1.1 Goals of TclRAL

TclRAL was designed to achieve the following goals:

1. To provide relation values as native Tcl objects.
2. To provide a rigorous and complete set of relational operators over the relation values.
3. To provide variables to hold relation values and a useful set of integrity constraints on the values those variables may hold.
4. To serve as a framework for exploring the use of relational concepts in Tcl programming.

1.2 Availability of TclRAL

TclRAL is an open source project housed at sourceforge.net and available under the same license as Tcl itself. See TclRAL `http://sourceforge.net/projects/tclral` for complete a complete description of the project. As of this writing, Version 0.8 is available.

2 Relational Algebra

The Relational Model of Data is the application of logic and set theory to the management of data. It is most often discussed in terms of DATABASE MANAGEMENT SYSTEMS (DBMS). Database management systems are a major application of computing technology and have tremendous commercial importance. However, relations are logical entities divorced from any particular implementation and they may be realized in many different ways. Although TclRAL shares many concepts with DBMS, it is important to distinguish carefully between the logical view of relations and the particulars of a given implementation. In this section we define the basic concepts of relations and how those concepts are represented in TclRAL. There are many good and detailed expositions of relational theory and the interested reader my wish to explore them further. Here, space limits us to just enough explanation to provide the context for discussing TclRAL.

TclRAL follows very closely the relational algebra as formulated by Date[1] and Darwin[2]. The parts of TclRAL that deal with referential integrity were patterned after the relational ideas of the Shlaer-Mellor Method, now known as Executable UML[3, 4, 5]. There are three primary entities in this formulation:

- Tuples provide the basic unit of data aggregation.
- Relations embody sets of tuples.
- Relvars are variables that specifically store relation values.

In the next sections we give an overview of these concepts as they are realized in TclRAL.

2.1 Tuple

A TUPLE consists of a heading and a set of corresponding values. The heading consists of a set of *attribute* names and corresponding *data types*. Attribute names must be distinct and every attribute is associated with a data type. The data type may be any valid Tcl data type, such as string, int, double, list, *etc*. Attribute data types may also be Tuple or Relation types, supporting a form of nesting or grouping. In a tuple value, each attribute has exactly one value associated with it. That value must be coercible into the type that is defined for the attribute.

As a well behaved Tcl value, Tuple values have a string representation. A Tuple value is represented as a three element list where the elements are:

1. The keyword, `Tuple`.
2. A list of attribute name / attribute type pairs.
3. A list of attribute name / attribute value pairs.

For example:

```
set d {
    Tuple
    {DogName string Weight double}
    {DogName Ralph Weight 4.7}
}
```

sets the variable `d` to a Tuple value that has two attributes, DogName and Weight[1].

In a sense the `Tuple` type is similar to the `dict` type and is useful in some of the same contexts. Like dictionaries, a tuple type consists of key-value pairs and each key must be a unique name. However, the primary use of tuples is as the components of relations as discussed below. At first consideration it may seem unusual to have to specify data type information when defining a Tuple. This could be considered not

---

[1] More precisely, `d` is set to a string value that if used later in a `::ral::tuple` operation will be converted into a Tuple value.

in keeping with the "Tcl way". The type of an attribute is a very important integrity constraint on the values. When an attribute of a tuple is assigned a value, that value must be coercible to the type given for the attribute. Since string is the universal type in Tcl, any value supplied for a string typed attribute is acceptable. The type binding will become more important later when we discuss computational operations on relations.

2.2 Relation

A RELATION consists of a heading and a body. The Relation heading is much like that of a Tuple heading, *i.e.* a set of attribute names and attribute data types. The body of a relation is a set of tuple values. Each attribute of each tuple in the body of a relation has a value and that value must correspond to the data type defined for the attribute. In addition each relation must have at least one IDENTIFIER[2]. An identifier is a set of attributes for which the values must be unique for all tuples in a relation. Fundamentally, relations are sets and sets do not have duplicated members. An identifier defines the set of attributes across which uniqueness is to be considered. A relation may have many identifiers, however, no identifier may have a set of attributes that is a subset of another identifier, *i.e.* the identifiers must be minimal.

Like Tuple values, Relation values also have a string representation. The string representation of a Relation value is a list of four elements:

1. The keyword, `Relation`.
2. A list of attribute name / attribute type pairs.
3. A list of identifiers, each one of which may be a list of attribute names.
4. A list of tuple values forming the body of the relation.

For example:

```
set r {
    Relation
    {DogName string Breed string Weight double}
    DogName
    {
        {DogName Fred Breed Poodle Weight 17.0}
        {DogName ChiChi Breed Dalmation Weight 22.0}
    }
}
```

defines a relation consisting of three attributes (`DogName`, `Breed` and `Weight`), a single identifier consisting of a single attribute (`DogName`) and with a body containing two tuples.

---

[2] Also known as a candidate key or a primary key. I prefer the term identifier to avoid some of the confusion surrounding the use of the term *key* in the DBMS arena.

2.3 Relvar

TclRAL defines a separate variable space for storing significant relation values. This is a bit unusual since relation values may be stored in ordinary Tcl variables also. The RELVAR variable space serves two distinct purposes:

1. Relvars provide storage locations for relation values and operators to modify in place the value contained in a relvar.
2. Relvars may have integrity constraints defined between them and TclRAL will insure that the values contained in the relvar storage satisfy the integrity constraints.

In a traditional DBMS, relvars represent the persistent storage of the database. While TclRAL does not provide for any transparent persistence, the relvars fulfill a similar role as in a DBMS. As we will see below, integrity constraints are defined on relvars and the only operators that modify a relation value in place are those operators that deal with relvars. Integrity constraints help insure that the state of the relation values held in the relvars transitions from one valid state to another valid state as the program executes. It is often the case that the relation values contained in relvars serve as the leaf terms in the algebraic expressions that compute the values of interest to the program.

The operators that deal with relation values typically exhibit CLOSURE, *i.e.* the operators take relation values as arguments and return a new relation value as the result. In that context, ordinary Tcl variables serve to hold temporary results that may be reused in other calculations or for the sake of program clarity. However, Tcl variables are very convenient and so a relvar also has an associated Tcl variable so that the familiar dollar variable reference still works ($Dog yields the relation value stored in the Dog relvar). Further, the relvar variable space uses the same NAMESPACE based naming conventions as ordinary Tcl variables. This avoids introducing new notation and makes managing a complex set of relvars easier.

2.4 Table Representation

It is convenient to represent relation values in a tabular format and TclRAL provides several commands to present relation values in a more visually intuitive table arrangement. However, we must not get too accustomed to the implied ordering of rows or columns in a table. Clearly some order must be imposed for display purposes, but that order is not part of the relation value itself. There are no operations in TclRAL that access tuples based on some notion of row index. Tuples in a relation may only be referenced by the values of their attributes. Also there are no operations that access attributes based on some notion of column index. Attributes are referenced only by their names. Clearly, the implementation stores attributes in some order within a tuple and stores tuples in some order within a relation, however, the implementation is free to rearrange that ordering in any way that it chooses. In practice, the TclRAL implementation takes some care not to arbitrarily rearrange the order of tuples and attributes based on the principle of least surprise. But logically there is no prescribed order to the tuples in a relation or to the attributes in a tuple. One very practical consequence of this is that you should not use list or string operators on the string representations of relation values. All the required operators are supplied by TclRAL and those operators will maintain the integrity and invariants of the relation values.

3 Operations

In this section we will describe the operations that are provided by TclRAL. The manual pages supplied with TclRAL give the formal and detailed syntax and semantics of each command in the package. Here we will try to focus on why particular operations are useful and how they apply to our running example. First we consider Tuple operations, then Relation operations and finally Relvar operations.

3.1 Running Example

For the purposes of this paper, we will use a small running example to illustrate the concepts of TclRAL. In this example we will consider managing a dog adoption scenario. Necessarily, the example is oversimplified to fit the constraints of this space and does not represent any real situation. We will keep track of dogs and people who own them. We are interested in knowing who owns what dogs, when they became the owner and how we might contact those owners if the need arises. We will only keep track of people who actually own dogs that they have obtained from us. So we can have the situation where a dog is not owned by anyone. We also insist that we be able to contact owners. We will contact owners by phone or email or both if the owner happens to be particularly communications enabled.

To formalize this scenario, we will use six relvars. Figure 1 shows a graphical representation of these relvars. In this notation, rectangles represent relvars. The names of the relvar's attributes are listed inside the rectangles. Those attribute names preceded by an asterisk (*) are part of an identifier and those preceded by a hyphen (-) are non-identifying attributes. The lines in the graphic represent referential integrity constraints (discussed in Section 3.5 below). Each constraint has a name (*e.g.* `R1`) that is written along the line representing the constraint. Where the line connects to a relvar, a single character represents the multiplicity and conditionality of the constraint. Relvar attributes that have a parenthetical reference to a constraint name (*e.g.* `(R2)`) are referring attributes in the named constraint. The schema of our example could be represented graphically in many ways, such as an entity-relationship diagram or even in UML. The notation shown is similar to, and borrowed from, that of Shlaer and Mellor[3]. To make our example complete we will need to populate the relvars with relation values. So we use the values as shown in Figures 2 to 7 as our example population. The representation shown is that returned from the `::ral::relformat` command. This command gives a printable string representing the relvar in tabular form. Those attributes in the heading that are part of any identifier are marked with an equal sign (=).

In the examples given below we will often refer to these relvars. TclRAL is contained in the `RAL` package which creates commands in the `::ral` namespace. As a

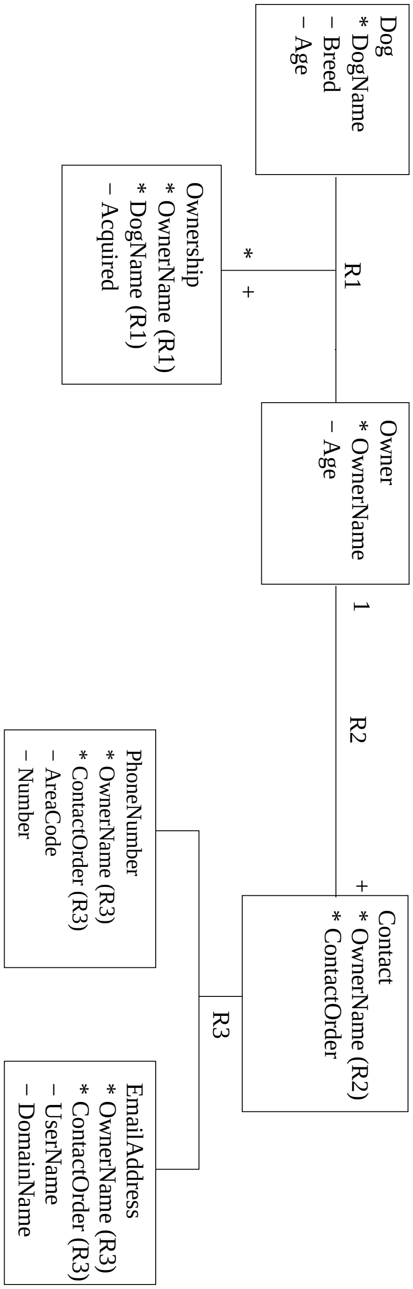

Figure 1: Graphical Representation of the Running Example

```
+=======+---------+---+
|DogName|Breed    |Age|
|string |string   |int|
+=======+---------+---+
|Fido   |Poodle   |2  |
|Sam    |Collie   |4  |
|Spot   |Terrier  |1  |
|Rover  |Retriever|5  |
|Fred   |Spaniel  |7  |
|Jumper |Mutt     |3  |
+=======+---------+---+
```

Figure 2: Dog Relvar

```
+=========+---+
|OwnerName|Age|
|string   |int|
+=========+---+
|Sue      |24 |
|George   |35 |
|Alice    |30 |
|Mike     |50 |
|Jim      |42 |
+=========+---+
```

Figure 3: Owner Relvar

```
+=========+=======+--------+
|OwnerName|DogName|Acquired|
|string   |string |string  |
+=========+=======+--------+
|Sue      |Fido   |2001    |
|Sue      |Sam    |2000    |
|George   |Fido   |2001    |
|George   |Sam    |2000    |
|Alice    |Spot   |2001    |
|Mike     |Rover  |2002    |
|Jim      |Fred   |2003    |
+=========+=======+--------+
```

Figure 4: Ownership Relvar

```
+=========+============+
|OwnerName|ContactOrder|
|string   |int         |
+=========+============+
|Sue      |1           |
|Sue      |2           |
|George   |1           |
|Alice    |1           |
|Mike     |1           |
|Mike     |2           |
|Mike     |3           |
|Jim      |1           |
+=========+============+
```

Figure 5: Contact Relvar

```
+=========+============+--------+--------+
|OwnerName|ContactOrder|AreaCode|Number  |
|string   |int         |string  |string  |
+=========+============+--------+--------+
|Sue      |1           |111     |555-1212|
|George   |1           |408     |555-2020|
|Alice    |1           |555     |867-4309|
|Mike     |1           |800     |555-3890|
|Mike     |2           |866     |555-8821|
+=========+============+--------+--------+
```

Figure 6: PhoneNumber Relvar

```
+=========+============+--------+------------+
|OwnerName|ContactOrder|UserName|DomainName  |
|string   |int         |string  |string      |
+=========+============+--------+------------+
|Sue      |2           |sue     |mymail.com  |
|Mike     |3           |mikey   |yourmail.com|
|Jim      |1           |jimbo   |domail.com  |
+=========+============+--------+------------+
```

Figure 7: EmailAddress Relvar

convenience for the examples, we will assume that the `::ral` commands have been imported in the namespace of the example.

3.2 Tuple Operations

Relations are composed of sets of tuples, so a Tuple data type is necessary to complete the ability to work with relation values. There are 15 options for the `::ral::tuple` ensemble command. It is worth grouping them into several categories. All the commands that implement tuple operations, except one, exhibit closure, *i.e.* they take Tuple valued arguments and return a new Tuple result. The exception is `tuple update` which modifies in place the tuple value contained in a Tcl variable. This command is used as part of the method in which relvars are updated as we will see below.

3.2.1 Structural Operations

The commands in this category are used to create tuples or extract subsets of the attributes or otherwise affect the attribute structure of a Tuple value.

create The `create` command creates a new tuple. This is a procedural way to create a tuple as an alternative to composing its string representation.

extend The `extend` command creates a new tuple from an existing tuple by adding attributes.

project The `project` command creates a new tuple from an existing tuple by selecting a subset of the attributes.

eliminate The `eliminate` command is complementary to `project`. It creates a new tuple by discarding attributes. Sometimes it is easier to think about which attributes are to be discarded rather than which ones are to be included.

rename The `rename` command creates a new tuple that has new names for its attributes. Multiple attribute names can be changed, but the attribute types and values remain unchanged.

```
% set t [tuple create {DogName string Breed string Age int}\
    {DogName Fido Breed Poodle Age 2}]
Tuple {DogName string Breed string Age int}
{DogName Fido Breed Poodle Age 2}
% puts [tuple eliminate $t Age]
Tuple {DogName string Breed string}
{DogName Fido Breed Poodle}
% puts [tuple rename $t DogName Name]
Tuple {Name string Breed string Age int}
{Name Fido Breed Poodle Age 2}
```

3.2.2 Attribute Access

The commands in this group are used to access the attribute values of a tuple, moving them between tuples and regular Tcl variables.

assign The `assign` command creates Tcl variables and places attribute values into them. This makes it possible to decompose a tuple into Tcl variables so that values can be used in other procedures or expressions. The interface for `tuple assign` is intended to be mnemonic of the core `lassign` command.

extract The `extract` command obtains one or more attribute values from a tuple. No variable assignments are made and the values are returned from the command.

get The `get` command returns a list of attribute name / attribute value pairs. The list is suitable to use as an argument to `array set` or can be operated on by any of the `dict` command options.

update The `update` command modifies in place a tuple value contained in a Tcl variable. This is the only tuple command that behaves in this way and it is useful in conjunction with the `relation update` command described below.

```
set t {
    Tuple
    {DogName string Breed string Age int}
    {DogName Fido Breed Poodle Age 2}
}
% tuple assign $t
3
% puts $Breed
Poodle
% puts [tuple extract $t DogName]
Fido
% array set dogtuple [tuple get $t]
% parray dogtuple
dogtuple(Age)     = 2
dogtuple(Breed)   = Poodle
dogtuple(DogName) = Fido
% tuple update t Age [expr {[tuple extract $t Age] + 1}]
Tuple {DogName string Breed string Age int}
{DogName Fido Breed Poodle Age 3}
```

3.2.3 Comparison

Only one comparison operation is defined and that is tuple equality. Because relative ordering of attributes in a tuple is not defined, tuple equality cannot be determined by simple string comparison.

equal The tuple `equal` command determines if two tuples are equal. Two tuples are equal if and only if they have the same attribute names, attribute types and attribute values. Since attribute order does not matter, equal tuples may have different string representations. Another way of putting it is that there are, in general, many string representations of any given tuple.

3.2.4 Introspection

In keeping with the usual conventions of Tcl, introspection into the structure of a tuple is supported.

degree The `degree` command returns the number of attributes in a tuple.

heading The `heading` command returns the heading of tuple.

attributes The `attributes` command returns as a list the attribute names of a tuple.

3.2.5 Composition

It is possible to have Tuple values that have attributes that are themselves Tuple valued. This possiblity makes it necessary to have operators that will construct tuple valued attributes from scalar attributes and to perform the inverse.

wrap The `wrap` command creates a new tuple that has a new tuple valued attribute composed from other attributes in the tuple.

unwrap The `unwrap` command is complementary to `wrap` and “flattens” out a tuple valued attribute.

```
% set t2 [tuple wrap $t Props {Breed Age}]
Tuple {DogName string Props {Tuple {Breed string Age int}}}
{DogName Fido Props {Breed Poodle Age 2}}
% tuple unwrap $t2 Props
Tuple {DogName string Breed string Age int}
{DogName Fido Breed Poodle Age 2}
```

3.3 Relation Operations

There are 38 options for the `::ral::relation` ensemble command. We will group the operations in order to get a handle on the large number of operators. One very important property of the `relation` ensemble commands is closure. Most of the commands take relation values as argument and return a relation value as the result (those that do not usually return simple integer or boolean values). None of the `relation` subcommands modify a relation value in place. This allows arbitrary nesting of operations in the form that we are all very familiar with in Tcl. There are commands in the `ral` package that modify values in place and they are all grouped in the `relvar` ensemble.

3.3.1 Fundamental Set Operations

TclRAL provides the usual set operations across relation values. For all these operators the relation value arguments must be of the same type. This means that they must have the same attributes and those attributes must have the same data types. These operations are:

union The command, `union`, computes the set union between two or more relation values. The tuples that are in the result are those that appear in at least one of the arguments, remembering that relation values never have any duplicates.

intersection The command, `intersect`, computes the set intersection between two or more relation values. The result contains those tuples that appear in all of the arguments.

difference The command, `minus`, computes the set difference between two relation values. The result contains those tuples in the first relation that do not appear in the second one. Note that for this operation, argument order is important.

insertion The command, `include`, returns a new relation with additional tuples inserted. This is much like union, except that an error is raised if any attempt is made to insert a duplicate tuple.

comparison The full complement of comparison between two relation values is also provided by the `is` command. Comparisons may only be made between relation values of the same type, *i.e.* those relation values that have the same attribute names and corresponding attribute data types.

- equality (==)
- inequality (!=)
- subset (<=)
- proper subset (<)
- superset (>=)
- proper superset (>)

For example, consider:

```
set newDog {
    Relation
    {DogName string Breed string Age int}
    DogName
    {
        {DogName Puffy Breed Poodle Age 1}
    }
}
% relformat [relation union $Dog $newDog]
+=======+---------+---+
|DogName|Breed    |Age|
|string |string   |int|
+=======+---------+---+
|Fido   |Poodle   |2  |
|Sam    |Collie   |4  |
|Spot   |Terrier  |1  |
|Rover  |Retriever|5  |
|Fred   |Spaniel  |7  |
|Jumper |Mutt     |3  |
|Puffy  |Poodle   |1  |
+=======+---------+---+
% relation is $Dog < [relation include $Dog\
    {DogName Puffy Breed Poodle Age 1}]
1
% relation is $newDog == $Dog
0
```

3.3.2 Restriction and Projection

A very common operation is to select some subset of tuples from a relation or some subset of attributes from a relation. Casually speaking, we are often interested in some set of rows or some set of columns. As discussed before, there are no operators in TclRAL that correspond to array indexing operations. Tuples and attributes may only be referred to by values and names, respectively. TclRAL provides the following operations for obtaining parts of a relation value:

restrict The command, `restrict`, returns a relation by selecting tuples where an arbitrary Tcl expression evaluates to true. Each tuple in the argument relation value is successively assigned to a variable and an expression is evaluated. If the expression returns true, then the tuple is included in the result. The expression is an arbitrary Tcl expression evaluated by `expr`.

restrictwith The `restrictwith` command is a variation on `restrict` where values of the attributes are split out into separate Tcl variables. This is particularly convenient in simpler cases.

choose The command, `choose`, returns a relation that contains at most one tuple whose identifying attribute values are given. In some cases the values of the attributes of an identifier are known and just that one tuple in a relation is required.

emptyof The command, `emptyof`, returns a relation that has the same heading as its argument, but whose body is the empty set. This is useful combined with, `relation include`, for building up a relation from data gathered from other sources in a program.

project The command, `project`, returns a relation that contains only the attributes given as arguments.

eliminate The command, `eliminate`, is complementary to project in that it returns a relation containing all the attributes except the ones given as arguments. Sometimes it is easier to think in terms of what needs to be discarded rather than what needs to be retained.

Consider the following examples:

```
% relformat [relation restrict $Dog d\
    {[tuple extract $d Age] < 3}]
+=======+-------+---+
|DogName|Breed  |Age|
|string |string |int|
+=======+-------+---+
|Fido   |Poodle |2  |
|Spot   |Terrier|1  |
+=======+-------+---+
% relformat [relation restrictwith $Dog {$Age <= 3}]
+=======+-------+---+
|DogName|Breed  |Age|
|string |string |int|
+=======+-------+---+
|Fido   |Poodle |2  |
|Spot   |Terrier|1  |
|Jumper |Mutt   |3  |
+=======+-------+---+
% relformat [relation choose $Owner OwnerName Mike]
+=========+---+
|OwnerName|Age|
|string   |int|
+=========+---+
|Mike     |50 |
+=========+---+
% relformat [relation project $Ownership OwnerName Acquired]
+=========+========+
|OwnerName|Acquired|
|string   |string  |
+=========+========+
|Sue      |2001    |
|Sue      |2000    |
|George   |2001    |
|George   |2000    |
|Alice    |2001    |
|Mike     |2002    |
|Jim      |2003    |
+=========+========+
% relformat [relation eliminate $Dog Age]
```

```
+=======+---------+
|DogName|Breed    |
|string |string   |
+=======+---------+
|Fido   |Poodle   |
|Sam    |Collie   |
|Spot   |Terrier  |
|Rover  |Retriever|
|Fred   |Spaniel  |
|Jumper |Mutt     |
+=======+---------+
```

3.3.3 Multiplication and Division

The operators in this section are concerned with combining two or more relation values. The archetypical operation in the category is `join`, where two relation values are merged together in a very specific manner. TclRAL provides several useful variations on this theme.

product The `times` command computes the Cartesian product over two or more relations.

join TclRAL has the natural `join` command. There are many flavors and variations on the basic `join` operator, but only natural join is supported here.

semijoin The `semijoin` command yields a join where only the attributes of one of the relations is retained. Roughly it gives the set of related tuples in another relation. In many cases, `semijoin` is a more appropriate operation than `join`.

semiminus The `semiminus` command is complementary to `semijoin`. It returns a relation containing tuples not related to another relation.

```
% relformat [relation join $Owner $Contact]
+=========+---+============+
|OwnerName|Age|ContactOrder|
|string   |int|int         |
+=========+---+============+
|Sue      |24 |1           |
|Sue      |24 |2           |
|George   |35 |1           |
|Alice    |30 |1           |
|Mike     |50 |1           |
|Mike     |50 |2           |
|Mike     |50 |3           |
|Jim      |42 |1           |
+=========+---+============+
% relformat [relation semijoin $PhoneNumber $Contact]
```

```
+=========+============+
|OwnerName|ContactOrder|
|string   |int         |
+=========+============+
|Sue      |1           |
|George   |1           |
|Alice    |1           |
|Mike     |1           |
|Mike     |2           |
+=========+============+
% relformat [relation semiminus $PhoneNumber $Contact]
+=========+============+
|OwnerName|ContactOrder|
|string   |int         |
+=========+============+
|Sue      |2           |
|Mike     |3           |
|Jim      |1           |
+=========+============+
```

division The `divide` command implements the relational divide operation. This operation is useful in contexts where you are performing "find all of" type operations. The operation is a bit complicated and involves three relations, the dividend, the divisor and the mediator. The headings of the dividend and divisor must be disjoint and the heading of the mediator must be the union of the headings of the dividend and divisor. The result (the quotient) is a new relation that has the same heading as the dividend and contains all the tuples from the dividend whose corresponding tuples in the mediator include all the tuples in divisor. An example will help clarify the situation. In this example we seek to find all the Dogs owned by both Sue and George.

```
% relformat [set dividend [relation project $Dog DogName]]\
    Dividend:
+=======+
|DogName|
|string |
+=======+
|Fido   |
|Sam    |
|Spot   |
|Rover  |
|Fred   |
|Jumper |
+=======+
Dividend:
```

```
---------
% relformat [set divisor [relation project\
    [relation restrict $Owner t\
    {[tuple extract $t OwnerName] eq "Sue" ||\
        [tuple extract $t OwnerName] eq "George"}] OwnerName]]\
    Divisor:
+=========+
|OwnerName|
|string   |
+=========+
|Sue      |
|George   |
+=========+
Divisor:
--------
% relformat [set mediator [relation eliminate $Ownership Acquired]]\
    Mediator:
+=========+=======+
|OwnerName|DogName|
|string   |string |
+=========+=======+
|Sue      |Fido   |
|Sue      |Sam    |
|George   |Fido   |
|George   |Sam    |
|Alice    |Spot   |
|Mike     |Rover  |
|Jim      |Fred   |
+=========+=======+
Mediator:
---------
% relformat [relation divide $dividend $divisor $mediator]\
    "All dogs owned by both Sue and George"
+=======+
|DogName|
|string |
+=======+
|Fido   |
|Sam    |
+=======+
All dogs owned by both Sue and George
-------------------------------------
```

Notice in this example that the quotient multiplied by the divisor yields a subset of the mediator. The key here is that the result includes only those tuples from the dividend where all of the corresponding values from the divisor are found in the mediator.

3.3.4 Computational Operations

The operators discussed so far do not do any computation. They have only provided a means to obtain subsets of relation values or build new relation values by combining or partitioning relations. In all cases the attribute values have not changed. In this section the operations evaluate expressions and place the results into relation values as new attributes.

extend The `extend` command adds new attributes to a relation and sets the values of those attributes to be the result of an expression. The expression values are computed using the core `expr` command so any computation that can be done in Tcl can be placed in the result. Extending a relation value is a tuple-wise operation. Simple units conversions are an example. If we are interested in the age of a dog in months we can obtain that by extending Dog, as in:

```
% relformat [relation eliminate\
    [relation extend $Dog d\
        AgeInMonths int {[tuple extract $d Age] * 12}]\
    Age]
+=======+---------+-----------+
|DogName|Breed    |AgeInMonths|
|string |string   |int        |
+=======+---------+-----------+
|Fido   |Poodle   |24         |
|Sam    |Collie   |48         |
|Spot   |Terrier  |12         |
|Rover  |Retriever|60         |
|Fred   |Spaniel  |84         |
|Jumper |Mutt     |36         |
+=======+---------+-----------+
```

summarize The `summarize` command allows for computations that range across multiple tuples. The concept is to select a subset of the tuples in a relation and generate the value of a new attribute as a function of the subset. Another relation is used to control how the subsets are selected and this is called the *per* relation. If the *per* relation is a projection of non-identifying attributes from the original relation value, then there will be, in general, multiple tuples in the original relation that have values that correspond to the values of the *per* relation. The net effect is that the *per* relation controls the selection of the subsets. Consider the question of finding the number of dogs acquired in each year. To accomplish this we would like to consider the `Ownership` relation in such a way that those tuples that have the same value of `Acquired` are together. Then it is a simple matter of taking the cardinality of that subset of `Ownership`. This is what the summarize command accomplishes.

```
% relformat [relation summarize $Ownership\
    [relation project $Ownership Acquired] o\
    YearlyTotal int {[relation cardinality $o]}]
+========+-----------+
|Acquired|YearlyTotal|
|string  |int        |
+========+-----------+
|2001    |3          |
|2000    |2          |
|2002    |1          |
|2003    |1          |
+========+-----------+
```

We can see that the projection of Ownership.Acquired was extended by an attribute whose value was calculated by taking the cardinality of another relation held in the "o" variable. The "o" relation is constructed by finding those tuples in Ownership that match the value for each distinct value in the projection of the Acquired attribute.

This same operation works even when the projection of the original relation value has no attributes at all (the nullary projection). By summarizing over the "DEE" relation, we can obtain a relation with a single attribute and a single tuple to yield a single value that is a function of the entire relation.

```
proc ravg {rel attr} {
    set sum 0
    foreach val [relation list $rel $attr] {
        incr sum $val
    }
    return [expr {$sum / [relation cardinality $rel]}]
}
% set DEE {Relation {} {{}} {{}}}
Relation {} {{}} {{}}
% relformat [relation summarize $Dog $DEE o\
    OverallAvgAge int {[ravg $o Age]}]
+=============+
|OverallAvgAge|
|int          |
+=============+
|3            |
+=============+
```

The "DEE" relation is that relation value whose attributes are the empty set and whose body is the single tuple that is also the empty set[3]. The other relation in this pair, "DUM", has an empty body. These two relations play important roles as operator identities and operator annihilators. Clearly, computations other than just taking the cardinality are possible, such as taking an average. Indeed if you are using Tcl 8.5, the `::tcl::mathfunc` mechanism allows creating aggregate relation operations that map to the `expr` command syntax.

---

[3] Indeed the only tuple that such a relation could have.

3.3.5 Introspection

It is very much in keeping with the “Tcl way” to provide introspection commands. For relation values, these commands report the sizes and structure of the relation. The commonly useful commands are `isempty` and `isnotempty`.

cardinality The `cardinality` of a relation is the number of tuples in its body.

isempty The `isempty` command is shorthand for `expr {[relation cardinalty $r] == 0}`.

isnotempty The `isnotempty` command is shorthand for `expr {[relation cardinalty $r] != 0}`.

degree The `degree` of a relation is the number of attributes in its heading.

heading The `heading` command returns a three element list that is the first three parts of the string representation of a relation.

attributes The `attributes` command returns a list of attributes for the argument relation.

identifiers The `identifiers` command returns the list of identifiers for the given relation. Each identifier, in turn, is a list of attributes.

```
% relation cardinality $Dog
6
% relation isempty [relation emptyof $Dog]
1
% relation isnotempty $Dog
1
% relation degree $Dog
3
% relation heading $Dog
Relation {DogName string Breed string Age int} DogName
% relation attributes $Dog
DogName Breed Age
% relation identifiers $Dog
DogName
```

3.3.6 Interface to Other Tcl Data Types

There are a number of cases where the particular structure of a relation value matches that of an existing Tcl data type. In these cases, TclRAL provides commands to conveniently move data from relation values into other Tcl data types. The goal here is to make it easier to interface relation data with other, presumably pre-existing Tcl commands.

First we recognize that a Tuple value can represent a Relation value of cardinality one without any loss of attribute values. So we provide a means to extract the tuple from a relation that is of cardinality one. The `relation tuple` command will convert a relation value that contains a single tuple into a tuple value. It is an error to invoke `relation tuple` with a relation value of cardinality that is not one. With the tuple value, all the tuple commands discussed above are available. Most useful are `tuple assign` to assign attributes into Tcl variables and `tuple extract` to obtain one or more attribute values from a tuple. Indeed the need to move attribute values into Tcl variables or values is common enough to warrant the `relation assign` and `relation extract`. These commands are short hand commands for invoking the corresponding tuple commands on the return value of `relation tuple`. Consider:

```
set mike [relation choose $Owner OwnerName Mike]
% relation cardinality $mike
1
% relation assign $mike
2
% puts $OwnerName
Mike
% puts $Age
50
% puts [relation extract $mike OwnerName]
Mike
% puts [relation extract $mike OwnerName Age]
Mike 50
```

Both arrays and dictionaries[4] provide a simple one-to-one mapping of a key to a value. In relation terms, this implies that, for relation values that have a single identifier and that identifier consists of a single attribute and where there is only one other attribute in the relation, the tuples in the body of the relation can be stored in a Tcl array or dictionary without any loss of attribute values. Although this is a very specialized condition for a relation value, it is not an uncommon one and having commands to move data to other types of Tcl values is particularly convenient.

In our example, the Owner relvar is a simple mapping of the owner's name to the owner's age and therefore matches the form to be transformed into a dictionary or an array.

```
% dict get [relation dict $Owner] Mike
50
% relation array $Owner ownerarray
% parray ownerarray
ownerarray(Alice)  = 30
ownerarray(George) = 35
ownerarray(Jim)    = 42
ownerarray(Mike)   = 50
ownerarray(Sue)    = 24
```

---

[4] TclRAL must be build against Tcl 8.5 in order to obtain support for dictionary types.

The projection of a relation value can also be used to create such simple mappings, as in:

```
% relation array [relation project $Dog DogName Breed] dogbreed
% parray dogbreed
dogbreed(Fido)   = Poodle
dogbreed(Fred)   = Spaniel
dogbreed(Jumper) = Mutt
dogbreed(Rover)  = Retriever
dogbreed(Sam)    = Collie
dogbreed(Spot)   = Terrier
```

Lists are a very important and useful data type in Tcl. The `relation list` command extracts the value of an attribute from all the tuples in a relation. If that relation happens to be of degree one or if the attribute forms an identifier, then the resulting list is also a set and that set may be used with the `set` package from TCLLIB.

```
% relation list $Dog DogName
Fido Sam Spot Rover Fred Jumper
```

TclRAL also provides commands to interface to the `matrix` package in TCLLIB. A matrix can hold any relation value without loss of any attribute values. The `relformat` command that we have already encountered, is actually implemented by converting a relation value into a `matrix` and then using the `report` package to generate text. This approach gives considerable flexibility for controlling the form of the output.

3.3.7 Miscellaneous Operations

There are always a few items in any categorization that seem to defy the pattern. In this section we discuss some of the more unusual operations in TclRAL. The commands in this section represent operations that are either clumsy to compute using the other relation operators or for which there is a significant implementation advantage when doing the computation internally.

tclose The `tclose` command computes the transitive closure of its relation value argument. The transitive closure operation is useful when dealing with hierarchically structured data. One familiar example is that of dealing with include file dependencies in "C" language source code. If some "C" source file includes a file which in turn includes other files, the original source needs to be rebuilt when any of the included files are modified. This implies that there is a graph of dependencies and we need to know if there is any path from a "C" source file to some included file regardless of how many intervening files there might be. The `tclose` command can compute this information. It operates on binary relation values and has a tuple in the result if there exists a path between any nodes of the implied graph. Consider the following relation value that defines which files immediately include other files in a set of "C" source files. This data can be gleaned from the files by examining each "C" source file in isolation. If we are then interested in finding the list of files whose modification dates must be examined to determine if `a.c` needs to be rebuilt we could proceed as follows.

```
set includes {
    Relation
    {Src string Inc string}
    {{Src Inc}}
    {
        {Src a.c Inc a.h}
        {Src a.c Inc b.h}
        {Src a.h Inc aa.h}
        {Src aa.h Inc stdio.h}
        {Src b.h Inc stdio.h}
    }
}
% relformat [set tclosure [relation tclose $includes]]
+======+=======+
|Src   |Inc    |
|string|string |
+======+=======+
|a.c   |a.h    |
|a.c   |b.h    |
|a.c   |aa.h   |
|a.c   |stdio.h|
|a.h   |aa.h   |
|a.h   |stdio.h|
|b.h   |stdio.h|
|aa.h  |stdio.h|
+======+=======+
% relation list [relation project\
    [relation restrictwith $tclosure {$Src eq "a.c"}]\
    Inc]
a.h b.h aa.h stdio.h
```

rank The `rank` command adds a new integer attribute that is the relative ranking of another attribute. The ranking is determined conceptually by sorting the relation value on an given attribute and determining, for each tuple, how many of the tuples in the relation have a value of the attribute that is less than or equal to that of the given tuple. This command is particularly convenient when trying to determine some ordinal relationship among the values of an attribute. For example, suppose we wish to know the two youngest dogs. We could proceed as in:

```
% relformat [set rankedDogs [relation rank $Dog Age AgeRank]]
+=======+---------+---+-------+
|DogName|Breed    |Age|AgeRank|
|string |string   |int|int    |
+=======+---------+---+-------+
|Fido   |Poodle   |2  |2      |
|Sam    |Collie   |4  |4      |
|Spot   |Terrier  |1  |1      |
|Rover  |Retriever|5  |5      |
|Fred   |Spaniel  |7  |6      |
|Jumper |Mutt     |3  |3      |
+=======+---------+---+-------+
% relformat [relation project\
    [relation restrictwith $rankedDogs {$AgeRank <= 2}]\
    DogName]
+=======+
|DogName|
|string |
+=======+
|Fido   |
|Spot   |
+=======+
```

tag The `tag` command creates a new identifier that is an incrementing integer value based on the ordering of an attribute. It is often useful to tag a relation when you wish to project away an identifying attribute and avoid losing any potential duplicates. It is also possible to create new identifiers from an existing identifier by adding an ordering attribute. For example, the following expression creates a new identifier {OwnerName AcqTag} which shows the order that a dog was acquired by a particular Owner.

```
% relformat [relation tag $Ownership -ascending Acquired\
    -within OwnerName AcqTag]
+========+=======+--------+======+
|OwnerName|DogName|Acquired|AcqTag|
|string   |string |string  |int   |
+=========+=======+--------+======+
|Sue      |Sam    |2000    |0     |
|George   |Sam    |2000    |0     |
|George   |Fido   |2001    |1     |
|Alice    |Spot   |2001    |0     |
|Sue      |Fido   |2001    |1     |
|Mike     |Rover  |2002    |0     |
|Jim      |Fred   |2003    |0     |
+=========+=======+--------+======+
```

group The `group` command produces a new relation by combining several attributes into a relation valued attribute. This gives a form of nesting.

```
% relformat [relation group $Ownership\
    DogAcquisition DogName Acquired]
+=========+------------------+
|OwnerName|DogAcquisition    |
|string   |Relation          |
+=========+------------------+
|Sue      ||+=======+--------+|
|         ||DogName|Acquired||
|         ||string |string  ||
|         ||+=======+--------+|
|         ||Fido   |2001    ||
|         ||Sam    |2000    ||
|         ||+=======+--------+|
|George   ||+=======+--------+|
|         ||DogName|Acquired||
|         ||string |string  ||
|         ||+=======+--------+|
|         ||Fido   |2001    ||
|         ||Sam    |2000    ||
|         ||+=======+--------+|
|Alice    ||+=======+--------+|
|         ||DogName|Acquired||
|         ||string |string  ||
|         ||+=======+--------+|
|         ||Spot   |2001    ||
|         ||+=======+--------+|
|Mike     ||+=======+--------+|
|         ||DogName|Acquired||
|         ||string |string  ||
|         ||+=======+--------+|
|         ||Rover  |2002    ||
|         ||+=======+--------+|
|Jim      ||+=======+--------+|
|         ||DogName|Acquired||
|         ||string |string  ||
|         ||+=======+--------+|
|         ||Fred   |2003    ||
|         ||+=======+--------+|
+=========+------------------+
```

ungroup The `ungroup` command inverts the operation of the group command, *i.e.* a relation valued attribute is “flattened” into a set of scalar attributes.

tuple For relations that contain a single tuple, the `tuple` command converts the Relation type value to a Tuple type value.

iteration There are times when it is convenient to iterate across the tuples of a relation.

Iteration is not needed as often as one might first expect since most relation operators function across the entire set. Indeed one of the more powerful aspects of the relational approach is the ability to do set-wise operations obviating the need to write iteration loops repeatedly. However, when generating output and under some other circumstances, examining a relation on a tuple by tuple basis and in some particular tuple order is necessary. For this, TclRAL provides the `foreach` command. By analogy with the `foreach` command in the Tcl core, `relation foreach` makes each tuple of a relation available as a relation of cardinality one. It also allows for visiting the tuples in the relation in a particular order.

```
% relation foreach d $Dog -descending DogName {
      tuple assign [relation tuple $d]
      puts $DogName
  }
Spot
Sam
Rover
Jumper
Fred
Fido
```

rename For operations that create relation values of a different type from their arguments, the possibility exists that the result would have duplicated attribute names if the arguments had at least on common attribute name. The `rename` command allows attributes names to be changed and provides a means to avoid the errors that would otherwise arise form duplicated attribute names.

3.4 Relvar Operations

TclRAL defines a separate variable space specifically to store relation values. This may seem unusual since as we have already seen, ordinary Tcl variables readily hold relation values. In a database situation, relvars hold those relation values that persist in the database. Although TclRAL does not provide any transparent persistence, relvars do serve to strictly partition those operations that modify relation values in place from those that return new relation values (*i.e.* `::ral::relvar` commands *vs.* `::ral::relation` commands). Also, relvars do serve as the basis of the load/store persistence that is provided by TclRAL. Another important use of relvars is for defining integrity constraints. Integrity constraints are discussed in more detail below.

The relvar variable names follow the familiar namespace resolved names like ordinary Tcl variables. Indeed each time a relvar is created, a corresponding Tcl variable of the same name is also created. Thus a relvar named `::myspace::Dog` would have a Tcl variable also named `::myspace::Dog`. There are a few implications of this. In order to create the relvar, `::myspace::Dog,` the namespace, `::myspace,` must already exist, as would be the case for creating an ordinary Tcl variable. TclRAL places a trace on the Tcl variable to prevent it from being written. All writes to relvars should come via TclRAL and preventing writes is meant to help accidental coding errors since determined scripts can overcome the trace placed by TclRAL. However, the existence of the Tcl variable that holds the same relation value as the relvar is very convenient for programming since the values held in relvars are often the starting terms for a relational expression. The relvar variable space is resolved in the same way as for Tcl variable name resolution. Relvars created without fully resolved names (*i.e.* without a leading "::") are placed in the current namespace. Relative relvar references (*i.e.* without a leading "::") are resolved first in the current namespace and then in the global namespace. These rules also apply to the names given to integrity constraints. This gives us a means of holding a set of relvars and their constraints together in a namespace-like arrangement where naming conflicts can be more easily avoided.

The operations provided for relvars are given in the next sections. Again we have grouped the operations into logically convenient sections.

3.4.1 Assignment

A fundamental relvar operation is to assign the relvar a value. It is important to remember that you may not change the type of a relvar by assignment. Only relation values that are of the same type as the relvar may be assigned to the relvar.

creation A relvar is created using the `relvar create` command. This establishes the type of relation value that may be assigned to the relvar. The created relvar has an empty body.

set By analogy to the Tcl core command, `set`, `relvar set` assigns a new value to a relvar if one is supplied and returns the current value in any case.

unset Relvars may be deleted by `unset`ting them. Again, this command follows the pattern of the core Tcl `unset` command. Note however, that you may not unset relvars that have constraints defined upon them. Those constraints must be deleted first.

3.4.2 Insertion

A relvar may be populated by inserting tuple values. The `relvar insert` command will insert zero or more tuple values into a relvar. Insertion semantics are such that an error is raised by any attempt to insert a duplicate tuple.

```
% relformat [relvar create MyDog [relation heading $Dog]]
+=======+------+---+
|DogName|Breed |Age|
|string |string|int|
+=======+------+---+
+=======+------+---+
% relformat [relvar insert MyDog\
    {Age 10 DogName Joan Breed {Afghan Hound}}\
    {Breed Dachshund Age 1 DogName Alfred}]
+=======+------------+---+
|DogName|Breed       |Age|
|string |string      |int|
+=======+------------+---+
|Joan   |Afghan Hound|10 |
|Alfred |Dachshund   |1  |
+=======+------------+---+
```

3.4.3 Deletion

Two forms of deletion are provided. One form is very general and the other is useful when a set of identifying attributes is known. In all cases the number of tuples deleted is returned.

delete In its first form, the `delete` command will delete those tuples from a relvar where an expression evaluates to true. The value of each tuple in the relvar is made available in a Tcl variable.

deleteone In many cases you know the value of a set of identifying attributes and examining all the tuples in a relation to find the one the delete is not necessary. For this case the `deleteone` command is useful.

```
% relvar delete MyDog d {[tuple extract $d Age] > 2}
1
% relformat $MyDog
+=======+---------+---+
|DogName|Breed    |Age|
|string |string   |int|
+=======+---------+---+
|Alfred |Dachshund|1  |
+=======+---------+---+
% relvar deleteone MyDog DogName Alfred
1
% relformat $MyDog
+=======+------+---+
|DogName|Breed |Age|
|string |string|int|
+=======+------+---+
+=======+------+---+
```

3.4.4 Update

Like `delete`, `update` also comes in two forms for the same reason. For `update`, an expression determines which tuples are to be updated. For `updateone`, the tuple to update is determined by supplying the attribute values for an identifier. In both cases a script is executed for each tuple to update. The script can modify the tuple to be updated and whatever the value is at the end of the script is updated into the relvar.

```
% relvar update MyDog d {[tuple extract $d Age] > 2} {
    tuple update d Age [expr {[tuple extract $d Age] + 1}]
  }
1
% relvar updateone MyDog d {DogName Alfred} {
    tuple update d Age [expr {[tuple extract $d Age] + 1}]
  }
1
% relformat $MyDog
+=======+------------+---+
|DogName|Breed       |Age|
|string |string      |int|
+=======+------------+---+
|Joan   |Afghan Hound|11 |
|Alfred |Dachshund   |2  |
+=======+------------+---+
```

3.4.5 Introspection

The only introspection command provided by relvars is to obtain a list of the names of all the relvars. The `relvar names` command returns a list of all the relvar names in fully resolved form. Less than all the names can be obtain by supplying an optional matching pattern argument.

3.5 Relvar Constraints

Up to this point we have been mainly concerned with the structure and operations that can be performed on relations. Another important aspect of managing data is integrity. We have seen some integrity constraints already. In this section we will discuss how TclRAL constrains the values of relations helping to insure correct program operation.

First, since each attribute has a declared data type, TclRAL insures that the value of each attribute can be interpreted as that data type. Indeed each tuple and relation attribute is converted to the declared type when the value is of the attribute is set and any attempt to set an attribute to a value that cannot be interpreted as the declared data type will cause an error. One important reason for this is to insure that calculation type operations on values (*e.g.* **SUMMARIZE**) will not result in errors because the data values are not of the proper arithmetic type. Of course, any attribute can be set to be the `string` type and all values are allowed. In this way, one can then choose to interpret the string in a variety of ways as is commonly done in Tcl.

Another important constraint is that all relations must have at least one identifier and duplicated tuples with respect to the identifiers of a relation are not allowed. Since relations are fundamentally sets and sets do not have duplicate members, constraining the body of a relation to have tuples that are unique with respect to the identifiers insures all the amazing properties that sets have. Most of the work to insure the set aspect of relations happens automatically in the various operators. For example, **PROJECT**ing a set of non-identifying attributes will result in a relation value that potentially can have tuples elided since they would otherwise be duplicates. This is sometimes disconcerting to programmers when they first begin working with relations but is essential to maintain the fundamental set notion of relations.

Usually, the set of relvars that represent the semantics of a particular problem have natural relationships that arise from the semantics of the problem domain. In our running example, a natural relationship exists between Dogs and Owners. In the relational view, these relationships are implemented by having attributes in one relvar refer to attributes in the related relvar. Reference in this case implies that the values of attributes in the referring relvar have the same values as corresponding attributes in the referred to relvar. The is the classic "foreign key" reference that is common in database management systems.

Constraints in general are a means of insuring that the values of the relvars in a program move from one valid state to another valid state. There are many ways that constraints can be expressed but broadly we are interested in either *procedural constraints* or *declarative constraints*. Procedural constraints are those that are determined by evaluating a relational expression. The system knows little if anything about the contents of the expression associated with a procedural constraint. It only knows to evaluate the expression and the constraint is satisfied if the expression evaluates to true. Declarative constraints are those that are determined by the system based on a set of assertions about the relationships between the relvars. TclRAL is concerned with declarative constraints. This is not to say that procedural constraints are unimportant and indeed TclRAL may support them in the future.

The declarative constraints supported by TclRAL are concerned with defining the referential integrity among a set of relvars. Some relvars contain attributes whose values match those of attributes in another relvar. It is convenient to talk about a *referring* relvar and a *referred to* relvar. In addition to the existence of some reference we are also concerned with the multiplicity and conditionality of the reference. The notion of multiplicity determines whether or not a given tuple in a relvar may be referred to more than one time. The notion of conditionality determines whether or not a given tuple is referred to at all. With the ideas of reference, multiplicity and conditionality, we can build a very powerful set of declarations that can be used to insure the integrity of the data in the relvars without having to explicitly program the checks.

3.5.1 Association Constraints

Association constraints declare that a set of attributes in a referring relvar have the same values as the attributes which constitute an identifier in a referred to relvar. In our running example, R2 is an association constraint. It is declared as:

```
relvar association R2\
    Contact OwnerName + Owner OwnerName 1
```

This command declares an association type constraint named, `R2`, that exists between the `Contact` relvar and the `Owner` relvar. `Contact` is the referring relvar and `Owner` is the referred to relvar. One or more tuples of `Contact` (as indicated by the + argument) have values of `Contact.OwnerName` that are the same as some value of `Owner.OwnerName`. Also every tuple of `Owner` (as indicated by the 1 argument) has a value of `Owner.OwnerName` that is equal to the value of `Contact.OwnerName` in exactly one tuple of `Contact`. Association constraints read left to right from referring relvar to referred to relvar. In general, the referring attributes can be a list of attributes, with the number of referring attributes being equal to the number of referred to attributes with the reference being between the corresponding attributes in the two lists. However the referred to attributes must constitute an identifier of the referred to relvar (`Owner` in this case). The multiplicity and conditionality are represented given by the characters below, which are intended to be reminiscent of their usage in regular expression syntax[5]:

+ ==> one or more (at least one)

***** ==> zero or more (no constraint at all)

1 ==> exactly one

? ==> zero or one (at most one)

For referring relvars (*e.g.* `Contact`), the multiplicity and conditionality can be any of the above four cases as dictated by the problem particulars. For the referred to relvars (*e.g.* `Owner`), the multiplicity and conditionality must be one of either "1" or "?".

The association type of constraint is very similar to the classic foreign key reference constraint from database systems. In TclRAL it is a bit more expressive in terms of being able to declare both the referring relvar and referred to relvar multiplicity and conditionality.

3.5.2 Partition Constraints

Partition constraints declare a complete, disjoint set partitioning among a set of relvars. In our running example R3 is a partition constraint. It is declared as:

```
relvar partition R3\
    Contact {OwnerName ContactOrder}\
    PhoneNumber {OwnerName ContactOrder}\
    EmailAddress {OwnerName ContactOrder}
```

This constraint declares that `Contact` is a super-set of `PhoneNumber` and `EmailAddress`. Conversely, `PhoneNumber` and `EmailAddress` are sub-sets of `Contact`. Every tuple of a subset relvar refers to exactly one tuple of its super-set relvar and every tuple in the super-set is referred to by exactly one tuple from exactly one of the subset relvars. The referred to attributes in the super-set must constitute an identifier (as is the

---

[5] My thanks to my colleague, Paul Higham, for letting me blatantly steal this idea from him.

case for all sets of referred to attributes). The singularity of the partition means that the referring attributes in the subset relvars may also be used as identifiers, however TclRAL does not require that the subset referring attributes be declared as an identifier (although they are in the example). Although it is possible to have only one subset relvar in the constraint, in practice there is little utility in partitioning a set into a single and therefore necessarily improper subset and such a partitioning is equivalent to a one to one association constraint.

The partition constraint may appear on first glance to be the same as a set of "1 -> ?" type association constraints between the sub-type relvars and the super-type relvar. However, such an arrangement would allow for the possibility of a tuple in the super-type relvar to be unreferenced by any tuples from the sub-type relvars. It is this situation that the partition constraint specifically covers.

It is convenient to consider a partition constraint as enforcing the "is a" concept in the sense of:

> `Contact` is a `PhoneNumber` or an `EmailAddress`

One consequence of the partition constraint is that the semijoins of all the sub-type relvars to the super-type relvar have no intersection. However, it is important not confuse the fundamental set partition idea behind a partition constraint with any concept of inheritance, particularly inheritance based on inclusion polymorphism that is found in many object oriented programming languages. This confusion is easily compounded by the fact that it is common design practice to place attributes that are common among the sub-type relvars as attributes of the super-type relvar since the singularity and unconditionality of the references make keeping track of common attributes in the super-type relvar much more convenient. In TclRAL, there is no concept of inheritance at all[6]. What is true is that every tuple in `Contact` is referred to by exactly one tuple in either `PhoneNumber` or `EmailAddress`.

3.5.3 Correlation Constraints

Association constraints can describe the integrity constraints when relvars are related in a one-to-one fashion or a one-to-many fashion. However, two other situations commonly arise. Some relvars are associated in a many-to-many fashion and sometimes it is necessary to store information about the association itself. In these cases, simply adding referring attributes to one relvar is not sufficient and a third relvar is required to hold the required number of referring attributes or the information about the association itself. Correlation constraints cover these situations. In our example, the `Ownership` relvar is a correlation relvar that exhibits both of the properties described above. It correlates the many-to-many association between `Dog` and `Owner` and the attribute `Ownership.Acquired` is the year that an `Owner` became the owner of a `Dog`. It is descriptive of the ownership relationship itself and not of either the `Owner` or the `Dog`.

Correlation constraints declare referential associations between two relvars that are mediated by a third relvar, the correlating relvar. The correlating relvar contains refer-

---

[6] That is not to say inheritance is a bad thing or that TclRAL doesn't need some concept of inheritance. It is just not there currently.

ential attributes that refer to the two relvars that participate in the relationship. In our running example, R1 is a correlation constraint. It is declared as:

```
relvar constraint R1 Ownership\
    OwnerName + Owner OwnerName\
    DogName * Dog DogName
```

This constraint declares that `R1` is a correlation constraint between `Owner` and `Dog` with the `Ownership` relvar mediating the correlation. For every tuple in `Ownership`, `Ownership.OwnerName` refers to some tuple in `Owner` and `Ownership.DogName` refers to some tuple in `Dog`. Further, every tuple in `Owner` is referred to one or more times by the tuples in `Ownership`. For `Dog`, every tuple is referred to zero or more times by some tuple in `Ownership`. Thus, the references by `Ownership` are unconditional and for this example, the multiplicity and conditionality of the constraint insures that every `Owner` owns one or more `Dog`s, but that `Dog`s may exist that are not owned by any `Owner`.

It is required that the attributes referred to by the correlation relvar constitute an identifier for the participating relvar. This implies that the union of the attribute sets that form the two references in the correlation relvar will also server to identify an tuple in the correlation relvar. However, TclRAL does not insist that the correlation relvar have an identifier that is the union of the two sets of referring attributes (although the example does so).

Sometimes a correlation constraint needs to be even more confining in the sense that every tuple of each of the participating relvars must be correlated. A matrix, for example, is a correlation of every row and every column such that the correlation represents the Cartesian product of the rows and columns. For this case, an optional *-complete* argument may be given and this will insure that the number of tuples in the correlation relvar equals the product of the number of tuples in the two related relvars.

Correlation constraints appear as if they can be decomposed into two association constraints, as in:

```
relvar association R1-Owner\
    Ownership OwnerName + Owner OwnerName 1
relvar association R1-Dog
    Ownership DogName * Dog DogName 1
```

Indeed, replacing `R1` by `R1-Owner` and `R1-Dog` will result in data values of the relvars being constrained in the same way. However, TclRAL distinguishes association constraints from correlation constraints because:

- Semantically there is only one relationship and that is between Owner and Dog.
- There is redundant information in the two ASSOCIATION definitions in that the multiplicity of the referred to relvar is always 1.
- Two distinct association constraints cannot be used to declare a *complete* correlation.

3.5.4 Constraint Evaluation

It is important, during the execution of a program, that modifications to the values of the relvars transition from one consistent state to another consistent state. However, frequently it requires more than one operation to make the constraints between relvars consistent. For example, if two relvars that are associated on a one-to-one unconditional basis (***e.g.*** `relvar association X1 A B_ID 1 B B_ID 1`), then any insertion or deletion from one relvar must have a corresponding insertion or deletion in the other. So we must have some concept of when it is appropriate to evaluate the relvar constraints.

TclRAL evaluates constraints at the end of a `relvar eval` command or, if execution is outside of a `relvar eval` command, then at the end of any relvar command that modifies a relvar. For example:

```
% relvar eval {
    relvar insert Owner {OwnerName Tom Age 22}
    relvar insert Ownership {OwnerName Tom DogName Jumper Acquired 2006}
    relvar insert Contact {OwnerName Tom ContactOrder 1}
    relvar insert PhoneNumber {OwnerName Tom ContactOrder 1 AreaCode 808\
        Number 555-2357}
}
```

executes as a transaction that leaves the relvars consistent and:

```
% relvar insert Owner {OwnerName Tom Age 22}
for correlation ::R1(::Owner <== [+] ::Ownership [*] ==> ::Dog), in relvar ::Owner
tuple {OwnerName Tom Age 22} is not referenced by any tuple
for association ::R2(::Contact [+] ==> [1] ::Owner), in relvar ::Owner
tuple {OwnerName Tom Age 22} is not referenced by any tuple
```

is executed outside of a transaction and leaves the `Ownership` and `Contact` relvars in an inconsistent state. Anytime a constraint evaluation fails, the values of the relvars are rolled back to their previous state. The `relvar eval` command may be nested to an arbitrary depth.

3.5.5 Constraint Introspection

In the previous sections we saw the commands needed to create constraints on relvars. Here we examine the other constraint commands. All other constraint manipulation is provided by the `::ral::relvar constraint` option and the additional suboptions.

Constraints may be deleted using `relvar constraint delete`. It is important to remember that all constraints that refer to a relvar must be deleted before the relvar itself may be deleted (*i.e.*, only unconstrained relvars may be deleted). The constraints that a particular relvar participates in is available from `constraint member`. Introspection into the constraints is provided by `constraint name` and `constraint info`. These commands give the names of the constraints and details of their definitions.

```
% lsort [relvar constraint names]
```

```
::R1 ::R2 ::R3
% relvar constraint info R1
correlation ::R1 ::Ownership OwnerName + ::Owner OwnerName
DogName * ::Dog DogName
% relvar constraint info R3
partition ::R3 ::Contact {OwnerName ContactOrder}
::PhoneNumber {OwnerName ContactOrder} ::EmailAddress
{OwnerName ContactOrder}
```

3.6 Poor Man's Persistence

TclRAL treats relation structured information in the same way that Tcl treats any data stored in variables. There is no transparent persistence supported by the language. Many different ways have grown up over time to provide data persistence to programs that need to have information preserved between invocations. For some of the more complex data structures in TCLLIB the packages provide the ability to serialize and deserialize the data structure so it can be stored and retrieved easily. TclRAL provides three mechanisms to help saving relvar data between program invocations. Note here that we are concerned with saving the state of the relvars and their constraints as the significant program data that must persist. Since relation values have a string representation, they are easily stored and retrieved by conventional means.

- The `serialize` and `deserialize` commands produce a string that contains all the information to save and restore the state of a set of relvars. Clearly, the serialized string may be very large.
- The `storeToMk` and `loadFromMk` commands will save and load the relvars in a Metakit database. Metakit is well suited to this storage not only because of its efficiency but also because a Metakit view may be nested, easily handling relation valued attributes.
- The `dump` command produces a Tcl script that when evaluated will reproduce the relvar state. Storing data as the Tcl script that will reconstruct the data is an old trick. It has some distinct disadvantages, but is very convenient when small changes need to be made with your favorite text editor or when the dump is made part of a Tcl package that, for example, initializes an empty schema to be used by the package.

All of these mechanisms are unsatisfactory in the sense that they require a program to explicitly manage the loading and storing of the relvar data and in that all the relvar data must be able to be resident in memory at the same time. This ultimately limits the scale of program to which TclRAL may be applied, but for a large class of applications these limitations are not onerous and several million bytes of program data are easily handled.

4 Other Approaches

There are many approaches to dealing with relation structured data. These have been well summarized elsewhere[6]. We are going to choose only a couple of examples to consider. There are also many other approaches to dealing with program data that are quite fashionable (*e.g.* XML) that we will not consider here since they are not relationally oriented. One way of comparing the TclRAL approach to others is simply to divide the world into the SQL and non-SQL categories. For SQL based approaches there is a vast industry that supports the language and its implementations. Tcl is blessed with bindings to all the popular DBMS and so we will restrict our attentions here to SQLite `http://www.sqlite.org`, which is of more particular interest to Tcl'ers. In the non-SQL category, Ratcl `http://www.vlerq.org/vlerq/ratcl` is very close in spirit to TclRAL.

I will not devote much space here to SQL based approaches as these have been discussed at length in so very many places. SQLite is a phenomenal program that is so useful in so many contexts as is evidenced by is enormous popularity. However, SQLite is fundamentally about programming in SQL, despite the significant help that the Tcl bindings to SQLite give to Tcl programmers. SQL has many pros and cons, but the essential difficulty for a Tcl programmer (and most other programmers for that matter) is that one has to know two different languages and divide the program requirements between two distinct syntactic environments. When I program SQL from Tcl, which I do on many occasions, I am inevitably frustrated by the syntax considerations of SQL. I like to program in Tcl and I don't want to have to know two languages when I am already programming in my favorite one. This need to move between two different programming environments is sometimes called an *impedance mismatch*. TclRAL has as one of its primary goals the minimization of the mismatch between Tcl language constructs and relational data model concepts.

Ratcl is part of a larger project known as Vlerq. Vlerq is a very interesting and ambitious project that clearly intends to provide transparent, persistent, relationally structured data to programs (among many other things). Ratcl provides many relational operators, however there are some distinct design differences, namely:

- Ratcl is not purely relational. It does allow duplicate rows in a *view*. It also supports the notion of a row index and column index where values can be accessed based on positional knowledge.
- Ratcl views do not have an external string representation. This makes it difficult to hand views across interpreter boundaries. There are some special rules used to manage the views when the Tcl variable holding the view goes out of scope. This leads to some difficulties in situations where shimmering occurs.
- Expression evaluation within Ratcl operations has minor differences from the Tcl core `expr` command and requires special definitions to supply additional functions.
- Ratcl is truly a Tcl binding to the relational data structuring of Vlerq and that binding ultimately limits the degree to which it may be tightly integrated into the internal Tcl type system.

Those design differences in aside, Ratcl does provide virtually transparent persistence with a very efficient back-end storage mechanism making it one of the more remarkable approaches to non-SQL based relationally structured data, especially for Tcl programmers.

At this time it appears that Tcl is blessed with several different approaches to relationally structured data. This is analogous to the many different approaches to object oriented programming that Tcl has. I see this as positive in the same way that several different object oriented extensions have given the Tcl community the opportunity to determine what works best under the conditions of actual practice. It is my hope that the same distillation would take place with the various approaches to relationally structured data.

5 Future Work

As of this writing, TclRAL is available as version 0.8. This version is a complete rewrite and refactoring of the previous version. It is hoped that this has yielded a more easily maintained code base. This version also introduces a more consistent command interface. However, much remains to be done.

Relvar tracing Variable tracing is an important mechanism in Tcl. Being able to place traces on relvars is equally important.

Cascading update and delete Changing the values of referential attributes implies making many changes to keep referential integrity. Since the that information is known via the constraint definitions, TclRAL should be able to help make that task less tedious.

Default values Relvar definitions are an appropriate place to specify default values for attributes that may be used if an attribute value is otherwise missing. This would save programs from having to keep track of default values on their own. A related issue is that of system assigned identifiers.

Virtual relvars Version 0.7 of TclRAL had primitive support for virtual relvars (a.k.a views). This support was removed from version 0.8. It should return in a future version when a more complete implementation can be written to incorporate updating and constraint evaluation properly.

Procedural constraints Currently only the declarative type of constraints may be defined and enforced. It may be desirable to have some form of procedural constraint evaluation. Exactly how this would interact with relvar tracing is not clear.

Transparent persistence Transparent persistence would greatly increase the scale of problem to which TclRAL could be applied and is a desirable feature if unfortunately rather complicated to implement well.

Code improvements There is always a need for more code refactoring, testing and documentation.

Index

References

[1] Date, C.J., *An Introduction to Database Systems*, 8th ed., ISBN 0-321-19784-4.

[2] Date, C.J. and Hugh Darwin, *Databases, Types, and the Relational Model: The Third Manifesto*, ISBN 0-321-39942-0.

[3] Shlaer, Sally and Stephen J. Mellor, *Object Oriented Analysis: Modeling the World in Data*, ISBN 0-13-629023-X

[4] Shlaer, Sally and Stephen J. Mellor, *Object Oriented Analysis: Modeling the World in States*, ISBN 0-13-629940-7

[5] Mellor, Stephen J. and Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, ISBN 0-201-74804-5

[6] Wippler, Jean-Claude, *Relational algebra for Tcl: introducing Ratcl and Rasql*, http://www.equi4.com/docs/tcl2005e/ratcl.pdf