A Nested Relation Implementation for Semistructured Data

T. H. Merrett[*]
McGill University, Montreal, Canada

April 21, 2005

Abstract

A simple implementation of nested relations also supports recursively nested trees of indefinite depth, directed acyclic graphs for sharing common subexpressions, cyclic graphs for links and cross-references, and union types for attributes. This can all be built without going beyond the capabilities of an algebra of flat relations which includes relational recursion. Its advantage is to offer shortcuts to thinking about programs over the kind of data thus modelled, and to support, to this end, syntax for relational expressions which includes paths of nested attributes, regular expressions on these paths, and wildcards for generality and for partially known data schemas. This last is the essential aspect of "semistructured" data. Embedding such semistructured data within a matrix of ordinary text finally leads to new capabilities which include the analysis of text with or without implicit schemas induced by text markup.

Keywords nested relations path expressions semistructured data

Contents

1 Introduction

Serge Abiteboul, Peter Buneman, and Dan Suciu provide [1] a fusion of the two major lines of work on semistructured data over the past decade. Three decades before, the late Ted Codd gave [9] the simple form of flat relations, which excel at representing data on secondary storage because they abstract it to the appropriate level of whole files. The structure of flat relations has a trivial syntax because it is simple. The work on semistructured data has focused on data descriptions which generalize Codd's work, and on related query formalisms.

The relational algebra, which Codd also provided, abstracts over looping, and so is the basis for a very high-level programming language. This goes beyond query languages such as SQL because a query language is specialized and notoriously must be extended to provide, say, logical inference, or calculations on spatial or temporal data. Since semistructured data is frequently an aspect of scientific databases (e.g., genome databases or bibliographies), it seems important to include it in a programming language for secondary storage which can also handle such calculations.

We show that a formalism (with the required general programming capabilities, although we do not dwell on them in this paper) based on flat relations can indeed cope with all the subtleties of semistructured data and so include this new flexibility into a language which already deals with expert system rules, geospatial data, numerical calculations, and the like.

The rest of this introductory section reviews the relational algebra and the complementary domain algebra in the context of a modified syntax which is more suited to general programming than the apparently similar query-language syntax. Going beyond the limitations, which soon appear, of query language, we can introduce nested relations through a simple idea which needs only the most minor syntactic support.

In the sections following, we introduce syntactic sugar for referring to nested relations, and show a couple of useful representations, one of which is the flat-relation implementation of nesting (section 2). In section 3, we expand the syntax to recursive nesting, which has the same implementation representation. Crossreferencing by links, which also support data sharing, is explained in section 4, and then union types in section 5 complete the formalism for semistructured data. The syntax explained in the first five sections is sugar and adds nothing to the capabilities of the flat relational algebra given in the Introduction, but it offers more powerful ways to think about certain problems. However, a cautionary example shows that the new thinking is not always better than the old, and in section 7 we move on to the analysis of text with semistructured data embedded in it.

1.1 Relational Algebra

While we can assume that the algebra of flat relations is familiar to the reader, the syntax we need for it may not be. We restrict our attention to three well-known operators—a combination of selection and projection called the T-selector, natural join (**join**), and set union (**union**)—, three less common operators used for comparison of relations—natural composition **comp**, relational equality, = , and relational division, **sup**—, and a new relational operator called **grep**. We also give a unified syntax for updates.

1.1.1 T-selectors

The T-selector places the relational operand at the extreme right, for ease of expression-building, followed (to the left) by the selection condition, in turn followed (now on the extreme left) by the projection list. On the relation (describing the bill of materials to manufacture an electric outlet)

BoM

(*assembly*	*qty*	*subassembly*)
`wallplug`	1	`cover`
`wallplug`	1	`fixture`
`cover`	1	`plate`
`cover`	2	`screw`
`fixture`	2	`plug`
`fixture`	2	`screw`
`plug`	2	`connector`
`plug`	1	`mould`

the T-selector to find the components of `plug`s and their quantities is

[*subassembly, qty*] **where** *assembly*=`"plug"` **in** *BoM*

Any number of attributes may appear in the projection list, and any condition that evaluates to **true** or **false** on each separate tuple of the operand may be used after **where**. ("T" in "T-selector" stands for "tuple".) The special cases of projection or selection are obtained by omitting the **where**-clause or the projection list, respectively.

Another important special case makes the T-selector a Boolean predicate:

[] **where** *assembly*=`"plug"` **in** *BoM*

gives a relation on no attributes ("nullary"). A nullary relation can evidently have only two states: empty (no tuples in the operand satisfy the selection condition), or not empty (some tuples are selected). The empty nullary is interpreted as **false** and the non-empty nullary as **true**. The above example is pronounced "something where assembly='plug' in BoM".

1.1.2 Joins

The natural join is written as an infix operator, **join**. This differs from the apparently more flexible SQL convention of writing the join conditions explicitly in syntax which is SQL T-selector syntax except that a second operand is introduced. The advantages of infix notation are that the join operator is readily seen for what it is (and this obliges the programmer to have a clear idea of what natural join means), and that expressions of several operations may be articulated. Since **join** defaults to Cartesian product if the operands share no attributes, combining this with a T-selector regains flexibility in case we need a join condition which does not test join attributes for equality.

The full form of natural join includes a specification of the attributes to be matched between the operand relations, still in infix form. Here is an example, using a copy of *BoM*, above, renamed *Bom*′(*assembly*′,*qty*′,*subassembly*′). It matches *subassembly* and *qty* from *BoM* with (respectively) *assembly*′ and *qty*′ from *Bom*′.

Bom [*subassembly,qty* **join** *assembly*′,*qty*′] *Bom*′

The result contains the union of the attribute sets of the two operands, but we notice that the attribute pairs *subassembly* and *assembly′*, and *qty* and *qty′* duplicate each other, and can be considered alias names for the same attributes in the result.

(*assembly*	*qty*	*subassembly*	*assembly′*	*qty′*	*subassembly′*)
wallplug	1	cover	cover	1	plate
fixture	2	plug	plug	2	connector

(This example is contrived for illustrative purposes rather than being particularly useful.)

The **union** operator is, generally, an outer join, extending set union to relations just as natural join extends set intersection to relations. In this paper we need only set union, so we just call it **union**. It is an infix operator between two relations.

While **join** and the general form of **union** can be seen as extensions to relations of set-valued operators on sets (intersection and union, respectively), we also need extensions to relations of truth-valued operators on sets, such as tests for non-empty intersection, containment, and equality. These are given, respectively, by natural composition [11] (**comp**), division (**sup**), and an equality test (=). We illustrate natural composition by finding grandparents given a relation *Parent*(*Sr, Jr*).

Parent[*Jr* **comp** *Sr*]*Parent*

The result has attributes *Sr* and *Jr*, like *Parent*, but *Sr* comes from the first operand and *Jr* from the second: the join attributes disappear in the result because they are used to specify the sets of values (associated with each value of the non-join operands) that are compared (for empty intersection, in the case of *comp*). The other two operators mentioned here have similar semantics, except that the sets of values specified by the join attributes are compared for containment (**sup**) or equality (=).

1.1.3 Grep

A new relational operator is suggested by the need we shall have in section 7 for textual pattern matching. We would like to be able to find a substring or a match for a more general regular expression in a relation without knowing in advance which attribute it is in. For this, we consider **grep** (which stands for "get regular expression pattern", as it does in Unix) to be a relational operator.

grep `"plug"` **in** *BoM*

returns, like selection, each whole tuple in which some attribute contains the (substring) pattern, `"plug"`:

(*assembly*	*qty*	*subassembly*)
wallplug	1	cover
wallplug	1	fixture
fixture	2	plug
plug	2	connector
plug	1	mould

Grep can be made more specific by allowing parameters. One parameter is of type **integer** and gives the position in the attribute where the match started. The second parameter is of type **attribute** and gives the attribute itself that matched the pattern. Neither, either, or both these parameters may be supplied to **grep**, and the type of the variable chosen for each tells **grep** what it is. Typically, we could use *pos* (**integer**) and *attr* (**attribute**):

grep(*attr,pos*) `"plug"` **in** *BoM*

(*assembly*	*qty*	*subassembly*	*attr*	*pos*	)
wallplug	1	cover	assembly	4	
wallplug	1	fixture	assembly	4	
fixture	2	plug	subassembly	0	
plug	2	connector	assembly	0	
plug	1	mould	assembly	0	

Note that this requires a metadata type **attribute** for attributes. We shall make further use of this in section 6.

Note that a pattern may be found several times in a tuple, in which case there will be more than one value of *attr* or *pos*. **Grep** produces additional tuples in this case, distinguished by the different values of *attr* or *pos*.

grep(*attr,pos*) `"l"` **in** *BoM*

(*assembly*	*qty*	*subassembly*	*attr*	*pos*)
`wallplug`	1	`cover`	`assembly`	2
`wallplug`	1	`cover`	`assembly`	3
`wallplug`	1	`cover`	`assembly`	5
`wallplug`	1	`fixture`	`assembly`	2
`wallplug`	1	`fixture`	`assembly`	3
`wallplug`	1	`fixture`	`assembly`	5
`fixture`	2	`plug`	`subassembly`	1
`plug`	2	`connector`	`assembly`	1
`plug`	1	`mould`	`assembly`	1
`plug`	1	`mould`	`subassembly`	3

Note that *pos* can be used to extract *proximity* of two occurrences of a pattern. If two successive values of *pos* are within a pre-specified distance, a proximity test may be considered satisfied. (To do this requires domain algebra which is not covered in section 1.2.) Note also that *pos* can be used to *rank* tuples or attributes according to how often a pattern occurs. (This requires equivalence reduction in the domain algebra of section 1.2.)

Of course, **grep** supports regular expressions, so that we can ask for attributes containing `er` or `re`

grep(*attr,pos*) `"er|re"`

or use the wildcard symbol, ., and the Kleene star, `*`, to cover gaps

grep(*attr,pos*) `"e.*r|r.*e"`

This will include `connector`, as well as `cover` and `fixture`
from the `er|re` example.

As well as these basic regular expression operators, **grep** supports derived or specialized operators. For example, finding `plug` or `Plug` can use

grep(*attr,pos*) `"[Pp]lug"` **in** *BoM*

or, for case insensitivity in general, we can use string operators ^ (or **uppercase**) or _ (or **lowercase**) to convert the operand to upper or lower case, respectively:

grep(*attr,pos*) `"plug"` **in** _*BoM* or
grep(*attr,pos*) `"plug"` **in lowercase** *BoM*

(This approach will, of course, give a *result* in upper or lower case.)

Note that **grep** will match a pattern if it occurs anywhere in the operand, so we are not obliged to say

grep(*attr,pos*) `".*plug.*"` **in** *BoM*

to find `plug` anywhere in the attributes of *BoM*. Indeed, this is not equivalent to our very first example, because it will return *pos*=0 for every tuple, since the pattern given actually starts at the beginning of each attribute that contains `plug`, even `wallplug`. To insist on exact match of an attribute to a pattern, we use the **grep** symbols indicating the beginning and ending of a string

grep(*attr,pos*) `"^plug$"` **in** *BoM*

This will not match `wallplug`. (Note that the **grep** symbol ^ has a quite different meaning from the uppercase string operator ^.)

Variables may be useful instead of the `.*` construct, so that the additional parts of the string can be returned. To provide this, we allow the programmer to define further attributes in the **grep** parameter list which can be used as wildcard variables in the pattern. For example,

grep(*attr,pos;x,y*) `"\x plug\y"` **in** *BoM*

gives

(*assembly*	*qty*	*subassembly*	*attr*	*pos*	*x*	*y*)
`wallplug`	`1`	`cover`	`assembly`	`4`	`wall`	
`wallplug`	`1`	`fixture`	`assembly`	`4`	`wall`	
`fixture`	`2`	`plug`	`subassembly`	`0`		
`plug`	`2`	`connector`	`assembly`	`0`		
`plug`	`1`	`mould`	`assembly`	`0`		

Note that the semicolon in the parameter list for **grep** separates the list of attributes that are recognized by type from the list of wildcard variables, which all have type **string** and are recognized by position. Note also that we use parentheses as delimiters in the pattern, unless whitespace happens to separate variable from constant parts of the pattern. This allows `x` and `y` to be part of the constant pattern.

As well as pattern constants, which we have shown above, **grep** can work with pattern variables, which could be attributes or top-level variables. For example, an equivalent to our first example (**grep** `"plug"` **in** *BoM*) would be

fix <− [*subassembly*] **where** *assembly*=`"fixture"` **in** *BoM*;
grep(*attr,pos*) *fix* **in** *BoM*

When the pattern variable is a relation of many tuples, each value is treated as an alternative—the patterns are **or**ed—and there will be at least one tuple in the result for each match.

1.1.4 Statements

An important supplement to the relational algebra are the operators that turn relational expressions into statements. These are the assignment and view operators. Assignment evaluates an expression and names the result.

Grandparent <− *Parent*[*Jr* **comp** *Sr*]*Parent*

creates the new relation, *Grandparent*, discussed earlier with attributes *Sr*, and *Jr*. View looks similar to assignment but names the *expression* without evaluating it: it can be thought of as a parameterless function.

GPview **is** *Parent*[*Jr* **comp** *Sr*]*Parent*

creates the view, *GPview*, with the same attributes as *Grandparent* but no data. If *GPview* is subsequently used in an assignment statement, for example, the evaluation is done then. The advantage of this postponement is that subsequent updates to *Parent* will be reflected in later references to *GPview*. This is not the case for *Grandparent*.

We can also have recursive views. These can be used to find transitive closure (for example, *Ancestor*(*Sr, Jr*) from *Parent*(*Sr, Jr*)) or in an inference engine to fire all rules starting with a given set of facts. To illustrate the simplest forms of these recursive views, we show the transitive closure that gives ancestor.

Ancestor **is** *Parent* **union** *Parent*[*Jr* **comp** *Sr*] *Ancestor*;

In the next section we show a more elaborate recursion than this on *BoM*, using **join** instead of **comp**.

1.1.5 Updates

Finally, **update** is an operator which, like assignment and view definition, gives a statement and has side-effects. (The relational algebra up to the discussion of assignment is purely functional, as is the domain algebra which follows.) This one operator has variants which accomplish all three forms of update, **add**, **delete**, and **change**. **Update** works always on subrelations, never at the level of individual tuples. We can use **union** to give the semantics of **add**:

update <*relation1*> **add** <*relation2*>;

is equivalent to

<*relation1*><− <*relation1*> **union** <*relation2*>;

except that the update can be done in place rather than by copying <*relation1*>, which most implementations of the latter would do.

Delete can similarly be defined in terms of a **difference** operator, which we have not given in this paper.

Change may be followed by statements which operate on attributes (so that full discussion depends on knowing the domain algebra, below) to alter them in place, and may have an optional **using** clause which specifies another relation whose **join** with the relation to be changed identifies the part of that operand relation whose tuples will be changed. (The conditional expression of the domain algebra gives an alternative way to select tuples to change.) The full exploitation of these two notions is very expressive and we illustrate only a simple variant. In *BoM*, replace `wallplug` by `outlet` wherever it appears in *assembly*[1], using the auxiliary relation

auxBoM(*assembly*	*newassembly*)
`wallplug`	`outlet`

update *BoM* **using** *auxBoM* **change** *assembly* <− *newassembly*;

We cannot update wildcard variables in **change** mode:

update *BoM* **change** *x* <− `"base"`
  using where $x \neq$ `""` **grep**(;*x*) `"\x plug"` **in** *BoM*;

does not change any attribute in *BoM* containing `plug` to `baseplug`, because *x* is not an attribute of *BoM* and so cannot be updated. Instead, we must identify the attribute and change that, which we do by extracting the value of the generated attribute *attr*. This introduces the metadata operator, **eval**, which is useful for all kinds of metadata.

update *BoM* **change eval** *attr* <− `"baseplug"`
  using grep(*attr*) `"^plug$"` **in** *BoM*;

does change any attribute in *BoM* containing `plug` to `baseplug`.

We may want to update part of the value found by **grep** instead of the whole of it. We could permit two more attributes before the semicolon in the **grep** parameter list, of types **type** and **any**, containing, respectively, the type and the value of the attribute that matched. Then we could change, say, `plug` to `socket` anywhere it is found by an update such as

update *BoM* **change eval** *attr* <− (*type*) (**substr**(**eval** *attr*,0,*pos*−1)
  cat `"socket"` **cat substr**(**eval** *attr*,*pos*+**len**(*val*)))
  using grep(*attr,pos,type,val*;) `"plug"` **in** *BoM*;

The **cat** operator concatenates strings. The final string resulting from the two concatenations must be cast to the *type* of the attribute being changed.

Update to the **grep** parameter *attr* is also possible; to *pos* is unnecessary; and to *type* is not always possible except indirectly, because of the independence of domains from relations. Attributes such as *attr, type* and *val* appear again with the **transpose** operator discussed at the end of section 1.3.

The **add** mode of **update** reveals the important consideration that relations may be *polymorphic*. For example (although this may not be good database design), we might want to add to *BoM* a relation giving the costs of the final components.

ComponentCost

(*assembly*	*cost*)
`plate`	`0.75`
`screw`	`0.05`
`connector`	`0.20`
`mould`	`1.35`

The result of

update *BoM* **add** *ComponentCost*;

could be written with null values (since the outer join, which the full form of **union** implements, can be considered to generate null values):

[1]For this paper, no update used as an example will persist; that is, the next time we see *BoM* in the paper, it will be the original *BoM*.

BoM

(*assembly*	*qty*	*subassembly*	*cost*)
`wallplug`	1	`cover`	$\mathcal{DC}$
`wallplug`	1	`fixture`	$\mathcal{DC}$
`cover`	1	`plate`	$\mathcal{DC}$
`cover`	2	`screw`	$\mathcal{DC}$
`fixture`	2	`plug`	$\mathcal{DC}$
`fixture`	2	`screw`	$\mathcal{DC}$
`plug`	2	`connector`	$\mathcal{DC}$
`plug`	1	`mould`	$\mathcal{DC}$
`plate`	$\mathcal{DC}$	$\mathcal{DC}$	`0.75`
`screw`	$\mathcal{DC}$	$\mathcal{DC}$	`0.05`
`connector`	$\mathcal{DC}$	$\mathcal{DC}$	`0.20`
`mould`	$\mathcal{DC}$	$\mathcal{DC}$	`1.35`

Here, we introduce the "don't care" null value, which behaves for all operations as if it isn't there: it is the identity of any scalar operation on it, and so on. An attribute containing only $\mathcal{DC}$ is the same as an attribute which isn't there. Conversely, any attribute which isn't in a relation can be thought of as being an attribute of the relation but containing only $\mathcal{DC}$. So the above relation could also be thought of (and stored) as two separate subrelations, one on attributes *assembly, qty* and *subassembly*, and the other on attributes *assembly* and *cost*.

A second variant of the **change** mode of **update** takes advantage of this polymorphism to allow inserting or removing of *attributes* (as opposed to adding or deleting tuples in the **add** and **delete** modes, respectively). We can remove an attribute from a whole relation (*BoM* as modified above)

update *BoM* **change remove** *cost*;

or from selected tuples

update *BoM* **change remove** *cost* **using where** *assembly*=`screw` **in** *BoM*;

The effects of these would be, respectively, to restore *BoM* to its original three attributes (but retaining the four extra tuples that came from *ComponentCost*), or to remove the *cost* only of the tuple of *ComponentCost* containing `screw` (i.e., setting its cost to $\mathcal{DC}$ in the table above).

Any attribute to be inserted into a relation, or part of a relation, must be virtual and defined by an expression of the domain algebra, discussed next. Example syntax is

update *BoM* **change insert** <some virtual attribute>;

or

update *BoM* **change insert** <some virtual attribute> **using where** *assembly*=`screw` **in** *BoM*;

and the meaning (after we have explained virtual attributes) is clear by comparison with the **remove** examples.

We may use both **insert** and **remove** in one **update**, to achieve replacements. Each may be followed by a list of attribute names. (Without ambiguity, but possibly with more confusion, the keywords **remove** and **insert** could be replaced by **delete** and **add**, or even **drop** and **add**, respectively.)

Note that for the flat relations we have discussed so far, these modes to **insert** and **remove** attributes are not very useful, since the same results can be had, albeit by copying data, by projections. For nested relations (section 1.3), on the other hand, they are more useful.

1.2 Domain Algebra

The relational algebra alone does not permit us to do the kind of calculations needed by a programming language. We need operations on attributes, constituting what is known as the "domain algebra" (although it should be called the "attribute algebra"). As the relational algebra abstracts over the components of the relations (so that the syntax above looks like conventional algebra on numbers, with unary and binary operators, except the operands can be thought of as whole files), so the domain algebra abstracts over the relations involved. This is just about the only subtlety of the domain algebra, which is otherwise straightforward, but it is a very important subtlety: being able to write operations on attributes without reference to the relations they may be part of greatly reduces the mental effort of programming. In consequence, all domain algebra statements, like relational algebra views, may be considered to define parameterless functions.

Because the domain algebra is divorced from the relations containing the attributes, the result of any domain algebra expression is a "virtual" attribute, and must at some point be "actualized" if it is ultimately to be part of a relation. Suppose we had joined BoM and BoM', above, on *subassembly* and *assembly*$'$ (instead of on *subassembly,qty* and *assembly*$'$,*qty*$'$)

Bom$''$ <− *Bom* [*subassembly* **join** *assembly*$'$] *Bom*$'$

and so had

Bom$''$

(*assembly*	*qty*	*subassembly*	*assembly*$'$	*qty*$'$	*subassembly*$'$)
wallplug	1	cover	cover	1	plate
wallplug	1	cover	cover	2	screw
wallplug	1	fixture	fixture	2	plug
wallplug	1	fixture	fixture	2	screw
fixture	2	plug	plug	2	connector
fixture	2	plug	plug	1	mould

Then the virtual attribute *qtyp* defined by the domain algebra statement

let *qtyp* **be** *qty*×*qty*$'$;

could be actualized by the projection

[*assembly,qtyp,subassembly*$'$] **in** *Bom*$''$

to give

(*assembly*	*qtyp*	*subassembly*$'$)
wallplug	1	plate
wallplug	2	screw
wallplug	2	plug
fixture	4	connector
fixture	2	mould

We often write a virtual attribute outside the parentheses containing the attributes of a relation on which we intend to actualize it, in order to show the values it would eventually have.

Bom$''$

(*assembly*	*qty*	*subassembly*	*assembly*$'$	*qty*$'$	*subassembly*$'$)	*qtyp*
wallplug	1	cover	cover	1	plate	1
wallplug	1	cover	cover	2	screw	2
wallplug	1	fixture	fixture	2	plug	2
wallplug	1	fixture	fixture	2	screw	2
fixture	2	plug	plug	2	connector	4
fixture	2	plug	plug	1	mould	2

(And note that this display contains one important extra tuple missing from the projected actualization above. We will return to this issue at the end of this section.)

A domain algebra expression may contain any scalar operation on an attribute or any number of attributes. The above example, *qty*×*qty*$'$, is the simple binary operation of multiplication. These expressions are called "scalar", or "horizontal" because they operate within tuples and tuples are represented as horizontal rows in the most popular, tabular, representation of relations.

In addition, there may be "vertical" or "aggregate" operations such as summing all the values of a single attribute. An example is the group-by sum, or "equivalence reduction", that adds up the quantities, *qtyp*, for each pair of *assembly* and *subassembly*$'$:

let *qty*$''$ **be equiv** + **of** *qtyp* **by** *assembly*, *subassembly*$'$;

Note that **equiv** takes an operator such as + as an operand; the only restriction is that the operator must be commutative and associative or else the result is undefined because relations are unordered. This is also true for **red**, the operator that defines simple aggregation without the **by** clause.

The relational and domain algebras work together by using the relational algebra to actualize any virtual attribute. Here is the example of a bill-of-materials "explosion", the more elaborate recursion that we referred to earlier, using *BoM*.

```
        let assembly′ be assembly;
        let qty′ be qty;
        let subassembly′ be subassembly;
        let qty″ be equiv + of qty×qty′ by assembly, subassembly′;
        let qty‴ be qty + qty″;
        let qty be qty‴;
        let subassembly be subassembly′;
        BoMtc is [assembly,qty,subassembly] in [assembly,qty‴,subassembly′] in
           (BoM union [assembly,qty″,subassembly′] in
            (BoM [subassembly join assembly′] [assembly′,qty′,subassembly′] in BoMtc));
```

Note that we start with three domain algebra statements which simply rename attributes (making it unnecessary to have a separate copy, *BoM′*, of *BoM*). This is followed by the definition of *qty″*, having the same meaning as above, by a single domain algebra expression which combines both scalar and aggregate operations. The definition of *qty‴* is to combine quantities when *assembly* and *subassembly* are connected by paths of different lengths. The final two domain algebra statements rename again. Note that both are cyclic when considered together with the previous domain algebra statements. However, in the relational algebra that follows next, actualization is done in separate projections, so there are in fact no cycles. The final statement is a recursive view in the relational algebra which defines the "transitive closure" of *BoM*, in which the new quantity is the sum over parallel paths of the product along sequential edges of the original quantities.

The domain algebra has all been defined without reference to the one statement of relational algebra (but, of course, with it in mind). That one statement is fairly complex, but follows the same form as the view defining *Ancestor*, above: a **union** of the source relation, *BoM*, with a **join** of *BoM* with the recursively defined result, *BoMtc*. If such a result had to be defined with a domain algebra which explicitly named relations, it would be really intricate. (Try writing it in Prolog!).

Two points arise from this example. First, it avoids the loss of the second tuple involving `2 screws`, mentioned above, so that `wallplug` will indeed be shown to have `4 screws`. (This is from the sum over parallel paths. `Wallplug` also has `4 connectors`, due to the product over sequential edges.) Second, while cycles may appear in the definitions of virtual attributes, they must be resolved by actualization into non-cyclic (or nonrecursive) calculations. When we extend the discussion to nested relations, we will find it legitimate to have recursively defined attributes, so long as each recursive cycle bridges two different levels of nesting.

In section 1.1.5 we referred to conditional expressions in the discussion of updates. These provide another illustration of scalar domain algebra coupling with relational algebra, so we show how to change occurrences of `cover` in either attribute of *BoM* to `top`.

```
        update BoM change {
                assembly <− if assembly="cover" then "top" else assembly;
                subassembly <− if subassembly="cover" then "top" else subassembly;
        };
```

We can also improve the update from section 1.1.5 which uses **grep**:

```
        update BoM change eval attr <− (if x="wall" then "base" else x) cat "plug"
                using grep(attr;x) "\x plug" in BoM;
```

changes `wallplug` to `baseplug` wherever it appears as an attribute value in *BoM*.

1.3 Nesting: Attributes are Relations

The relations we have discussed so far are "flat", in keeping with Codd's first normal form [10]: no attribute may have set or tuple values. It is not appropriate in a programming language to have such a caste system, whereby some data types (sets, tuples, and hence relations) have fewer privileges than others (strings, integers, etc.). So we violate first normal form by the simple expedient of allowing, in each tuple, the values of any attribute to be relational.

Since this is an apparently expanded data structure, which we call nested relations, we should have a formalism to manipulate it, or it would be as useless for programming as relations without the relational and domain algebras. Fortunately, we find a formalism by an expedient as simple as that which gave nested relations: we permit the operators of the domain algebra to include the operators of the relational algebra. That is, for nesting, the domain algebra subsumes the relational algebra.

The only mechanisms we need beyond this idea are those for raising and lowering levels of nesting. Even here, we can almost entirely again avoid adding new syntax.

Here is a three-level nested relation, written out in a self-evident way. Outside the parentheses, we also show some virtual attributes which we discuss next.

company								
(*cname*	*address*			)	*compname*	*addcity*	*nameAddCity*	
	(*street*		*city*	*codezip*)	(*cname*)	(*city*)	(*cname*	*city*)
	(*num*	*cname*)						
Dink Inc.	1	Dink St	Dinkton	D1N3T0	Dink Inc.	Dinkton	Dink Inc.	Dinkton
	13	Dink St						
	1	Dink St	Dinkville	D1N3V1		Dinkville	Dink Inc.	Dinkville
FemtoSoft	10000	No Way	Rapa City	R8P8C1	FemtoSoft	Rapa City	FemtoSoft	Rapa City
KiloSoft	314	Speed Way	Adroit	48207	KiloSoft	Adroit	KiloSoft	Adroit

The first virtual attribute we declare introduces the new syntax for adding a new level of nesting.

let *compname* **be relation**(*cname*);

This makes *cname* an attribute of a new relation, which itself could be a virtual attribute of either *company* or *street*, both of which have *cname* as attribute: above we show the result only for *company*. The **relation**() syntax allows grouping of any number of attributes into a new relation, but the relation is a singleton: it has only one tuple for each tuple of the relation it is nested in (the "outer relation").

The second virtual attribute illustrates projection subsumed into the domain algebra.

let *addcity* **be** [*city*] **in** *address*;

This creates a virtual attribute, shown for *company*, which is a relation on the attribute *city*.

We now have two virtual attributes, *compname* and *addcity*, so we can combine them with a binary relational operator, **join**, in the domain algebra.

let *nameAddCity* **be** *compname* **join** *addcity*;

This gives a relation on the two attributes, *cname* and *city*, using the Cartesian product mode of **join**, since the two operands have no common attribute, and we did not explicitly specify join attributes (say, [*cname* **join** *city*]).

We could create a new top-level relation by projecting *nameAddCity* from *company*,

NameAddCity <− [*nameAddCity*] **in** *company*;

but note that the result is a nested relation (horizontal lines mark the boundaries between the outer tuples):

NameAddCity	
(nameAddCity)	
(*cname*	*city*)
Dink Inc.	Dinkton
Dink Inc.	Dinkville
FemtoSoft	Rapa City
KiloSoft	Adroit

We might like to "flatten" this into a single-level relation. This requires combining all the outer tuples into one, which we can do by taking the union of the inner relation, *nameAddCity*, with itself over all tuples: **red union of** *nameAddCity*. It also requires getting rid of the name, *nameAddCity*. We can do both of these by writing the reduction directly in a projection list, without giving the domain expression a name. Then the system has no choice but to raise the level.

NameCity <− [**red union of** *nameAddCity*] **in** *company*;

NameCity

(*cname*	*city*	)
Dink Inc.	Dinkton	
Dink Inc.	Dinkville	
FemtoSoft	Rapa City	
KiloSoft	Adroit	

Note that level raising requires no new syntax, just anonymous projection. But it does require care in making sure that the outer relation is a singleton, so that the anonymous expression usually involves a **red union of**.

Implementing nested relations is easy, and shows that their benefit is ease of thinking about certain problems rather than extending the capabilities of flat relations. We build a nested relation as a collection of flat relations, linked by "surrogate" values. Thus, *company* becomes

company

(*cname*	*address*)
Dink Inc.	37
FemtoSoft	22
KiloSoft	48

.address

(*.id*	*street*	*city*	*codezip*	)
37	144	Dinkton	D1N3T0	
37	156	Dinkville	D1N3V1	
22	132	Rapa City	R8P8C1	
48	111	Adroit	48207	

.street

(*.id*	*num*	*cname*)
144	1	Dink St
144	13	Dink St
156	1	Dink St
132	10000	No Way
111	314	Speed Way

These separate flat relations are implicitly joined on the attribute that is the name of the nested relation and the `.id` attribute of the nested relation. Note that the surrogate values held in these attributes permit one-to-many joins. Implementing the formalism that subsumes the relational algebra into the domain algebra is straightforward: the join is made and reductions are translated to equivalence reductions by `.id`. Where suitable, the full join need not be made, but only part of it, by a process of pointer dereferencing on secondary storage.

Update on nested relations likewise requires that statements involving the relational algebra be allowed after the **change** keyword, including, of course, nested **update** statements. For example, we change `Dinkton` to `Dinkburg` in *city* in *address* in *company*:

```
update company change
        update address change
                city <− if city="Dinkton" then "Dinkburg" else city;
```

A useful new operator of the domain algebra generates special nested attributes. **Transpose**, followed optionally by parameters of types **attribute, type**, or **any** creates a relation on attributes which are these parameters, containing a tuple for each attribute of the relation it is actualized in. Thus

let *xpose* **be transpose**(*attr,type,val*);

gives the virtual attribute *xpose* in *ComponentCost* from section 1.1.5

ComponentCost				
(*assembly*	*cost*)	*xpose*		
		(*attr*	*type*	*val*)
plate	0.75	assembly	string	plate
		cost	real	0.75
screw	0.05	assembly	string	screw
		cost	real	0.05
connector	0.20	assembly	string	connector
		cost	real	0.20
mould	1.35	assembly	string	mould
		cost	real	1.35

We see from this why types **type** and **any** must be introduced. Type **any** is an example of a union type, which we shall encounter again in section 5. We will mostly be interested in the **attribute** parameter in this paper, so do not pursue the others.

(Updates using **transpose** are similar to updates using **grep**, and provide a way to change an attribute name under certain circumstances, for example: **change** *attr* <− <*newAttribute*> will replace any selected values of *attr* by $\mathcal{DC}$ and introduce *newAttribute* polymorphically into the relation.)

Using **transpose**, we can begin a discussion of finding the paths in a nested relation (and hence its schema) which we shall be able to complete once we have considered recursive nesting in section 3. For this application, we also need a keyword, **self**, which returns the name of the relation it is actualized in. For *company*,

let *xpose* **be transpose**(*attr*);
let *path* **be self**/*attr*;

gives the virtual attributes *xpose* and *path* shown below for every subrelation (they are actualizable on all of the subrelations because they do not refer to any attributes). (The definition of *path* is equivalent to

let *path* **be self cat** "/" **cat** *attr*;

with suitable castings.)

company										
(*cname*	*address*							)	*xpose*	
	(*street*				*city*	*codezip*)	*xpose*		(*attr*)	*path*
	(*num*	*cname*)	*xpose*				(*attr*)	*path*		
			(*attr*)	*path*						
Dink Inc.	1	Dink St	num	street/num	Dinkton	D1N3T0	city	address/city	cname	company/cname
			cname	street/cname			codezip	address/codezip		
	13	Dink St	num	street/num						
			cname	street/cname						
	1	Dink St	num	street/num	Dinkville	D1N3V1	city	address/city		
			cname	street/cname			codezip	address/codezip		
FemtoSoft	10000	No Way	num	street/num	Rapa City	R8P8C1	city	address/city	cname	company/cname
			cname	street/cname			codezip	address/codezip		
KiloSoft	314	Speed Way	num	street/num	Adroit	48207	city	address/city	cname	company/cname
			cname	street/cname			codezip	address/codezip		

Note that **transpose** returns data only on scalar attributes; it does not report on nested attributes or provide recursive structure by itself. Doing that will be our goal in section 3. (Note also that we have cheated above: **self** would really pick up *xpose* in each case instead of the name of the relation two levels up (*street*, etc.). This can be fixed by not naming *xpose*, as we will see in section 3.)

The above result is highly redundant, because we did not ask **transpose** to produce a *value*, so we remove the repetitions and raise the level by an anonymous **red union**. For example:

let *paths* **be** [**red union of** [*path*] **in** *xpose*] **in** *street*;

gives the virtual attribute

company					
(*cname*	*address*				)
	(*street*		*city*	*codezip*)	*paths*
	(*num*	*cname*)			(*path*)
Dink Inc.	1	Dink St	Dinkton	D1N3T0	street/num
	13	Dink St			street/cname
	1	Dink St	Dinkville	D1N3V1	street/num
					street/cname
FemtoSoft	10000	No Way	Rapa City	R8P8C1	street/num
					street/cname
KiloSoft	314	Speed Way	Adroit	48207	street/num
					street/cname

This is still redundant, but before raising the level still higher, we modify the definition of *paths* to include the *path*s at the new level.

let *path*$'$ **be self**/*path*;
let *paths* **be** [**red union of** [*path*] **in** (([*path*] **in** *xpose*) [*path* **union** *path*$'$]
  [*path*$'$] **in** [**red union of** [*path*] **in** *xpose*] **in** *street*] **in** *address*;

giving

company					
(*cname*	*address*				)
	(*street*		*city*	*codezip*)	*paths*
	(*num*	*cname*)			(*path*)
Dink Inc.	1	Dink St	Dinkton	D1N3T0	address/street/num
	13	Dink St			address/street/cname
	1	Dink St	Dinkville	D1N3V1	address/city
					address/codezip
FemtoSoft	10000	No Way	Rapa City	R8P8C1	address/street/num
					address/street/cname
					address/city
					address/codezip
KiloSoft	314	Speed Way	Adroit	48207	address/street/num
					address/street/cname
					address/city
					address/codezip

One more union and level-raising will produce all the paths in *company*, but we will save this closure until we discuss recursive nesting in section 3.

2 Syntactic Sugar for Nested Queries

We can investigate in more detail elements which are likely to be frequently used in querying nested relations, and can thus be captured by simplified syntax. By "query" we mean mainly T-selectors and **grep** operations; we consider binary operations only briefly.

The T-selector (section 1.1) has three components: projection list, selection condition, and relational expression. The first two involve domain expressions, but since nesting subsumes relations as domains, the projection list and the relational expression may be considered together. The selection condition is a special kind of domain expression, with Boolean value, and must be considered separately.

We start with projection. We saw an example in section 1.3, which used anonymous **red union of** to raise the level of nesting. We start the present discussion by projecting *address* from *company* in this way.

Address <− [**red union of** *address*] **in** *company*;

Our syntactic sugar will turn this into

Address <− *company*/*address*;

where we concatenate attribute names into a path by the / operator (so that the path looks like a directory path in an operating system such as Unix). The result of either is

```
         Address
         (street                 city          codezip)
          (num   cname        )
             1   Dink St         Dinkton       D1N3T0
            13   Dink St
             1   Dink St         Dinkville     D1N3V1
         10000   No Way          Rapa City     R8P8C1
           314   Speed Way       Adroit        48207
```

We can go deeper.

Street <− [**red union of** [**red union of** *street*] **in** *address*] **in** *company*;

becomes

Street <− *company*/*address*/*street*;

giving the top-level relation, *Street*(*num,cname*). Going all the way to a leaf looks different in the unsweetened syntax:

Cname <− [**red union of** [**red union of** [*cname*] **in** *street*] **in** *address*] **in** *company*;

becomes

Cname <− *company*/*address*/*street*/*cname*;

giving *Cname*(*cname*).

Now suppose we wanted to project *cname* either directly from *company* or, two levels down, from *street*.

```
Cname <− [red union of
              relation(cname) union [red union of [cname] in street]
              in address] in company;
```

or, eliding the **relation**() operator since it has only one operand, *cname*

```
Cname <− [red union of cname union [red union of [cname] in street]
              in address] in company;
```

becomes

Cname <− *company*/(*address*/*street*/)?*cname*;

where the regular expression operator, ?, allows zero or one occurrences of its operand, *address*/*street*/. The result of this last is

```
Cname
(cname)
Dink Inc.
FemtoSoft
KiloSoft
Dink St
No Way
Speed Way
```

The syntactic sugar can be used at either end of a regular T-selector.

company/*address*/*street*/*cname*;

is the same as

address/*street*/*cname* **in** *company*;

or

street/*cname* **in** *company*/*address*;

or

cname **in** *company*/*address*/*street*;

where we have omitted the brackets, [..], around the projection list, since only one attribute is projected.

To project multiple attributes from some deeper level is also easy.

[**red union of** [**red union of** [*num, cname*] **in** *street*] **in** *address*] **in** *company*;

is

[*num, cname*] **in** *company/address/street*;

Selection introduces Boolean selection conditions. Suppose we wanted to find tuples of *company* where *address/city* is `Dinkton`.

DinkCity <− **where** ([] **where** *city*=`"Dinkton"` **in** *address*) **in** *company*;

which, of course, becomes the syntactic sugar

DinkCity <− **where** *address/city*=`"Dinkton"` **in** *company*;

Note that attribute paths in a selection condition have a different interpretation from attribute paths serving as projections. The nullary expression, [] **where** *city*=`"Dinkton"` **in** *address*, is just the Boolean "something where city = 'Dinkton' in address" of section 1.1.1.

Now move from T-selectors to the second unary relational operator, **grep**. In select mode (section 1.1.3), **grep** is not recursive:

grep `"Way"` **in** *company*

returns an empty result.

grep(*attr,pos*) `"Dink"` **in** *company*

gives, because `"Dink"` is a substring of the top-level scalar attribute, *cname*

```
(cname      address                                    )  attr    pos
            (street              city         codezip  )
            (num   cname)
Dink Inc.      1   Dink St   Dinkton     D1N3T0        cname    0
              13   Dink St
               1   Dink St   Dinkville   D1N3V1
```

We must be explicit to go deeper. For example, to find attributes containing "Dink" in *street*, we can proceed in two steps

let *streetDink* **be grep**(*attr*) `"Dink"` **in** *street*;
[**red union of** [**red union of** [*attr*] **in** *streetDink*] **in** *address*] **in** *company*

or in one step

[**red union of** [**red union of** [*attr*] **in grep**(*attr*) `"Dink"` **in** *street*] **in** *address*]
in *company*

Syntactic sugar makes this shorter.

attr **in grep**(*attr*) `"Dink"` **in** *company/address/street*

Now that we have established two meanings for expressions concatenating attributes into paths, depending whether the context is a projection or a condition, we can easily interpret such paths if they appear as arguments or operands of the binary operators.

Update..change uses a third interpretation of the same syntactic sugar. Changing `Dinkton` to `Dinkburg` in *city* in *address* in *company* at the end of section 1.3 can be written:

```
update company/address change
        city <− if city="Dinkton" then "Dinkburg" else city;
```

(but not

```
update company change
        address/city <− if city="Dinkton" then "Dinkburg" else city;
```

since path expressions are not maeningful on the left of an assignment). The first abbreviates

```
update company change
    update address change
        city <− if city="Dinkton" then "Dinkburg" else city;
```

We have, thus far, introduced only one regular expression operator, *i.e.*, `"?"`. More such operators appear as we elaborate further in the next three sections.

3 Recursive Nesting

We were led to nested relations by the consideration that limiting some data types relative to others complicates a programming language by multiplying special cases. A similar consideration leads to recursive data types, in our case, recursive nesting: why should a relation not be an attribute of itself? The bill of materials appears to be a potential application.

assembly								
(*component*	*subassembly*							)
	(*qty*	*component*	*subassembly*					)
			(*qty*	*component*	*subassembly*			)
					(*qty*	*component*	*subassembly*	)
`wallplug`	1	`cover`	1	`plate`	$\mathcal{DC}$			
			2	`screw`	$\mathcal{DC}$			
	1	`fixture`	2	`plug`	1	`mould`	$\mathcal{DC}$	
					2	`connector`	$\mathcal{DC}$	
			2	`screw`	$\mathcal{DC}$			

(The null value, $\mathcal{DC}$, is one of two possible nulls, and stands for irrelevant, or "don't care".)

To find all components from this, the domain algebra must allow recursive definitions of virtual attributes across levels of nesting.

let *cmpnt* **be** *component* **union** [**red union of** *cmpnt*]
in *subassembly*;

As with relational recursion, *cmpnt* is initially empty. At the lowest level, it then takes on the value of *component*. Apart from the recursion and initialization, we recognize the idiom that translates to the syntactic sugar (*subassembly*/)?*component*.

Considering initialization and recursion gives the Kleene star in the syntactic sugar.

let *cmpnt* **be** (*subassembly*/)**component*;

Alternatively, we could write the projection directly.

assembly/(*subassembly*/)**component*;

These each give

(*component*)
`wallplug`
`cover`
`fixture`
`plate`
`screw`
`plug`
`mould`
`connector`

The Kleene star gives our syntactic sugar half the capabilities of regular expressions. The other half, alternatives, will appear in section 5.

Implementing recursive nesting adds nothing new. The *.id* attribute is enough. The flat relations for *assembly* are *assembly*(*component,subassembly*) and *.subassembly*(*.id,component,subassembly*), with *.subassembly*/*subassembly* and *.subassembly*/*.id* sharing surrogate values to build the recursion.

assembly		*.subassembly*			
(*component*	*subassembly*)	(*.id*	*qty*	*component*	*subassembly*)
wallplug	13	13	1	cover	27
		13	1	fixture	24
		27	1	plate	$\mathcal{DC}$
		27	2	screw	$\mathcal{DC}$
		24	2	plug	31
		24	2	screw	$\mathcal{DC}$
		31	2	mould	$\mathcal{DC}$
		31	2	connector	$\mathcal{DC}$

[2]

The mechanisms we have explored for recursive nesting allow us to exploit fully another useful feature, the "wildcard", even for fixed nesting. Suppose we wanted to project *cname* from all levels of *company* (section 2), but did not wish to write down the attribute names for intermediate levels (or perhaps do not even know what they are).

Cname <– [*cname* **union** [**red union of** [**red union of** *cname*] **in** .] **in** .]
    in *company*;

or

Cname <– *company*/(./.)?*cname*;

We can go further, and not even bother with how many levels there are, by using domain algebra recursion

let *cn* **be** *cname* **union** [**red union of** *cn*] **in** .;
let *cname* **be** *cn*;
Cname <– [**red union of** *cname*] **where !null**(*cname*) **in** [*cn*] **in** *company*;

This works through intermediate levels containing no *cname* attribute if we assume that the absence of an attribute is the same thing as the presence of that attribute with nothing but null values (hence the test for nullity). This is a plausible assumption, being the converse of considering an attribute with all null values in a relation not to be an attribute of the relation (section 1.1.5).

The syntactic sugar for this uses the Kleene star

Cname <– *company*/(./)*cname*;

which can be abbreviated to

Cname <– *company*/*/*cname*;

or even

Cname <– *company*//*cname*;

To find *cname* more than one level down,

Cname <– *company*/(./)+*cname*;

[2]It is instructive to write the code that "explodes" the bill of materials in this recursively nested form. It follows the lines of the explosion of the flat *BoM*, but is a little more involved. We write it explicitly without sugar (or further explanation except to note the use of syntax to initialize a recursion).

let qty' **be** *qty*;
let $component'$ **be** *component*;
let *suba1* **be** [qty',$component'$] **in relation** (*qty,component*) **join** [*qty,component*]
    in *suba0*;
let qty'' **be** $qty \times qty'$;
let *suba2* **be relation** (*qty,component*) **union** [$component'$,qty'',*component*]
    in *suba1*;
let qty''' **be** $qty + qty''$;
let *suba3* **be** [$component'$,qty''',*component*] **in** *suba2*;
let qty'''' **be equiv** + **of** qty''' **by** *component*;
let *qty* **be** qty'''';
let *suba0* **initial** *subassembly* **be** [*qty,component*] **in** [qty'''',*component*]
    in [**red union of** *suba3*] **in** *subassembly*;
assemblyExplode <– [**red union of** *suba0*] **in** *assembly*;

is the syntactic sugar for

let *cn* **be** *cname* **union** [**red union of** *cn*] **in** .;
let *cname* **be** *cn*;
Cname <− [**red union of** *cname*] **where** !**null**(*cname*) **in** [**red union of** *cn*]
  in *company*;

If we also allow specialized constructs supported by **grep**, such as

[<any attribute in list>]	
[ˆ <any attribute not in list>]	
<regex> | <regex>	alternatives
\ <metasymbol>	e.g., \., *, \+, \?, \!, \(, \), \$, \[, \], \ˆ, \ |, ..

we can multiply them with ∗ or + in the same way. For example, to find *cname* but not in *street*,

Cname <− *company*/([ˆ(*street*)]/)**cname*;

would be the syntactic sugar for

let *cn* **be** *cname* **union** [**red union of** *cn*] **in** [ˆ(*street*)];
let *cname* **be** *cn*;
Cname <− [**red union of** *cname*] **where** !**null**(*cname*) **in** [*cn*] **in** *company*;

Parentheses are needed around *street* in the bracketted exclusion list so it is not taken as an exclusion of the letters `s, t, r,` or `e`, which is the interpretation of this regular expression when applied to strings as opposed to attribute names.

The most general expression we can make with this syntactic sugar is

company(/.)*   or *company*//

which *flattens company* to

(*cname*	*city*	*codezip*	*num*)
Dink Inc.	$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$
$\mathcal{DC}$	Dinkton	D1N3T0	$\mathcal{DC}$
$\mathcal{DC}$	Dinkville	D1N3V1	$\mathcal{DC}$
Dink St.	$\mathcal{DC}$	$\mathcal{DC}$	1
Dink St.	$\mathcal{DC}$	$\mathcal{DC}$	13
Femtosoft	$\mathcal{DC}$	$\mathcal{DC}$	$\mathcal{DC}$
$\mathcal{DC}$	Rapa City	R8P8C1	$\mathcal{DC}$
No Way	$\mathcal{DC}$	$\mathcal{DC}$	10000
$\mathcal{DC}$	Adroit	48207	$\mathcal{DC}$
Speed Way	$\mathcal{DC}$	$\mathcal{DC}$	314

given a suitable definition of **union** for relations with only some, or no, common attributes. This flattening is simpler for a recursively nested relation:

assembly//

gives

(*qty*	*component*)
$\mathcal{DC}$	wallplug
1	cover
1	fixture
1	plate
2	screw
2	plug
1	mould
2	connector

Of course, the hierarchical relationships among the components are lost: this operation does not generate the ternary flat relation retaining these relationships.

Finally, we can finish the code we started at the end of section 1.3 to find all paths in the nested relation *company*.

```
let path be self/attr;
let path′ be self/path;
let paths be [path] in (([path] in transpose(attr)) [path union path′]
        ([path′] in [red union of paths] in .));
[red union of paths] in company
```

(Because the **union** is qualified by attributes in [*path* **union** *path*′], this code is now too specialized to be reducible to simple syntactic sugar such as
company//**transpose**(*attr*)/*path*.)

The result is

```
(path)
city/address/street/num
city/address/street/cname
city/address/city
city/address/codezip
city/cname
```

4 Links: Common Subexpressions and Crossreferences

Nested relations so far, even with recursive nesting, form hierarchies which are strict trees. This precludes the possibility of data sharing, and certainly of cycles. For example, `Dink St` is represented twice in the example of section 1.3. Since there is nothing in the flat relational data structure we use to implement nested relations that precludes directed acyclic graphs (*DAG*s) from being represented, or even cycles, we should look at this possibility for saving storage and for the further advantages of data sharing such as the impossibility of inconsistently updating one branch of a tree and not the other.

The bill of materials of section 3 provides an example of data sharing. Suppose we also wished to describe the subassembly `fixture` as a product, and hence as an assembly. Given only a tree structure, we would be forced to repeat the data for `fixture`, one level higher up in the nesting.

assembly								
(*component*	*subassembly*							)
	(*qty*	*component*	*subassembly*					)
			(*qty*	*component*	*subassembly*			)
					(*qty*	*component*	*subassembly*	)
`wallplug`	1	`cover`	1	`plate`	$\mathcal{DC}$			
			2	`screw`	$\mathcal{DC}$			
	1	`fixture`	2	`plug`	1	`mould`	$\mathcal{DC}$	
					2	`connector`	$\mathcal{DC}$	
			2	`screw`	$\mathcal{DC}$			
`fixture`	2	`plug`	1	`mould`	$\mathcal{DC}$			
			2	`connector`	$\mathcal{DC}$			
	2	`screw`	$\mathcal{DC}$					

and this would be reflected in the implementation.

assembly		*.subassembly*			
(*component*	*subassembly*)	(*.id*	*qty*	*component*	*subassembly*)
`wallplug`	13	13	1	`cover`	27
`fixture`	14	13	1	`fixture`	24
		27	1	`plate`	$\mathcal{DC}$
		27	2	`screw`	$\mathcal{DC}$
		24	2	`plug`	31
		24	2	`screw`	$\mathcal{DC}$
		31	2	`mould`	$\mathcal{DC}$
		31	2	`connector`	$\mathcal{DC}$
		14	2	`plug`	21
		14	2	`screw`	$\mathcal{DC}$
		21	2	`mould`	$\mathcal{DC}$
		21	2	`connector`	$\mathcal{DC}$

Since the implementation is easily able to save this redundancy, by changing a surrogate,

assembly		*.subassembly*			
(*component*	*subassembly*)	(*.id*	*qty*	*component*	*subassembly*)
`wallplug`	13	13	1	`cover`	27
`fixture`	24	13	1	`fixture`	24
		27	1	`plate`	$\mathcal{DC}$
		27	2	`screw`	$\mathcal{DC}$
		24	2	`plug`	31
		24	2	`screw`	$\mathcal{DC}$
		31	2	`mould`	$\mathcal{DC}$
		31	2	`connector`	$\mathcal{DC}$

we should have a way of indicating value sharing in the programmer representation. We can suggest using a *label* of some sort, which we are fairly free to invent.

assembly										
(*component*	*subassembly*			)						
		(*qty*	*component*	*subassembly*			)			
					(*qty*	*component*	*subassembly*			
								(*qty*	*component*	*subassembl*
`wallplug`		1	`cover`		1	`plate`	$\mathcal{DC}$			
					2	`screw`	$\mathcal{DC}$			
		1	`fixture`	`fixture:`	2	`plug`		1	`mould`	$\mathcal{DC}$
								2	`connector`	$\mathcal{DC}$
					2	`screw`	$\mathcal{DC}$			
`fixture`	`subassembly:fixture`									

The term "label" is reminiscent of **goto** in programming languages, which was considered undesirable because obscure code can result. In fact, shared subexpressions better resemble subroutines, since they have clear terminations as well as starts. We will use *subex* technically instead of "label" (although informally, we may use "label" since it is an English word).

We say we are "fairly free to invent" a representation of labels because the above is only a print representation of the data. We have yet to discuss a true programmer representation, to be used, for instance, in initializing (nested) relations. Now is the appropriate occasion to do this.

There are many possibilities, but we introduce a "mark-up" representation called xML, where $x \in \{$G, HT, SG, X, ..$\}$, so that xML has the characteristics of all the markup notations that originated with GML. This is quite a redundant representation, because it repeats the schemas for the data that we have been assuming were declared in the program prior to use (see section 5). It has the attraction of freeing us from such declarations should we want or need to avoid them. We revisit this discussion in section 6

We start with the simple, flat relation, $BoM(assembly,qty,subassembly)$, from section 1.1.

```
<Bom>
  <assembly>wallplug</assembly> <qty>1</qty> <subassembly>cover</subassembly>
  <assembly>wallplug</assembly> <qty>1</qty> <subassembly>fixture</subassembly>
  <assembly>cover</assembly> <qty>1</qty> <subassembly>plate</subassembly>
  <assembly>cover</assembly> <qty>2</qty> <subassembly>screw</subassembly>
  <assembly>fixture</assembly> <qty>2</qty> <subassembly>plug</subassembly>
  <assembly>fixture</assembly> <qty>2</qty> <subassembly>screw</subassembly>
  <assembly>plug</assembly> <qty>2</qty> <subassembly>connector</subassembly>
  <assembly>plug</assembly> <qty>1</qty> <subassembly>mould</subassembly>
</Bom>
```

This straightforward way of encoding a relation is fine if there are no complications such as omitted values. A more elaborate xML representation introduces a hidden tag, `<.tuple>`, to separate tuples.

```
<Bom>
  <.tuple>
   <assembly>wallplug</assembly> <qty>1</qty> <subassembly>cover</subassembly>
  </.tuple>
  <.tuple>
   <assembly>wallplug</assembly> <qty>1</qty> <subassembly>fixture</subassembly>
  </.tuple>
  <.tuple>
   <assembly>cover</assembly> <qty>1</qty> <subassembly>plate</subassembly>
  </.tuple>
  <.tuple>
   <assembly>cover</assembly> <qty>2</qty> <subassembly>screw</subassembly>
  </.tuple>
  <.tuple>
   <assembly>fixture</assembly> <qty>2</qty> <subassembly>plug</subassembly>
  </.tuple>
  <.tuple>
   <assembly>fixture</assembly> <qty>2</qty> <subassembly>screw</subassembly>
  </.tuple>
  <.tuple>
   <assembly>plug</assembly> <qty>2</qty> <subassembly>connector</subassembly>
  </.tuple>
  <.tuple>
   <assembly>plug</assembly> <qty>1</qty> <subassembly>mould</subassembly>
  </.tuple>
</Bom>
```

The straightforward form can work with nesting. Here is the xML representation of the recursively nested bill of materials, including a linking label, `subex`, and the additional entry for `fixture`.

```
<assembly>
  <component>wallplug</component>
  <subassembly>
    <qty>1</qty>
    <component>cover</component>
    <subassembly>
      <qty>1</qty>
      <component>plate</component>
      <qty>2</qty>
      <component>screw</component>
    </subassembly>
    <qty>1</qty>
    <component>fixture</component>
    <subassembly subex="fixture">
      <qty>2</qty>
      <component>plug</component>
      <subassembly>
        <qty>1</qty>
        <component>mould</component>
        <qty>2</qty>
        <component>connector</component>
      </subassembly>
      <qty>2</qty>
      <component>screw</component>
    </subassembly>
  </subassembly>
  <component>fixture</component>
  <subassembly:fixture/>
</assembly>
```

Note that the label, `fixture` (same name as the *component*, but that poses no problem), is the value of the xML "attribute", [3] subex, of the *subassembly* tag. Note also that the second tuple of *assembly* is a type:label pair (see section 5). It has a crossreference instead of a value, so we combine the tag with its endtag, giving `<subassembly:fixture/>`. (A more conventional xML alternative could be `<subassembly link="fixture"/>`, but we can avoid the extra keyword.)

In the above xML, some of the relational attributes are missing—those corresponding to $\mathcal{DC}$ in the print representation—but they are all the last attribute, *subassembly*, in the tuple, and so no confusion arises. Since it may help the reader, we show the same xML, with the `<.tuple>` hidden tag.

```
<assembly>
  <.tuple>
    <component>wallplug</component>
    <subassembly>
      <.tuple>
        <qty>1</qty>
        <component>cover</component>
        <subassembly>
          <.tuple>
            <qty>1</qty>
            <component>plate</component>
          </.tuple>
          <.tuple>
            <qty>2</qty>
```

[3] Unfortunately, "attribute" is used both in xML and in relations, with different meanings. We use it mainly in the relational sense in this paper, but hope that context will reveal our occasional use in the xML sense.

```
            <component>screw</component>
          </.tuple>
        </subassembly>
      </.tuple>
      <.tuple>
        <qty>1</qty>
        <component>fixture</component>
        <subassembly subex="fixture">
          <.tuple>
            <qty>2</qty>
            <component>plug</component>
            <subassembly>
              <.tuple>
                <qty>1</qty>
                <component>mould</component>
              </.tuple>
              <.tuple>
                <qty>2</qty>
                <component>connector</component>
              </.tuple>
            </subassembly>
          </.tuple>
          <.tuple>
            <qty>2</qty>
            <component>screw</component>
          </.tuple>
        </subassembly>
      </.tuple>
    </subassembly>
  </.tuple>
  <.tuple>
    <component>fixture</component>
    <subassembly:fixture/>
  </.tuple>
</assembly>
```

Links can also make cyclic references. For example, companies can be customers of (other) companies, so if we added a *customer* attribute to *company* (section 1.3), it could contain links back to *company*. As long as a company is unlikely to be its own customer, the data would not be cyclic, but the schema is. Here is the print representation of *company*, with a *customer* attribute. And just to show that even cyclic *data* causes no problem in principle, we make `Dink Inc.` its own customer.

```
company  (cname      address                                         customer)
                     (street                    city        codezip)
                      (num   cname         )
Dink     Dink Inc.       1   Dink St            Dinkton     D1N3T0   company:Dink
                        13   Dink St
                         1   Dink St            Dinkville   D1N3V1
𝒟𝒞       FemtoSoft   10000   No Way             Rapa City   R8P8C1   company:Dink
𝒟𝒞       KiloSoft      314   Speed Way          Adroit      48207    𝒟𝒞
```

The implementation representation with flat relations and surrogates adds an *.id* attribute to the outer relation, *company*:

company

(*.id*	*cname*	*address*	*customer*)
7	Dink Inc.	37	7
$\mathcal{DC}$	FemtoSoft	22	7
$\mathcal{DC}$	KiloSoft	48	$\mathcal{DC}$

and uses the same inner relations as in section 1.3:

.address

(*.id*	*street*	*city*	*codezip*)
37	144	Dinkton	D1N3T0
37	156	Dinkville	D1N3V1
22	132	Rapa City	R8P8C1
48	111	Adroit	48207

.street

(*.id*	*num*	*cname*)
144	1	Dink St
144	13	Dink St
156	1	Dink St
132	10000	No Way
111	314	Speed Way

The *x*ML representation is

```
<company>
 <.tuple coex=Dink>
  <cname>Dink Inc.</cname>
  <address>
    <street>
      <num type=integer>1</num> <cname>Dink St</cname>
      <num type=integer>13</num> <cname>Dink St</cname>
    </street>
    <city>Dinkton</city>
    <codezip>D1N3T0</codezip>
    <street>
      <num type=integer>1</num> <cname>Dink St</cname>
    </street>
    <city>Dinkville</city>
    <codezip>D1N3V1</codezip>
  </address>
  <customer:company:Dink/>
 </.tuple>
  <cname>FemtoSoft</cname>
  <address>
    <street>
      <num type=integer>10000</num> <cname>No Way</cname>
    </street>
    <city>Rapa City</city>
    <codezip>R8P8C1</codezip>
  </address>
  <customer:company:Dink/>
  <cname>KiloSoft</cname>
  <address>
    <street>
      <num type=integer>314</num> <cname>Speed Way</cname>
    </street>
    <city>Adroit</city>
    <codezip>48207</codezip>
  </address>
</company>
```

where we use a `<.tuple>` tag to identify the first outer tuple, and give it a `coex` (xML-)attribute in order to label it `"Dink"` for the cross-reference. Unlike the `fixture` *subex* attribute for the Bill of Materials example, the link tags `<customer:company:Dink/>` have an extra type qualifier because the referenced attribute has a different name from the referring attribute.

We have used `coex`, for "co-expression", instead of `subex`, as a pragma, since a cross-reference link gives rise to cycles in the schema, unlike a common subexpression link. In general, "coex" and "subex" are interchangeable in the syntax. If we are labelling tuples, we must identify them with `.tuple` tags, which have label (xML-)attributes. Between this and the *BoM* example, we see how we can label either a whole relation or an individual tuple. Note that links do not behave as "object identifiers": many tuples may all have the same link value.

The base type **string** is assumed. Data to be translated to **integer** is typed. (Since xML is sequential, we could type an attribute, such as *num*, only the first time it appears, and assume the type persists over subsequent appearances unless explicitly changed.)

4.1 Queries

For T-selector queries, links require nothing new. In the case of common subexpressions, where the data forms a DAG, the shared data can be thought of, for query purposes, as replicated at each sharing position, thus converting the DAG to a tree. In the case of cross-references, or co-expressions, cycles appear, and this means that we may need to use the query mechanisms introduced for recursive nesting, notably the Kleene star.

Here is a projection on the company name.

let *cn* **be** *cname* **union** [**red union of** *cn*] **in** *customer*;
let *cname* **be** *cn*;
[**red union of** *cname*] **in** [*cn*] **in** *company*

or

company/(*customer*/)**cname*

gives all three company names, `Dink Inc.`, `FemtoSoft`, and `KiloSoft`, in a relation with attribute *cname*. In principle, this code would loop forever, because of the data cycle on `Dink Inc.`, but since the result is a relation, the loop can stop as soon as the relation stops changing: we discover that `Dink Inc.` is the only addition each time around the cycle, and it was already in the answer.

5 Union Types: Alternative Forms of Attributes

In section 4, we used data constructs such as `subassembly:fixture` and `company:Dink` and called them type:label pairs, without saying anything about types. We also mentioned schema declarations. Allowing alternatives for attributes, such as *address* being either a simple string or the nested construct we have used so far, introduces dynamic typing, and requires us to start with these issues.

We might make the following declarations for *company* as we have formulated it so far.

domain *cname* **string**;
domain *city* **string**;
domain *codezip* **string**;
domain *num* **integer**;
domain *street*(*num,cname*);
domain *address*(*street,city,codezip*);
domain *customer company*;
domain *company*(*cname,address,customer*);
relation *company*(*cname,address,customer*); [4]

[4]The declarations for the recursively nested *assembly* would include
domain *subassembly*(*qty, component, subassembly*);
relation *assembly*(*component, subassembly*);

Here, nested relations are declared as attributes (**domain**) which may be of base type (**integer, string, ..**) or themselves relations. Note that the outer relation is now also declared as a domain. The linked attribute, *customer*, is declared to be of "type" *company*, in order for us to be able to refer to, for example, *customer/cname*, where *cname* is an attribute of *company*.

Thus, there are both base and composite types, and we consider relations or domains that are nested relations to be the latter.

To allow *address* to be a simple string as an alternative to the composite (*street,city,codezip*), we could write

domain *stctcd* (*street,city,codezip*);
domain *address* **string**|*stctcd*;

We could also allow *customer* to be a single value, as we have done so far, or a nested relation with multiple values. The values could either be strings or links to *company*, as above.

domain *namcomp cname*|*company*;
domain *custrel*(*namcomp*);
domain *customer cname*|*company*|*custrel*;

With these replacements for *address* and *customer*, the print representation of *company* becomes

company(	*cname*	*address*				*customer*	)
Dink	Dink Inc.	*stctcd*:				*company*:Dink	
		(*street*		*city*	*codezip*)		
		(*num*	*cname*)				
		1	Dink St	Dinkton	D1N3T0		
		13	Dink St				
		1	Dink St	Dinkville	D1N3V1		
$\mathcal{DC}$	FemtoSoft	10000 No Way, Rapa City R8P8C1				*custrel*:	
						(*namcomp*)	
						company:Dink	
						cname:Joe Dink Jr	
						cname:Bill Hatch	
$\mathcal{DC}$	KiloSoft	314 Speed Way, Adroit 48207				*cname*:Joe Dink III	

Note that we do not mention the base types (**string, integer,**..) explicitly: since this is print representation, they are displayed as strings.

The implementation representation with flat relations and surrogates adds an *.id* attribute to the outer relation, *company*.

company

(*.id*	*cname*	*address*	*customer*)
7	Dink Inc.	stctcd:37	company:7
$\mathcal{DC}$	FemtoSoft	10000 No Way, Rapa City R8P8C1	custrel:17
$\mathcal{DC}$	KiloSoft	314 Speed Way, Adroit 48207	cname:Joe Dink III

.stctcd

(*.id*	*street*	*city*	*codezip*)
37	144	Dinkton	D1N3T0
37	156	Dinkville	D1N3V1

.street

(*.id*	*num*	*cname*)
144	1	Dink St
144	13	Dink St
156	1	Dink St

.custrel

(*.id*	*namcomp*)
17	company:7
17	cname:Joe Dink Jr
17	cname:Bill Hatch

(We have omitted to identify base types, this time to simplify the presentation. If the schema is incomplete or not available, type information will be needed in the implementation.)

In xML representation,

```
<company>
 <.tuple coex=Dink>
  <cname>Dink Inc.</cname>
  <address type=stctcd>
    <street>
      <num type=integer>1</num> <cname>Dink St</cname>
      <num type=integer>13</num> <cname>Dink St</cname>
    </street>
    <city>Dinkton</city>
    <codezip>D1N3T0</codezip>
    <street>
      <num type=integer>1</num> <cname>Dink St</cname>
    </street>
    <city>Dinkville</city>
    <codezip>D1N3V1</codezip>
  </address>
  <customer:company:Dink/>
 </.tuple>
  <cname>FemtoSoft</cname>
  <address>10000 No Way, Rapa City R8P8C1</address>
  <customer type=custrel>
    <namcomp:company:Dink/>
    <namcomp type=cname>Joe Dink Jr</namcomp>
    <namcomp type=cname>Bill Hatch</namcomp>
  </customer>
  <cname>KiloSoft</cname>
  <address>314 Speed Way, Adroit 48207</address>
  <customer type=cname>Joe Dink III</customer>
</company>
```

Note the difference between the tags `<namcomp:company:Dink/>` and `<namcomp type=cname>Joe Dink Jr</namcomp>`. In the second, `Joe Dink Jr` is a stored value, and so must appear between separate start and end tags, and type information must be given as an xML attribute. In the first, there is no value, and while we could have used xML attributes, as in `<namcomp type=company link=Dink/>`, the more compact form does not need new keywords, and supports unlimited expansion of types if needed.

5.1 Queries

6 Schema Discovery

Given a nested relation for which we may not know the schema, it would be nice to find it out. For example, given the relation of section 1.3

```
company
(cname   address                                )
         (street              city   codezip)
         (num     cname)
```

we would like to generate a schema relation

schema			
(*attr*	*schema*		)
	(*attr*	*schema*)	
		(*attr*)	
`cname`			
`address`	`street`	`num`	
		`cname`	
	`city`		
	`codezip`		

which gives its schema as relational data.

We do it with a combination of **transpose** and **relation**, both from section 1.3, in a recursive attribute definition.

let *attr* **be self**;
let *schema* **be transpose**(*attr*) **union** [*attr,schema*] **in** .;
schema **in** *company*

where we see constructs we used in sections 1.3 and 3 to find all attribute paths.

We can expand this example to show the steps of the recursion.

Recursion 0.

company										
(*cname*	*address*								) *attr*	*schema*
	(*street*				*city*	*codezip*)	*attr*	*schema*		(*attr*)
	(*num*	*cname*)	*attr*	*schema*				(*attr*)		
				(*attr*)						
			`street`	`num`			`address`	`city`	`company`	`cname`
				`cname`				`codezip`		
`Dink Inc.`	`1`	`Dink St`			`Dinkton`	`D1N3T0`				
	`13`	`Dink St`								
	`1`	`Dink St`			`Dinkville`	`D1N3V1`				
`FemtoSoft`	`10000`	`No Way`			`Rapa City`	`R8P8C1`				
`KiloSoft`	`314`	`Speed Way`			`Adroit`	`48207`				

Recursion 1. (Since the data in *company* does not affect the result, we leave it out in the following.)

company												
(*cname*	*address*									) *attr*	*schema*	
	(*street*				*city*	*codezip*)	*attr*	*schema*			(*attr*	*schema*)
	(*num*	*cname*)	*attr*	*schema*				(*attr*	*schema*)			(*attr*)
				(*attr*)					(*attr*)			
			`street`	`num`			`address`	`city`		`company`	`cname`	
				`cname`				`codezip`			`address`	`city`
								`street`	`num`			`codezip`
									`cname`			

Recursion 2. (This is the last iteration for this example. Note that the *schema* attribute of *company* is the desired result.)

company													
(*cname*	*address*									) *attr*	*schema*		
	(*street*				*city*	*codezip*)	*attr*	*schema*			(*attr*	*schema*)	
	(*num*	*cname*)	*attr*	*schema*				(*attr*	*schema*)			(*attr*	*schema*)
				(*attr*)					(*attr*)				(*attr*)
			`street`	`num`			`address`	`city`		`company`	`cname`		
				`cname`				`codezip`			`address`	`city`	
								`street`	`num`			`codezip`	
									`cname`			`street`	`num`
													`cname`

We can also extract the types of the attributes.

let *attr* **be self**;
let *schema* **be transpose**(*attr, type*) **union** [*attr,type,schema*] **in** .;

to give

schema					
(*attr*	*type*	*schema*			)
		(*attr*	*type*	*schema*	)
				(*attr*	*type*)
cname	**string**				
address	$\mathcal{DC}$	street	$\mathcal{DC}$	num	**integer**
				cname	**string**
		city	**string**		
		codezip	**string**		

This is useful when the schema has links (even if they are cyclic). In the extended *company* relation of section 4, *customer* may be, cyclically, a *company*.

company	(*cname*	*address*				*customer*)
		(*street*		*city*	*codezip*)	
		(*num*	*cname*)			
Dink	Dink Inc.	1	Dink St	Dinkton	D1N3T0	Dink
		13	Dink St			
		1	Dink St	Dinkville	D1N3V1	
$\mathcal{DC}$	FemtoSoft	10000	No Way	Rapa City	R8P8C1	Dink
$\mathcal{DC}$	KiloSoft	314	Speed Way	Adroit	48207	$\mathcal{DC}$

Here is the schema, generated by the code above.

schema					
(*attr*	*type*	*schema*			)
		(*attr*	*type*	*schema*	)
				(*attr*	*type*)
cname	**string**				
address	$\mathcal{DC}$	street	$\mathcal{DC}$	num	**integer**
				cname	**string**
		city	**string**		
		codezip	**string**		
customer	*company*				

where the possibly cyclic recursion is captured by the self-reference in type *company*.

A more complicated example is from section 5. Here, *address* is a **string** or a tuple,

stctcd(*street,city,codezip*),

and *customer* is a **string**, *cname*, or a cyclic reference to *company*, or, thirdly, a relation, *custrel*, of many companies or *cname*s. For this example, the above code gives

schema						
(*attr*	*type*	*schema*				)
		(*attr*	*type*	*schema*		)
				(*attr*	*type*	*schema*)
cname	**string**					
address	stctcd	street	**relation**	num	**integer**	$\mathcal{DC}$
		cname	**string**	$\mathcal{DC}$		
		city	**string**	$\mathcal{DC}$		
		codezip	**string**	$\mathcal{DC}$		
address	**string**	$\mathcal{DC}$				
customer	*company*	$\mathcal{DC}$				
customer	custrel	namcomp	*company*	$\mathcal{DC}$		
		namcomp	*cname*	$\mathcal{DC}$		
customer	*cname*	$\mathcal{DC}$				

For recursive nesting, the original code works, but gives us all paths. Not to worry: any particular relation is finite. Thus the nested bill of materials in section 3

```
assembly
(component  subassembly                                          )
            (qty  component  subassembly                          )
                             (qty  component  subassembly         )
                                              (qty  component)
wallplug       1  cover         1  plate
                                2  screw
               1  fixture       2  plug          1  mould
                                                 2  connector
                                2  screw
```

will give

```
schema
(attr         schema                                  )
              (attr          schema                   )
                             (attr          schema)
                                            (attr)
component
subassembly   qty
              component
              subassembly    qty
                             component
                             subassembly    qty
                                            component
```

It would be nice to shorten this result to

```
schema
(attr         schema)
              (attr)
component
subassembly   qty
              component
              subassembly
```

and reveal the basic two relations,

assembly(*component,subassembly*), and
subassembly(*qty,component,subassembly*).

We can start by stopping the recursion when

([*attr*] **in** *schema*) = *schema*/*attr*

We can do this by introducing a terminating condition into the recursion.

```
let attr be self;
let schema
        target ([attr] in schema) = schema/attr
    be transpose(attr) union [attr,schema] in .;
```

This stops, in our example, after the second iteration, with

```
schema
(attr          schema                    )
               (attr          schema)
                              (attr)
component
subassembly    qty
               component
               subassembly    qty
                              component
```

and we are left with removing the second level of nesting.

If this were a LISP problem, we could define a function, *butlast*, and supporting functions, to find all but the last element of a list:

```
butlast(schema) is copybutlast(car(schema),cdr(schema));
copybutlast(s1,s2) is if null(cdr(s2)) then s1 else
        copybutlast(absorb(s1,car(s2)), cdr(s2));
absorb(s1,s2) is if null(s1) then s2 else
        cons(car(s1),absorb(cdr(s1),s2));
```

So all we have to do is to define the basic LISP functions, *car, cdr, cons*, and *null* for recursively nested relations. In the special case of a recursive relation with a single non-nested and a single recursively nested attribute, which we have in *schema*, these are

```
car(schema) is
        [red union of (attr ijoin [attr′] in schema)] in schema
        given {let attr′ be attr};
cdr(schema) is schema/schema;
cons(schema,x) is [attr,schema] in schema
        given {let attr be attr′;
                let schema be (attr in equiv union of relation(attr′) by attr)
                        join if [] in x then x else true
        };
null(x) is not [] in x;
```

In this mapping from recursively nested relations to a special case of LISP, an atom is a flat relation, a list is a recursively nested relation, and `NIL` is an empty relation. (Or $\mathcal{DC}$. `NIL` satisfies **not** [] **in** `NIL`, and so **false** is [] **in** `NIL`, pretty close to LISP's identification of `NIL` with **false**.) LISP is more general than this sketch because its lists can bifurcate. If we were to generalize to arbitrary recursively nested relations, we would have a structure of potentially any fanout. Since it is clear that much work would be needed to make this kind of code efficient for large structures on secondary storage, and since it is at present unclear if the enterprise would eventually be useful, we do not pursue it. We also do not take space to make comments on the above definitions, save for two. The **given** construct follows an expression with statements needed to give values to components of the expression. Note in the last line of the definition of *cons* that a relation can be **join**ed with **true**: **true** is the nullary projection of a non-empty relation, and is the identity of **join** (**join**ing with **false**, the nullary projection of an empty relation, on the other hand, gives an empty relation (which is not **false**) on the attributes of the other operand).

As well as finding the whole schema, as we have spent this section discussing, we might want to know partial schemas. We introduce a variant of **transpose** in the relational algebra (**transpose** is a domain algebra operator): **attribsOf** returns a relation on a single attribute of type **attrib** whose name is given as a parameter, which relation contains all the attributes of the operand, in this case whether scalar or not. Thus

attribsOf(*attr*) *company*

gives

attr
`cname`
`address`
`customer`

We can take the next step, having found the attributes of *company*, and ask about the attributes of *address* within *company*

$\textbf{attribsOf}(\mathit{attr})\ \mathit{company/address}$

gives

attr
`street`
`city`
`codezip`

and so on, like expanding a tree compressed on a display screen click by click.

(The difference, that **attribsOf** provides all attributes while **transpose** provides only scalar attributes, leads to an easy implementation to discover whether a relation is flat or not.

$\mathit{flat}(r)$ **is** ($\textbf{attribsOf}(\mathit{attr})\ r = [\textbf{red union of transpose}(\mathit{attr})]$ **in** r;

By the way, with our above definition that atoms are flat relations, $\mathit{atom}(r) = \mathit{flat}(r)$.)

7 Marked Up Text

We have seen a number of advantages for data representation and coding in the foregoing, but we have not always gained by introducing semistructured data. For example, the semistructured representation of both `wallplug` and `fixture` in section 4 is forced into contortions to accomplish the same result as the simple flat relation shown in section 1.1.1. A significant advantage of the semistructured approach appears when we embed it in *text* data to give *marked-up text.* Before we investigate marked-up text, we must say some things about plain text.

7.1 Text

Text seems to be very different from relations, having no intrinsic repetitions which would give rise to tuples, and depending absolutely on the order of its elements. However, if we capture the order by sequencing attributes, text can indeed seem to be a set of tuples. We could suppose that text has implicit attributes giving characters and their sequence, or words and their sequence, or sentences and their sequence, and so on. Thus, a text such as

> Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James and Jane had Tom in 1961 and Sue in 1962).

might be seen as the following relation, where we temporarily introduce attributes whose names begin with "." and might be system-generated.

(*.char*	*.charseq*	*.word*	*.wordseq*	*.sent*	*.sentsq*	*.para*	*.paraseq*)
`T`	`1`	`Ted`	`1`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`e`	`2`	`Ted`	`1`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`d`	`3`	`Ted`	`1`	`Ted..1932`	`1`	`Ted..1962).`	`1`
	`4`		`1`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`m`	`5`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`a`	`6`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`r`	`7`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`r`	`8`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`i`	`9`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`e`	`10`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
`d`	`11`	`married`	`2`	`Ted..1932`	`1`	`Ted..1962).`	`1`
:	:	:	:	:	:	:	:
`1`	`198`	`1962`	`49`	`Their..1962)`	`2`	`Ted..1962).`	`1`
`9`	`199`	`1962`	`49`	`Their..1962)`	`2`	`Ted..1962).`	`1`
`6`	`200`	`1962`	`49`	`Their..1962)`	`2`	`Ted..1962).`	`1`
`2`	`201`	`1962`	`49`	`Their..1962)`	`2`	`Ted..1962).`	`1`
`)`	`202`	`)`	`50`	`Their..1962)`	`2`	`Ted..1962).`	`1`
`.`	`203`	`.`	`51`	`Their..1962)`	`2`	`Ted..1962).`	`1`

This may seem wasteful, but that is not an issue. The text is stored as a sequence of ASCII (or other encoding) characters as usual, not as an ungainly relation all spelled out as above. An expansion such as the above would probably never be created, but projections and selections and other relational operators on these attributes are now available. Whether they are used effectively or clumsily is up to the programmer.

We can reinforce this point, and also gain flexibility, if we have an *attribute generator* instead of the collection of system-generated new names (*.char*, etc.) all needing to be remembered by the programmer. We can call such an attribute generator **text2attr** and use it as an operator on text which creates a relation (or, in the domain algebra, as an operator on text attributes which creates a nested relation). **Text2attr** takes parameters of types *pattern, string*, and *integer* (or some more specialized form of sequencing type). The pattern is used to define the element, and the programmer can use the string and integer to name the new attributes. (Either, but not both, of these two is optional, in case only the element or only the sequence is wanted.) Here are some simple definitions for character, word, and sentence elements.

text2attr(`"."`, *char*, *charseq*)
text2attr(`"\w"`, *word*, *wordseq*)
text2attr(`"(.|\n)*?(\.|\,)"`, *sent*, *sentseq*)

where \w stands for any word and \n meansnew line. (apart from ' and _), which we saw above is treated as words, and ^ and $ mean start and end of lines as usual.

For elements that are hierarchically related, **text2attr** could take a sequence of these triples/pairs of parameters, and could thus generate all of the above relation if needed.

text2attr(`"."`, *char*, *charseq*, `"\w"`, *word*, *wordseq*,
`"(.|\n)*?(\.|\,)"`, *sent*, *sentseq*)

We can also join two texts, using a *grep join* which creates a relation linking them. Like **text2attr**, the grep join supplies a pattern and names attributes for the resulting relation, which the grep join associates with the proper values according to the types of the attributes. We illustrate with the **igrep** join, where the **i** means that an intersection is being done, and also serves to distinguish this binary operator from the unary **grep** of section 1.1.3.

Given two texts, *Jtext*,

On his way to work,
Joe met Sue. "Let's
go out tonight", he invited
her. After work, he met her

at her apartment and they
went to a movie
which he enjoyed a lot.

and *Stext*,

Sitting in a movie
with Joe, Sue
wondered why she had accepted his
invitation. She had just
started to paint her apartment
and did not really have time.

the result of

JStextlink <− *Jtext* **igrep**("\s*",*pos1*,*val1*,*val2*,*pos2*) *Stext*;

is

JStextlink

(*pos1*	*val1*	*val2*	*pos2*)
4	his	his	64
12	to	to	101
21	Joe	Joe	25
29	Sue	Sue	30
61	invit	invit	68
69	her	her	110
93	her	her	110
93	her apartment	her apartment	110
97	apartment	apartment	114
114	and	and	124
128	to	to	101
131	a movie	a movie	12
156	a	a	12

We see first that one of *val1* or *val2* could have been omitted, since both are necessarily the same. Such omission is allowed in the attribute list for **igrep**: the positions of the parameters within their types determine the source of the value. Thus, omitting one of the **string** parameters, *val1* or *val2*, would leave only one **string** parameter, which is taken to be the value from the left-hand operand. The two **integer** parameters, *pos1* and *pos2*, need not be together; the first will be the sequence number (always by character position in the text) from the left-hand operand, and the second from the right.

Second, we note that the pattern, "\s*", is a *start* delimiter, saying that the comparisons start only after whitespace; that is to say, at the beginnings of words.

It may be more informative to remove noise words (by a conventional relational join) such as `a, and, her, his,` and `to`, giving the more compact result

(*pos1*	*val1*	*val2*	*pos2*)
21	Joe	Joe	25
29	Sue	Sue	30
61	invit	invit	68
100	her apartment	her apartment	114
131	a movie	a movie	12

Let's consider a symmetric difference variant of the grep join, **sgrep**, applied to units of lines (instead of words) in Sue's story, above, and the following minor reformatting of Sue's story.

Stext′

> Sitting in a movie
> with Joe, Sue wondered
> why she had accepted his
> invitation. She had just
> started to paint her apartment
> and did not really have time.

$$SSdiff <\!\!- \; Stext \; \textbf{sgrep}(\texttt{"\^{}"}, \; pos1, val1, val2, pos2) \; Stext';$$

gives

SSdiff

(*pos1*	*val1*	*val2*	*pos2*)
20	`with Joe, Sue`	$\mathcal{DC}$	$\mathcal{DC}$
34	`wondered why she had accepted his`	$\mathcal{DC}$	$\mathcal{DC}$
$\mathcal{DC}$	$\mathcal{DC}$	`with Joe, Sue wondered`	20
$\mathcal{DC}$	$\mathcal{DC}$	`why she had accepted his`	43

Apart from using character positions instead of line numbers, this result is identical to the Unix `diff` operator on these two texts:

```
2,3c2,3
< with Joe, Sue
< wondered why she had accepted his
---
> with Joe, Sue wondered
> why she had accepted his
```

We also have **ugrep**, analogous to set union, **dgrep**, inspired by set difference, and some other useful variants of binary grep, each with granularity controlled by regular expression patterns.

7.2 Marked-Up Text

The unary **grep** operator of section 1.1.3, **text2attr**, and the binary **grep** operators all allow us to generate attributes from text. Marking up the text with *x*ML tags specifies more elaborate attributes. For example, marking up

> Ted married Alice in 1932. Their children, Mary (1934) married Alex in 1954 (Joe was born to Mary and Alex in 1956) and James (1935) married Jane in 1960 (James and Jane had Tom in 1961 and Sue in 1962).

to

```
<Person>
  <Name>Ted</Name> married
  <Family><Spouse>Alice</Spouse> in <Married>1932</Married>. Their children,
    <Children><Name>Mary</Name> (<DoB>1934</DoB>) married
      <Family><Spouse>Alex</Spouse> in <Married>1954</Married>
        (<Children><Name>Joe</Name> was born to Mary and Alex in <DoB>1956</DoB>
        </Children>)
      </Family>
      and <Name>James</Name> (<DoB>1935</DoB>) married
      <Family><Spouse>Jane</Spouse> in <Married>1960</Married>
```

```
        (<Children>James and Jane had <Name>Tom</Name> in
         <DoB>1961</DoB> and <Name>Sue</Name> in <DoB>1962</DoB>
        </Children>).
      </Family>
    </Children>
  </Family>
</Person>
```

gives a source not only of the original text, but also of a nested relation

(*Name*	*Family*									)
	(*Spouse*	*Married*	*Children*							)
			(*DoB*	*Name*	*Family*					)
					(*Spouse*	*Married*	*Children*			)
							(*DoB*	*Name*	*Family*	)
Ted	Alice	1932	1934	Mary	Alex	1954	1956	Joe	$\mathcal{DC}$	
			1935	James	Jane	1960	1961	Tom	$\mathcal{DC}$	
							1962	Sue	$\mathcal{DC}$	

While we might expect this nested relation to be generated automatically, as semistructured data, from the marked-up text, to generate a nested relation *including* the text requires new attribute names, and hence an attribute generator, **mu2nest**. This takes as parameters a pattern which indicates what is to be excluded from the text, and attributes for the *content* (the text itself), the *start* position, and the *length* of the text. Of the last two, *start* retains sequencing information for the tags, which would be lost on removing the tags, and *length* is redundant (if there is a **length** operator or function) but useful.

Here is the result of executing

nestPerson <− **mu2nest**("<.*>",*content,start,length*) *Person*;

nestPerson							
(*content*	*start*	*length*	*Name*			*Family*	)
			(*content*	*start*	*length*)		
Ted .. 1962).	1	203	Ted	1	3	<Family><Spouse>Alice .. </Family>	

We see that **mu2nest** extracts the specified attributes at the top level and for unstructured attributes one level down. Since *Family* recursively contains marked-up text, **mu2nest** does not descend into *Family.* Before we consider recursive application of **mu2nest**, we look at special cases.

mu2nest("<.*>",*content,,length*) *Person*;

omits all *start* attributes.

nestPerson <− **mu2nest**("<.*>",*content*) *Person*;

does not need to expand *Name* as above, since *Name* has no further attributes, but for consistency we generate the unary attribute for *Name.*

nestPerson			
(*content*	*Name*	*Family*	)
	(*content*)		
Ted .. 1962).	Ted	<Family><Spouse>Alice .. </Family>	

To go deeper, we must apply **mu2nest** to *Family.*

let *nestFamily* **be mu2nest**("<.*>",*content,start,length*) *Family*;

We will not be able to show this virtual attibute unless we abbreviate all the names, so in the following result, *Person* becomes P, *nestPerson* becomes P', *content* becomes c, and so on in a self-explanatory way. To help us follow the result more clearly, the c, s and l attributes are prefixed by the attributes (P, N, etc.) they are generated for.

P' <− **mu2nest**("<.*>",c, s, l) P;

let F' **be mu2nest**("<.*>",c, s, l) F;

```
P'
(P.c        N              F                     )  F'
P.s   P.l   (N.c           )                        (F.c       S              M               C              )
            N.s    N.l                              F.s   F.l  (S.c        )  (M.c        )
                                                               S.s   S.l      M.s   M.l
Ted .. ).   Ted            <F><S>Alice ..  </F>     Alice .. ). Alice          1932            <N>Mary ..  ).
1     203   1      3                                13    191  13    5        22    4
```

(We show the *.s* and *.l* attributes underneath the *.c* attribute to reduce the width further, so each data tuple is on two lines.)

To expand *Person* at all levels, we need recursion. Since this example nests *Children* inside *Family*, and *Family* inside *Children*, we need two mutually recursive attributes. Using the abbreviations,

let F' **be** $[F.c, F.s, F.l, S, M, C']$ **in** **mu2nest**(`"<.*>"`,$.c, .s, .l$) F;
let C' **be** $[C.c, C.s, C.l, N, D, F']$ **in** **mu2nest**(`"<.*>"`,$.c, .s, .l$) C;
$P' <- [P.c, P.s, P.l, N, F']$ **in** **mu2nest**(`"<.*>"`,$.c, .s, .l$) P;

gives

```
P'
(P.c        N               F'                                                                                    ..)
P.s   P.l   (N.c        )   (F.c        )  S            M             C'                                          
            N.s   N.l       F.s   F.l      (S.c      )  (M.c      )   (C.c        )  N               D         F'  ..
                                           S.s   S.l    M.s   M.l     C.s   C.l      (N.c       )  (D.c      )   ..
                                                                                     N.s   N.l     D.s   D.l  ..
Ted .. ).   Ted             Alice .. ).    Alice        1932          <N>Mary .. ).  Mary          1934       ..
1     203   1     3         13    191      13    5      22    4       44    201      44    4       50    4    ..
                                                                                     James         1935       ..
                                                                                     121   5       128   4    ..
```

(The whole nested relation is too wide to show, but a synopsis, without the *content*s fields for the relation and relational attributes, is

```
Person
(Name   Family                                                          )
        (Spouse  Married  Children                                      )
                          (DoB    Name    Family                        )
                                          (Spouse  Married  Children    )
                                                            (DoB    Name  )
Ted     Alice    1932     1934    Mary    Alex     1954     1956    Joe
1,3     13,5     22,4     50,4    44,4    64,4     72,4     111,4   78,3
                          1935    James   Jane     1960     1961    Tom
                          128,4   121,5   142,4    150,4    182,4   175,3
                                                            1962    Sue
                                                            198,4   191,3
```

.)

It may be useful to have predefined patterns for special formats, such as \html, \latex, \ps, and \pdf, to save programming in extracting text.

mu2nest(`"\latex"`,*.content*) *Person*

Once we have a text, we can apply to it all the operations of section 7.1. For instance, we might want to know the sequence numbers of the words that are the names of Ted's and Alice's children.

let *words* **be** **text2attr**(`"(^|\s*.*\s*|$)|\p"`, *Name, Pos*) *Person.content*;
let *childpos* **be** *words* **join** [*Name*] **in** *Family*/*Children*;
nestPerson/*childpos*

giving

(*Name*	*Pos*)
Mary	44
Mary	94
James	121
James	156

(Note that the tags are in fact only on words 44 and 121: separating the tags from the text loses that information.)

In section 7.1, above, we linked two apparently related texts, considered as pure texts. With markup tags, we can control such links explicitly. Here are Joe's and Sue's stories marked up.

```
On his way to <B>work</B>,
<C>Joe</C> met <C>Sue</C>. "Let's
go out tonight", he <A>invit</A>ed
her. After work, he <A>met</A> her
at <B>her apartment</B> and they
went to a <A><B>movie</A></B>
which he enjoyed a lot.
```

```
Sitting in a <A><B>movie</A></B>
with <C>Joe</C>, <C>Sue</C>
wondered why she had accepted his
<A>invit</A>ation. She had just
started to paint <B>her apartment</B>
and did not really have time.
```

The three categories of tag, `A`, `B`, and `C`, in each story correspond to three attributes. Three attributes are not needed for the linking exercise we wish to show, but they illustrate some interesting points. First, note that `movie` is a value for two different attributes in each story, and, to avoid repeating the word, which would not make sense if we were to extract the text, we violate the usual prescription that tags be strictly nested. Nested tags now have the special meaning that they generate nested relations: we are happy with having flat relations in this example.

Second, we need a convention for extracting the flat relation from the text as marked up with these attribute tags. Specifically, the problem is where will tuples begin and end? We have not provided a nesting <`.tuple`> tag to specify this. The convention is simple: run through the tags, and as soon as any one repeats, start a new tuple. Thus, the relations we obtain, using **mu2nest**("`<.*>`",*content*) are

```
(A      B              C         ) | (A      B              C     )
        work           Joe   movie | movie  Joe
invit                  Sue         | invit                  Sue
met     her apartment              |        her apartment
movie   movie
```

8 Data on the Web

The World Wide Web is structured by the HyperText Markup Language, HTML, a descendent of SGML, and a variant of xML. The Web also includes other documents, such as plain text, PostScript files, Graphical Interchange Format files, etc., but these all appear as leaves in the Web, since they do not contain further links. Only HTML documents include the links from which the Web draws its name.

An HTML document contains semistructured data, with attributes given by xML tags; it also contains text, and so fits into the category of marked up text discussed in the previous section. The present section is thus an application of the above material. There is one additional consideration, which arises from HTML (and shared by XML and other languages in the family): tags may have "*attributes*", illustrated by `href`, `name`, `src`, and `width`, in the example below. These are leaves in the nested representation, having no contained tags, and considered identical to other leaf attributes.

The advance made by the relational model of data over earlier approaches is that it does not distinguish between data in RAM and data on secondary storage: all relations are conceptually "inside" the machine, whether they are in RAM or on disk or other secondary storage. This saves the programmer from explicit input/output operations. In our representation of the HTML Web, we similarly ignore any distinctions between local pages and pages which require remote access via the Internet. We will consider the whole accessible web to be contained in any page to which we happen to have access, thus eliminating explicit page fetches from the programmer's concerns. Such an abstraction is almost accomplished by the usual browser interfaces to the Web, but the operation of clicking on an anchor is still an explicit page fetch.

To accomplish this, we must view the anchor tag, <A>, of HTML, as a nested HTML document, including nested *Head* and *Body* attributes where appropriate, as well as attributes, contained tags, and contents. As an example, here is a two-document web, based on the text version of Ted's family tree.

```
<HTML> <!file famtree.html>
<Head><Title>Family Tree</Title></Head>
<Body>
<A href=bioTed.html>Ted</A> <A href=#TedAliceWedding>married</A>
Alice in 1932. Their children, Mary (1934) married Alex in 1954
(Joe was born to Mary and Alex in 1956) and James (1935) married
Jane in 1960 (James and Jane had Tom in 1961 and Sue in 1962).
<A name=TedAliceWedding> <br> <img src=Ted_Alice.jpg width=400> <br>
Ted and Alice Just Married, 1932 </A>
</Body>
</HTML>

<HTML> <!file biotext.html>
<Head><Title>Ted's Biography</Title></Head>
<Body>
Ted was born at McGill University and worked there until retiring.
</Body>
</HTML>
```

The nested relational representation of this small web, assuming that attributes suffixed *.content* have been created by applying **mu2nest**, is

HTML									
(Head	*Body*								)
(Title)	*(Body.content*	*A*							)
		(A.content	*href*	*Head*	*Body*	*name*	*img*		)
				(Title)			*(src*	*width)*	
Family Tree	Ted married..	Ted	bioTed.html	Ted's Biography	Ted was..				
		married	#TedAliceWedding						
		Ted and..				TedAliceWedding	Ted_Alice.jpg	400	
Head	*Body*								
(Title)									
Ted's Biography	Ted was..								

Since *img* does not need to be nested inside *A*, we also have the option of recording *img* as a direct attribute of *Body*. Thus, a schema which might be generated from this web is

relation *HTML*(*Head,Body*);
domain *Head*(*Title*);
domain *Body* **strg**|(*Body.content,A,img*);
domain *A*(*A.content,href,Head,Body,name,img*);
domain *img*(*src,width*);

Note first that `Ted's Biography` occurs twice, once as a part of `Family Tree`, and once as a web page in its own right. This would only happen if both pages were independently loaded into the nested relation, so we will ignore the second entry from now on.

The second thing to note is more important. *A* contains, optionally, the recursive attribute *Body* (along with *Head*), thus capturing the relational idea that the whole accessible web is present, even though it may not have actually been retrieved.

It is useful to be able to make distinctions among remote, local, and internal anchor hrefs. The above example illustrates the latter two, and remote hrefs will be familiar to all readers. These distinctions may allow programs to avoid expensive parts of a recursion, namely those that require access to remote web pages, if Internet connection is found to be expensive, or even to local ones, should even a disk access seem to cost too much. The distinctions can be made by examining the structure of the *href* data. Internal references start with the `#` symbol:

let *internal* **be grep** `"^#"` **in** *href* **in** *A*;

Local references must not start with "/":

let *local* **be grep** `"^[^/]"` **in** *href* **in** *A*;

Finally, remote references are neither internal nor local:

let *remote* **be where** (**not** []**in grep** `"^#"` **in** *href*) **and**
not []**in grep** `"^[^/]"` **in** *href* **in** *A*;

These virtual attributes may be used in path expressions just as if they were original attributes. For example, to find only internal references, write

HTML/Body/internal/href

(instead of

HTML/Body/A/href)

The above is a little simplistic. A more thorough specification of internal and local references would include comparisons of *href* with the URL of the page itself. This requires an additional attribute in *HTML*, which should also be called *href*, to hold these URLs, say `famTree.html` and, of course, `bioTed.html`. Comparisons could be from the upper level, say between *href* and *Body/A/href*, but this is awkward for defining the virtual attribute at the lower level. If the upper attribute had a different name, say *url*, we can suppose this name is visible from the lower level, just as in programming language scope blocks: we could compare *href* at the lower level directly with *url* from above. However, such a solution could not recurse. So we adopt a scoping convention from operating system file hierarchies, instead of from programming languages, and allow a ".." path element to make visible any higher name which has been redefined at a lower level. Thus, we compare *href* with *../../href*. In fact, since the higher *href* is visible and unambiguous only one level up (*Body* does not redefine *href* as one of its own attributes, and so the *href* in *HTML* is also a (constant) attribute on *Body*) we can get away in this example with comparing *href* with *../href*.

So to specify the alternative definition that a local *href* must be contained in the parent *href* as a prefix, we say

let *local* **be grep** `"^"`**cat** *../href* **in** *href* **in** *A*;

and this can be put together as an alternative to the previous definition

let *local* **be** (**grep** `"^[^/]"` **in** *href* **in** *A*)
    union grep `"^"`**cat** *../href* **in** *href* **in** *A*;

The type or format of a document (e.g., `html, ps, pdf, jpg, ftp`) can also be learned from its URL:

let *format′* **be** x **in grep**(;x) `"[^|\.]\x[^/\.][$|#]"` **in** *href*;

where "`[^|\.]`" specifies: from the start of the string or from a "."; "`[^/\.]`" specifies: no "/" or "." may occur after x; and "`[$|#]`" specifies: to the end of the string or to "#".

let *format* **be if** *format′*=`""` **then** `"html"` **else** *format′*;

says that an internal reference is always in HTML format.

This format, and other properties, can be gleaned alternatively from the HTTP headers, such as *Content-Type* (for format), *Content-Length, Date, Last-Modified* and so on. These could all be considered further attributes of *HTML* or of *A* and so accessible in the resulting nested relation.

9 Acknowledgements

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. Zhan Yu, Fan Guo, Yu Gu and Jiantao Xie have implemented the recursive domain algebra and the regular expression syntactic sugar, the metadata operators, the grep operators, and the text and marked-up text operators in the research system, relix.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : from Relations to Semistructured Data and XML.* Morgan Kaufmann, San Francisco, 2000.

[2] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *Internat. J. on Digital Libraries*, 1(1):5–19, 1997.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. Technical report, Stanford U., Palo Alto CA, 1997.

[4] G. E. Blake, M. P. Consens, P. Kilpelainen, P.-A. Larson, and F. Tompa. Text/relational database management systems: Harmonizing SQL and SGML. In *Proc. of the ADB'94 Conf.*, ., 1994.

[5] O. P. Buneman. Semistructured data. In *Proc. PODS*, ., 1996.

[6] O. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In H. V. Jagadish, editor, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 505–16, Montreal, Canada, June 1996. ACM.

[7] O. P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Workshop on Database Programming Languages*, ., 1995.

[8] D. D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[9] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–87, June 1970.

[10] E. F. Codd. Further normalization of the data base relational model. In R. Rustin, editor, *Data Base Systems*, pages 34–64. Prentice-Hall, Engelwood Cliffs, N. J., 1972. Courant Institute of Mathematical Sciences, New York University, 1971/5/24–25.

[11] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, Engelwood Cliffs, N. J., 1972. Courant Institute of Mathematical Sciences, New York University, 1971/5/24–25.

[12] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real-life recursion. In *Proc. of the ACM Symp. on Principles of Database Systems, PODS'90*, pages 404–16, ., 1990.

[13] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In U. Dayal and I. L. Traiger, editors, *Proc. of the ACM Internat. Conf. on Management of Data, SIGMOD'87*, pages 323–30, San Francisco, May 27–9 1987. ACM Press.

[14] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G+: Recursive queries without recursion. In L. Kershberg, editor, *Proc. of the Second Internat. Conf. on Expert Database Systems*, pages 355–68, Tysons Corner, Va., April 25–7 1988.

[15] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the web. In *Proc. of the Sixth Internat. Workshop on Research Issues in Data Engineering, RIDE'96*, New Orleans, Feb. 1996.

[16] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1997. (Also *Proc. PDIS*, 1996).

[17] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations for semistructured, hierarchical data. In Alex Gray and Per-Åke Larson, editors, *Proc. of the 13th Internat. Conf. on Data Engineering*, pages 79–90, Birmingham, U.K., April 7–11 1997. IEEE Computer Society.

[18] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc of the Internat. Conf. on Very Large Databases, VLDB?*, ., 1995?

[19] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, and J. Zobel. Atlas: A nested relational database system for text applications. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):454–70, 1995.

[20] D. Suciu. An overview of semistructured data. *SIGACT News*, 29(4):28–38, Dec. 1998.

[21] J. Thierry-Mieg and R. Durbin. acedb—a C.elegans database: Syntactic definitions for the ACeDB data base manager. www.acedb.org/Cornell/syntax.html, Dec. 1992.

[22] Fred Wobus. Aql—moviedb example queries. www.acedb.org/Software/whelp/AQL/MovieDB/queries.shtml, Dec. 1999.

A Semistructured Queries from Classical Papers

In the body of this paper, we have developed solutions for semistructured and marked-up data and querying as a logical result of extending relational nesting to completeness. In doing the work, we have depended for guidance on a number of papers in the literature from the mid-'80s on, and these are discussed here. This appendix works through query examples from the literature in chronological order: G+ and Graphlog, ACeDB, Atlas, SGML relations, Lorel, UnQL, OQL-doc, and the web query languages Weblog, WebSQL, and XML-QL.

A.1 G, G+ and Graphlog

The graphical query language, G, appeared [13] in 1987. Since a second paper, extending G to G+, gives simpler examples, we look there first, then return.

Saying "the extension of query languages to handle problems not solvable in relational algebra is an area of much current interest", Alberto Mendelzon's group at the University of Toronto published a paper in 1988 [14] which expressed transitive queries on suitable relations as Horn clauses with regular expressions. They expressed both data and queries as graphs, but it is possible to be more linear.

Parent

(*Sr*	*Rel*	*Jr*)
Don	F	Sue
Liz	M	Sue
Wil	F	Bob
Wil	F	Ted
Sue	M	Bob
Sue	M	Ted
Ted	F	Pam
Ann	M	Pam

1. Sue's children $(\texttt{Sue, M}, x)$
2. People with common ancestor $(x, y) \leftarrow (z, \texttt{[M|F]+}, x)$
   $(x, y) \leftarrow (z, \texttt{[M|F]+}, y)$
3. Ancestor or descendent $(x, y) \leftarrow (x, \texttt{[M|F]+}, y)$
   $(x, y) \leftarrow (y, \texttt{[M|F]+}, x)$

These first examples of the paper pertain to finding an arbitrary regular expression on the *Rel* attribute in a generalized closure of the relation. Brute-force evaluation can do just this: take the closure recursively, concatenate the values of *Rel* from each level together, and use **grep** to select the desired result. But since the paper is on *recursive queries without recursion*, and full closures require a lot of computing, we should do better. As in section 7, however, it is not clear that there is a nested version better than the flat relation above, even though we might be able to make the syntactic sugar quite compact and resembling the G+ queries.

The second set of examples in the paper look at summary or aggregation operators on numerical attributes. Example data is not given, so we use *BoM* from section 1.1 and a modification of data from [13] which includes distances between airports.

Flights

(*From*	*Line*	*Dist*	*To*)
Tor	AC	4200	Van
Tor	AC	880	Bos
Tor	AC	690	NY
Van	AC	4200	Tor
LA	AC	4700	Tor
Tor	AA	880	Bos
Bos	AA	380	NY
NY	AA	5000	LA
LA	AA	1000	SF
SF	AA	5200	NY

To find all components to assemble a wallplug and how many of each are needed, G+ writes the graphical equivalent of (`wallplug`,(×,+),x) while we find *BoMtc* from section 1.2 and then select

[*subassembly, qty*] **where** *assembly*=`"wallplug"` **in** *BoMtc*

To find the shortest distance between Toronto and San Francisco, G+ writes (`Tor`,(+,**min**),`SF`). We find the closure of the projection, [*From,Dist,To*] **in** *Flights*, using the same code as for *BoMtc* except that + is replaced by **min** and × is replaced by +, then

[*Dist*] **where** *From*=`"Tor"` **and** *To*=`"SF"` **in** *FlightsTC*

To find the city which is the longest distance (but not necessarily on the longest path) from Toronto,

let *maxDist* **be equiv max of** *Dist* **by** *From*;
[*maxDist,To*] **where** *From*=`"Tor"` **in** *FlightsTC*

or, since this gives `8400` from `Tor` to `Tor`,

[*maxDist,To*] **where** *From*=`"Tor"` **in where** *From*≠*To* **in** *FlightsTC*

(a rare example of the usually redundant double selection **in where**). This code appears to diverge from the G+ formulation, which involves (+,**max**) operators and must use special syntax to collapse multiple valuations into one. Note that we cannot replace **min** of the previous query by **max** in the closure computation, when the graph has cycles, because each traversal of a cycle increases the aggregate, and the computation will not stop.

(To find the city which is on the longest path from Toronto, we use the domain algebra non-recursively, but with an apparent cycle, to count levels while finding the closure of the projection [*From,To*] **in** *Flights*:

let *level* **be** 1;
FTFlights <− [*From,To,level*] **in** *Flights*;
let *level′* **be** *level* + 1;
let *level″* **be equiv min of** *level′* **by** *From, To′*;
let *level* **be** *level″*;
FTFlightsTC **is** [*From,To,level*] **in** [*From,To,level″*] **in**
    (*FTFlights* [*From,To,level* **union** *From,To,level′*]
    [*From,To′,level′*] **in**
    ([*From,To′,level′*] **in** (*FTFlights*[*To* **join** *From′*]
    [*From′,To′,level′*] **in** *FTFlightsTC*)));
let *maxLev* **be equiv max of** *level* **by** *From*;
[*maxLev,To*] **where** *From*=`"Tor"` **and** *maxLev*=*level* **in** *FTFlightsTC*;

To find the number of "edges in the graph" is to find the number of tuples:

let *edgeCt* **be red** + **of** 1;
[*edgeCt*] **in** *Flights*

does not need the closure.

Finally, a "same-generation" query can use the same level-counting technique on *Parent* that we used on *FTFlights* to find the longest path.

let *generation* **be** 0;
genParent <− [*Sr,Jr,generation*] **in** *Parent*;

```
let generation' be generation + 1;
let generation be generation';
genParentTC is genParent union [Sr,Jr,generation] in [Sr,Jr,generation'] in genParentTC;
let Jr' be Jr;
([Jr,generation] in genParentTC) comp [Jr',generation] in genParentTC
```

The earlier paper [13] poses a interesting query on *Flights*, “find first and last cities visited in all round trips from Toronto, in which the first and last flights are with Air Canada and all other flights (if any) are with the same airline”. It answers with the graphical equivalent to the union of x, y and z given

$\{$(`Tor`, `AC`,x),(x,`AC`,`Tor`)$\}$ and
$\{$(`Tor`, `AC`,y),($y, w+, z$),(z,`AC`,`Tor`)$\}$

However, the paper acknowledges a difficulty in implementing this answer, in that only simple paths of the graph are intended to be traversed, while a naive implementation will erroneously include the results of traversing the cycles indefinitely. The implementation must maintain a list of visited nodes. We can build up a closure with the *Line* attribute cumulatively concatenated, on which we can subsequently apply **grep**. In each tuple, we build up a nested relation of the visited nodes (*Visit*), and we connect an edge to a path in the growing closure only if 1) the path is not a loop, with the same endpoints, and 2) the nodes visited by the path do not contain the *From* node of the edge being added. Here is the closure

```
let Node be To;
let Visit be relation(Node);
FLTVFlight <- [From,Line,To,Visit] in Flights;
let Line' be Line;
let Line'' be Line cat Line';
let Line be Line'';
let Visit' be Visit;
let Visit' be Visit union Visit';
let Visit be Visit'';
let From' be From;
let To' be To;
let To be To';
let Start be relation(From);
FLTVFlightTC is FLTVFlight union [From,Line,To,Visit] in
        [From,Line'',To',Visit''] where ((Visit sep Visit')
        and ((Start[From sep Node] Visit'') or (From=To''))) in
        (FLTVFlight [To join From']
        [From',Line',To',Visit'] where From≠To in FLTVFlightTC);
```

(We have used a “sigma join”, **sep**, not discussed in the paper: this tests its two operands for separateness, *i.e.*, non-overlap; it is related to **comp** and **sup** (section 1.1.2).) and here is the final path selection

```
[From,Line,To] where ([] in grep(;w) "AC(w)*AC" in Line) in FLTVFlightTC
```

(Note that we have applied **grep** to a single attribute, considered as a nested relation.)

A later paper [12] extends G+ to Graphlog, which now includes negation. An example on *Parent* is “find descendents of `Don` who are not also descendents of `Ann`”.

```
([Jr] where Sr="Don" in ParentTC) diff [Jr] where Sr="Ann" in ParentTC
```

uses set difference.

In summary, Graphlog and its precursors, although specialized, provide a neater formalism than we do, but do not transcend in expressiveness the relational and domain algebras with recursion. Our implementations, above, have all been in terms of full closures, which could be inefficient: a Graphlog implementation could do better, but so could relational recursion.

A.2 ACeDB

"A C.elegans Database", ACeDB [21], appeared in 1992 to help genome researchers with their bibliographies and data, originally for the genome of the C.elegans nematode. It was subsequently an inspiration for work on semistructured data, so we look at some example queries on a movies database [22]. Here is an extract from the schema (ACeDB calls it the "model") of the movies database suitable for the queries we shall look at.

```
?Movie  Director ?Person XREF Directed UNIQUE Text
        Based_on ?Book XREF Script_for
        Cast ?Person XREF Stars_in UNIQUE Text
        Release Date DateType UNIQUE Text
        Rating UNIQUE Float UNIQUE Int
?Person Real_name UNIQUE Text
        Date_of_Birth UNIQUE DateType UNIQUE Text
        Relations Spouse ?Person UNIQUE DateType UNIQUE DateType
                  Children ?Person
        Directed ?Movie XREF Director
        Stars_in ?Movie XREF Cast
        Wrote ?Book XREF Author
?Book   Reference Publisher UNIQUE Text
                  Year UNIQUE Int
        Author ?Person XREF Wrote
```

As a set of nested relations, this is

relation *Movie*(*Name,Directors,Based_on,Cast,Release,Rating*);
domain *Directors*(*Director,Comments*);
domain *Cast*(*Actor,Role*);
domain *Release*(*Date,Country*);
relation *Person*(*Real_name,Date_of_Birth,Relations,Directed, Stars_in,Wrote*);
domain *Relations*(*Spouses,Children*);
domain *Spouses*(*Spouse,From,To*);
domain *Children*(*Child*);
domain *Directed*(*Film*);
domain *Stars_in*(*Film*);
domain *Wrote*(*Story*);
relation *Book*(*Reference,Authors*);
domain *Reference*(*Publisher,Year*);
domain *Authors*(*Author*);

which would be supplemented by declarations such as

domain *Name* **string**; etc.
domain *Director Person*;

(and so are *Actor, Spouse, Child* and *Author* declared as *Person*)

domain *Based_on Book*;

(and so is *Story* declared as *Book*)

domain *Film Movie*;

There are a number of points to note in this mapping from ACeDB.

- "`UNIQUE`" in ACeDB is a semantic constraint, specifying both singleton sets and functional dependences. Keys and functional dependences are not specified by our formalism in this paper. However, if an attribute is non-singleton (is not specified `UNIQUE` in ACeDB), it must be a nested relation, so we have had to invent plural names beyond the names specified in the ACeDB schema. (A relational entry gives us the "$*$" quantifier, zero or many.)

- Any ACeDB entry is optional, which is to say that the relational version is polymorphic and can omit any attribute. (Any entry thus gives us the "?" quantifier, zero or one.)
- ACeDB also does not specify the names of some fields, just their types, so we have invented names. This will be helpful to the reader, who is not left guessing as to the meanings (as in the Text field following *Director*, which we just called *Comments*, having been given no clue by the ACeDB schema).
- We have simplified the design by giving *Rating* and *Date_of_Birth* single numerical values.
- ACeDB uses "`XREF`" as a further semantic constraint to link mappings between object classes and their inverse mappings in the target class. Such semantics are not provided for our formalism in this paper.
- We have added the *Name* of the movie, which is implicit in the ACeDB example.
- Finally, the ACeDB example omits the ACeDB enumerated type construction, such as

```
Description  Recessive
             Dominant
             Semi-dominant
             Weak
```

  or exclusive enumerated types such as

```
Language UNIQUE French
                English
```

Here are ten queries from [22], selected to be challenging.

4. list all movies and their directors and possibly their real names
   let *Dir* **be** [*Real_name*] **in** *Directors*;
   [*Name,Dir*] **in** *Movie*

5. list authors of books on which movies are based
   [**red union of** [*Authors*] **in** *Based_on*] **in** *Movies*

10. list the movies that have been released after 1990, i.e. in 1991 and thereafter
   [*Name,Release/Date*] **where** *Release/Date*>`1990` **in** *Movie*

11. list all movies for which no release year information is stored
   [*Name*] **where** (**not** [] **in** [*Year*] **in** *Release*) **in** *Movie*

15. list all movies with more than 3 cast members
   [*Name*] **where** ([] **where** (**red** + **of** 1)>3 **in** *Cast*) **in** *Movie*

16. calculate the average age of all the actors
   let *Age* **be** *today* − *Date_of_Birth*;
   Actors <− [**red union of** [**red union of** [*Real_name,Age*] **in** *Actor*]
   in *Cast*] **in** *Movie*;
   let *avgAge* **be** (**red** + **of** *Age*)/(**red** + **of** 1);
   [*avgAge*] **in** *Actors*

20. show release date of first “James Bond” film

```
BondFilms <− [red union of (Name join [Date] in Release)]
        where ([] where Role="James Bond" in Cast) in Movie;
[Date] where Date=red max of Date in BondFilms
```

25. list bond films with the James Bond actor and list all the movies that actor has starred in

```
let Actors be [Actor,Actor/Stars_in/Name]
        where Role="James Bond" in Cast;
let Name′ be Name;
[red union of (Name join [Actor,Name′] in Actors)]
        where ([] where Role="James Bond" in Cast) in Movie
```

28. for all Bond actors list their films where they weren’t James Bond and who they played in those films

```
BondActors <− Cast/Actor where Cast/Role="James Bond" in Movie;
let Actor′ be Actor;
AllActors <− [red union of (Name join [Actor,Role]
        where Role≠"James Bond" in Cast)] in Movie;
AllActors join BondActors
```

29. list actors from Bond-films that didn’t play “James Bond”

```
let Actors be [Actor] where Role≠"James Bond" in Cast;
[red union of Actors] where
        ([] where Role="James Bond" in Cast) in Movie
```

Overall, our formulations, with or without syntactic sugar, are more compact than the ACeDB queries. A particular drawback of the ACeDB approach is that they must resort (from query 20 on) to a secondary, relational, formulation, so an ACeDB programmer is obliged to learn their variant of relational queries as well as the native ACeDB structure.

A.3 Atlas and SGML relations

Also first published in 1992, Atlas [19] went beyond conventional structured access to documents by publication date, author, citation count or date, etc., and offered text processing: retrieval on words, word phrases, and word parts; soundex; stemming; and ranking. Two years later, Tompa’s group at the University of Waterloo proposed [4] a relational interface to the Standard Generalized Markup Language, SGML, to accomplish a similar fusion of text processing and structured databases. We refer to this work as SGMLql for short.

We translate an example Atlas schema into nested relations.

```
Document [
  doc_id INTEGER,
  title TEXT,
  Authors [name (surname TEXT, firstname TEXT)],
  Nodes [node REF Hypertext]
] key = (doc_id)
Hypertext [
  id INTEGER,
  doc REF Document,
  content TEXT,
  Links [
    node REF Hypertext,
    linktype TEXT
  ] key = (node)
] key = (id)
```

As nested relations, this is

relation *Document*(*doc_id,title,Authors,Nodes*);
domain *Authors*(*surname,firstname*);
domain *Nodes*(*node*);
relation *Hypertext*(*id,doc,content,Links*);
domain *Links*(*node,linktype*);

with supporting declarations such as

domain *doc_id* **integer**; etc.
domain *node* **integer**;
domain *id* **integer**;
domain *doc Document*;

Note that we could have declared *node* to be a recursively nested *Hypertext*, but Atlas uses explicit joins, so our nested relations are in keeping with Atlas' approach to the following queries. `REF`erence in Atlas is a tuple comprising the global key of the relation referenced, and this is why Atlas must make the semantic specification of what the key of each relation is.

Note also that Atlas violates the relational model, and consistency between outer and inner relations by not removing duplicates, and by retaining the notion of tuple order for inner relations. It finds it needs to do this to support text, which is a sequence rather than a set. (The Atlas text type is also used for binary data such as images, but Atlas provides no special operators such as clipping or scaling.)

Here are our versions of the last six of the seven queries used to illustrate Atlas. (We have provided our own formulations in three cases where examples are not given or are elided.)

- For each document, list the title, its authors, and the contents of all of its nodes. (N.B. explicit join version.)

  [**red union of** (*title* **join** *Authors* **join** *Nodes*)] **in** *Document*
      [*node* **comp** *id*] [*id,content*] **in** *Hypertext*

- Retrieve all hypertext nodes that have at least three links from other nodes directed to it.

  let *ct* **be equiv** + **of** 1 **by** *id*;
  [*id*] **where** *ct*≥3 **in** [**red union of** (*id* **join** *Links*)] **in** *Hypertext*

- List all documents written by an author with surname McEnroe (case ignored).

  where ˆ(*Authors*/*surname*)=`"MCENROE"` **in** *Document*
      where [] **in** (**grep** `"MCENROE"` **in** ˆ(*Authors*/*surname*)) **in** *Document*

- List hypertext nodes that contain the phrase 'computer science' and the word 'education'. (Note that this query and the next combine to form one query in the Atlas paper.)

  where ([] **in** (**grep** `"computer science"` **in** *content*) **and**
      [] **in** (*grep* `"education"` **in** *content*)) **in** *Hypertext*

- List hypertext nodes that contain the phrases 'computer science' or 'computing science', etc., and the word 'education'. (This illustrates *stemming*, for which Atlas uses special operators but we use a relation, *stems*(*word,stem*), which serves as a dictionary of stems. Note that we use it twice, first to find the stem of 'computer', then to find all other words with this stem.

  let *word′* **be** *word*;
  where ([] **in grep**
      ((**where** *word′*=`"computer"` **in** [*word′*,*stem*] **in** *stems*) **comp** *stems*)
      cat `" science"` **in** *content* **and**
      [] **in** (*grep* `"education"` **in** *content*)) **in** *Hypertext*

- List all documents whose author surname sounds like McEnroe. (Atlas also uses a special operator for soundex queries, but we again use a dictionary, *soundex*(*word,sounds*.)

```
let word′ be word;
((where _word′="mcenroe" in [word′,sounds] in soundex) comp soundex)
        [word join surname]
        [red union of (doc_id join title join Authors join Nodes)]
        in Document
```

- Rank hypertext nodes according to how they satisfy a query on keywords ‘art’, ‘aesthetics’, ‘craft’, or ‘philosophy’. (This is the simplest kind of ranking: *rank* counts the number of different *pos*itions that match the pattern. We have just written out the four patterns inline, instead of constructing a relation containing them.)

```
let rank be equiv + of 1 by id;
[id,rank] in [id,pos] where
        [] in grep(pos) "art" "aesthetics" "craft" "philosophy"
        in content in Hypertext
```

SGMLql embeds structure in text, and allows free text anywhere. Their example might be marked up as follows.

```
<author>W.L. MORTON</author>, <refs><work>The Kingdom of Canada</work>,
<edition>2nd ed.</edition> <paren>(</paren><date>1969</date>)</refs>,
is the fullest one-volume history and the most
traditional<termin>.</termin>.. To understand the place of the colonies
that became Canada in the British Empire, the following are most useful:
<author>H.A. INNIS</author>, <refs><work>The Fur Trade in Canada</work>,
<edition>2nd ed.<\edition> <paren>(</paren><date>1956</date>) and
<work>The Code[sic] Fisheries</work>, <edition>rev.ed.</edition>
<paren>(</paren><date>1954</date>)</refs><termin>;</termin>... The
following works both introduce and analyse the development of the
remaining British colonies to self-governing communities and their union
in confederation. <author>W.S. MacNUTT</author> combines in a single
narrative the histories of the Atlantic provinces in <refs><work>The
Atlantic Provinces, the Emergence of Colonial Society, 1712--1857</work>
<paren>(</paren><date>1965</date>)</refs><termin>.</termin>
<author>FERNAND OUELLET</author> in his <refs><work>Histoire
\'{e}conomique et sociale du Qu\'{e}bec, 1760--1850</work>
<paren>(</paren><date>1966</date>; Eng. trans.  in prep.)</refs>,
applies with great success the demographic method of French
historiography to the little known domestic development of that province
<termin>.</termin>..
```

From this we can extract a nested relational schema.

```
domain biblio(author,refs)
domain refs(work,edition,date)
```

The SGMLql paper goes on to embed these domains in a parent relation

```
relation Encyclopedia(aid,title,cid,req_date,req_wc,due_date,article,biblio)
```

in which article ID, contributor ID, requested date, and requested word count amplify the abbreviated attributes *aid, cid, req_date*, and *req_wc*, respectively. This relation has further nested attributes

```
domain article(title,rest)
domain rest body|xref
domain xref(title)
domain body(chapter,keywords,summary)
domain chapter(paragraph)
```

In the above schema, we have omitted what might be called delimiter attributes, which have fixed or closely prescribed values: *termin* in *biblio* may be ";" or "."; *paren* in *refs* is always "("; and *see* in *xref* is always "see".

The paper offers eight queries, only one of which is so specialized to SGML that we cannot formulate it in a general system. Here are all but the first.

- Who contributed articles for which the proposed titles do not match the titles included in the article's body?
  [*cid,aid*] **where not**(*title* **comp** *article/title*) **in** *Encyclopedia*
- Find proposed titles and lengths of long articles on Canada. (We could have used *body* instead of *, but chose to keep the query the same as the SGMLql query.)
  [*title,req_wc*] **where** (*req_wc*>5000 **and** [] **in grep** `"Canada"`
  in *article/*/keywords*) **in** *Encyclopedia*
- Find all nodes of type "paragraph" and containing substring "Canada".
  where *attr*=`paragraph` **in grep**(*attr*) `"Canada"` **in** *Encyclopedia/**
- Select all nodes having an ancestor of type "chapter".
  Encyclopedia//chapter/**
- Select all nodes that have children.
  Encyclopedia/+
- Select all nodes where the parent has an (SGML) attribute of type status and value "Obs.". (This is specific to SGML, which SGMLql incorporates among its relations, so we can only express it with a similar translator from SGML structure to a relational representation.)
- Find the article identification, title, contibutor, and bibliography for entries where the bibliography has more than one citation but all are from the same author. (This query requires a domain algebra idiom to count the number of different values of an attribute, in this case, *author*, which uses *functional mapping*, not discussed in section 1.2.)
  [*aid,title,cid,biblio*] **where**
  ([] **where** (**red** + **of** 1)>1 **and**
  (**red max of fun** + **of** 1 **order** *author*)=1
  in *biblio*) **in** *Encyclopedia*

Atlas and SGMLql are the first text-querying systems from the literature that we discuss. Since semi-structured data is supposedly half-way between rigidly structured flat relations and unstructured text, the direction taken by these systems is important.

A.4 Lorel

The Lightweight Object REpository, LORE, supports the query Language, LOREL, which was developed for flexibility and to be more forgiving than earlier object-oriented query languages. This made it useful for both The Stanford-IBM Manager of Multiple Information Sources, TSIMMIS, a contemporaneous project on integration of heterogeneous data, and as an early semistructured query language. Since, as we would expect, the two major projects at Stanford generated a great deal of literature, we will focus on two papers [18, 3], which query the now well-known good living guide. ([17] illustrates an interesting soccer league database, but has no queries. It is concerned with schema discovery (we have looked at a simple implementation of this paper's notion of "continuation" ("dataguides" in Lorel) using **attribsOf** at the end of section 6), query optimization, and improving implementation of path expressions.)

Frodo's Guide to Good Living in the Bay Area is presented by [18] as a hierarchical table. As nested relations, we formulate it

relation *Frodos*(*Restaurant*|*Group*)
domain *Restaurant*(*Name,Category,Entree,Location*)
domain *Entree*(*Name,Price*)
domain *Location*(*Street,City,Phone*)
domain *Group*(*Name,Category,Performance,TicketPrice,Location*)
domain *Performances*(*Performance*)
domain *Performance*(*Date, Work*)
domain *Work* **string**|(*Title,Composer*)
domain *Composer*(*Name*)
domain *TicketPrice*(*AgeGroup,Price*)

Note that, as in section A.2, the relational formulation must distinguish multiple from singleton entries by adding plural names.

All queries from [18] can be expressed.

- Find names of all opera groups. (Note that, as usual, we have written an expression, not a statement. A name such as *Answer* is not automatically generated for the result. If a name is needed, it must be supplied by embedding the expression into an assignment or view statement—which is what most programming languages require.)

  Group/Name **where** *Group/Category*=`"Opera"` **in** *Frodos*

- Find names of all opera groups. (In this variant of the first query, we seek the whole group, producing a structured (nested) result.)

  Group **where** *Group/Category*=`"Opera"` **in** *Frodos*

- Find titles of all performances where the title is the same as one of the composers. (Here we do a natural composition on a singleton relation with attribute *Name* (generated by default from the scalar attribute *Name*) and *Composer*(*Name*). We cannot use "=" because this would also default to a relational operator, but one that returns **true** iff the two relations are identical. The query seeks intersection only between the sets in *Title* and *Composer*.)

  let *Name* **be** *Title*;
  Title **where** *Name* **comp** *Composer* **in** *Frodos/Group/Performance/Work*

- Find works performed by groups whose ticket price is known in advance.

  Performance/Work **where** [] **in** *TicketPrice* **in** *Frodos/Group*

- Find names of all groups located in Santa Clara county. (We replace the external predicate, *isInCounty*, of the Lorel paper, by the relation *isInCounty*(*City,County*): external predicates are special cases of procedures, or "computations", which we have not discussed in this paper, so we use **comp** instead as an alternative solution.)

  Name **where** *Location/City* **comp**
  ([*City*] **where** *County*=`"Santa Clara"` **in** *isInCounty*)
  in *Frodos/Group*

- Find the names of all groups such that either the group is an opera group or it performs on 3/19/95. (Our result is correct even for groups with no performance date provided, as is Lorel's, but for a slightly different reason. An absent attribute, or empty nested relation, is understood to have a $\mathcal{DC}$ null value, and this translates to a $\mathcal{DC}$ Boolean null in the Boolean condition. Boolean operators deal with null values just as other operators do: in particular, $\mathcal{DC}$ acts as the identity of **or** and so is effectively ignored.)

  Name **where** *Category*=`"Opera"` **or** *Performance/Date*=`"3/19/95"`
  in *Frodos/Group*

- Find the names of restaurants whose entrees all cost < \$10. (While we avoid needing a new keyword, SATISFIES, as Lorel does, we must go beyond the relational algebra discussed in this paper to include "QT-selectors" which allow quantification. Note that we cast *Entree/Price* to **integer** in order to resolve the union type of this attribute.)

Name **quant** (•=1)*Entree* **where** (**integer**)*Entree*/*Price*<10
  in *Frodos*/*Restaurant*

- Find the names of restaurants that offer more than seven entrees priced $10 or less. (Again, a QT-selector. No need for COUNT.)

  Name **quant** (#>7)*Entree* **where** (**integer**)*Entree*/*Price*<10
    in *Frodos*/*Restaurant*

- Select the titles of all performances of works by Gilbert and Sullivan.

  relation *GnS*(*Name*) <− {(`"Gilbert"`),(`"Sullivan"`)};
  Title **where** *Composer* **sup** *GnS* **in** *Frodos*/*Group*/*Performance*/*Work*

- Find names of restaurants located in Palo Alto. (As with the Lorel example, this query goes beyond the data shown in the Lorel paper and the schema shown above, and supposes that we are unsure of just where under *Restaurant* the *City* appears. Hence the need for the Kleene star in .*, which is elided to just * .)

  Name **where** */*City*=`"Palo Alto"` **in** *Frodos*/*Restaurant*

- Find all possible sequences of labels in the *Frodos* database. (We do not repeat the code at the end of section 3, but refer the reader to it and suggest replacing *company* by *Frodos*.)

- Find both *Name* and *Phone* objects for every group in Palo Alto. (We embed the expression into a statement to name the result.)

  LocalGroups <− [*Name*,*/*Phone*] **where** *Location*/*City*=`"Palo Alto"`
    in *Frodos*/*Group*;

- In the previous query, we don't have Lorel's problem of failing to correlate Name, Phone in the result, so we don't need FOREACH; but we could retain a nested structure anyway:

  let *GroupNP* **be** [*Name*,*/*Phone*] **where** *Location*/*City*=`"Palo Alto"`
    in *Group*;
  LocalGroups <− [*GroupNP*] **in** *Frodos*;

- Find name, type, and rating of restaurants (where type is category, and joining with an additional relation which gives ratings.)

  relation *BBB*(*Name,Rating*) <− {(`"Blues on the Bay", 4`),
    (`"The Greasy Spoon", 1`)}
  let *Type* **be** *Category*;
  RatedRestaurant <− *Frodos*/*Restaurant* **join** *BBB*;

The *Guide* in [3] is shown both textually as a hierarchical table and as a graph with edges labelled by attributes, leaves labelled by values, and each node identifying its object. It is simpler than *Frodos* but has some interesting aspects. Here is a schema.

relation *Guide*(*restaurant*);
domain *restaurant*(*category,name,addresses,zip,prices,nearbys*);
domain *addresses*(*address*);
domain *address* **string**|(*street,city,zip*);
domain *prices*(*price*);
domain *nearbys*(*nearby*);
domain *nearby restaurant*;
domain *zip* **string**|**integer**;

First, the schema is recursive on *restaurant*, since *nearby* is a descendent attribute of *restaurant* which is itself a *restaurant*. Second, an attribute such as *zip* occurs twice, in different places. It is understood that *zip* will probably appear not more than once in any *restaurant* entry, but, in keeping with the flexibility of semistructured data, this is not enforced.

The queries presented display these two aspects of *Guide*.

- Find the addresses of all restaurants in the 92310 zipcode. (First version: assume *zip* appears only under *address*. Note the cast, since *zip* is either **string** or **integer**.)

  where (**string**) *zip*=`"92310"` **in** *Guide/restaurant/address*

- Find the addresses of all restaurants in the 92310 zipcode. (Second version: *zip* may appear at either place.)

  where (**string**) *zip*=`"92310"` **in** *Guide/restaurant/**

  or

  where (**string**) *zip*=`"92310"` **in** *Guide/restaurant/address?*

- Find the names and zipcodes of all "cheap" restaurants.

  [**red union of** (*name* **join** ?/*zip*)] **in grep** `cheap"` **in** *Guide/restaurant*

- Return the names of all restaurants having an address with a zip code of 92310 or that are located on El Camino Real in Palo Alto.

  name **where** (**string**)*address/zip*=`"92310"` **or**
  (*address/street*=`"El Camino Real"` **and**
  address/city=`"Palo Alto"`) **in** *Guide/restaurant*

- Find the names of cheap restaurants with zipcode 92310.

  name **in grep** `cheap"` **where** (**string**) (?/*zip*)=`"92310"` **in** *Guide/restaurant*

- Return all restaurants that have two distinct paths to the same nearby restaurant. (We test *nearbys* for overlap with itself in two copies of *restaurant*. A clever compiler could make this as fast as using variables, the way Lorel does.)

  let *category′* **be** *category*;
  let *name′* **be** *name*;
  let *addresses′* **be** *addresses*;
  let *zip′* **be** *zip*;
  let *prices′* **be** *prices*;
  let *nearbys′* **be** *nearbys*;
  where *nearbys′* **comp** *nearbys* **in** *Guide/restaurant* **join**
  [*category′*,*name′*,*addresses′*,*zip′*,*prices′*,*nearbys′*] **in** *Guide/restaurant*

- Record that my favourite restaurant is the Saigon.

  myFavourite <− **where** *name*=`"Saigon"` **in** *Guide/restaurant*;

- Change my mind: my favourite restaurant is the Chef Chu.

  myFavourite <− **where** *name*=`"Chef Chu"` **in** *Guide/restaurant*;

- Delete *myFavourite*.

  myFavourite <− $\mathcal{DC}$;

  or

  update *myFavourite* **delete**;

- Create a new relation.

  relation *new* <−
  {< *a* >5< /*a* ><*bs*>< *b* >"X"< /*b* >< *b* >"Y"< /*b* >< /bs>};

- Update *price* in *Prices*. (Note that even if *Prices* originally had many tuples, this update will result in only one, since it makes every value 7 and duplicates are not allowed. In this and following examples, we assume, along with [3], that the schema is simple enough not to need explication.)

  update *Prices* **change** *price* <− 7;

- Create a new *Prices* relation. (This replaces the old *Prices* with another. It has the *same* effect as the previous update.)
  relation *Prices*(*price*) <− {(`7`)};

- Update *price* in *Prices* by adding 1 to each value.
  update *Prices* **change** *price* <− *price*+`1`

- Add "Sunnyvale" to the addresses of my favourite restaurant. (This updates the **string** alternative for *address*. It could be shortened by replacing *Sunnyvale* by the relation constant {<*address*>`Sunnyvale`}, but we show an explicit version.)
  relation *Sunnyvale*(*address*) <− {(`"Sunnyvale"`)};
  update *myFavourite/addresses* **add** *Sunnyvale*;

- Add the restaurant's city as a direct subobject of the restaurant object if the city is Palo Alto or Menlo Park. (The Lorel example does not delete *city* from *addresses* under these circumstances. Nor do we: doing so needs a **transpose** operation to reveal the type of *address*, and remove *city* only if it is a relation, not a string.)
  let *city* **be** *addresses/address/city*;
  update *Guide* **change** *restaurant* <−
  if *restaurant/name*=`"Palo Alto"` **or** *restaurant/name*=`"Menlo Park"`
  then [*category,name,addresses,city,zip,prices,nearbys*] **in** *restaurant*
  else *restaurant*;

- Transform all the *restaurant* labels to *eatery* labels.
  let *eatery* **be** *restaurant*;
  Guide <− *eatery* **in** *Guide*;

A.5 UnQL

Independently, Buneman and colleagues at the University of Pennsylvania were developing UnQL, a query language for unstructured data. Subsequently, UnQL and Lorel were more or less fused [1]. We look at the queries in four papers starting in 1995 [7, 6, 5], and [20].

The data structure underlying the UnQL work is a labelled tree without the exceptional leaf nodes used by Lore: the values that Lore stores on leaves are edge labels in UnQL just as every attribute is an edge label, and so data values and attribute names are treated equally. The bibliography database of [7] can be expressed as nested relations:
relation *bib*(*doc*);
domain *doc*(*topics,book*|*article*);
domain *topics*(*topic*);
domain *book*(*title,authors*);
domain *article*(*title,authors*);
domain *authors*(*author*);
and *topic, title*, and *author* have type **string**.

Note that "unstructured data" is not supposed to possess a schema separate from the data: we could go on to show their data as well by displaying the nested relations, but we have shown only the structure for brevity. This structure must be known to the query writer, as is clear from UnQL's formulation of the three queries the paper presents.

Example 1. Find the titles of all books on Genetics.
book/title **where** *topics/topic*=`"Genetics"` **in** *bib/doc*;

Example 2. Find the authors of all documents, regardless of the type of the document.
authors **in** *bib/doc/*.

Example 3. Find the title and topic[sic] of all books by Gonick and Wheels.
relation *GonWheel*(*author*) <− {(`"Gonick"`),(`"Wheels"`)};
[*book/title, topics*] **where** *book/authors*=*GonWheel* **in** *bib/doc*;

The paper then goes on to be the first to introduce cycles into the data structure, by separating author data from document data and linking both ways. In the relational representation, we need only close a loop:

domain *author* **string**;

becomes

domain *author*(*name,papers*);

with

domain *papers*(*doc*);

("*papers*" is used by the authors despite referring to books, too.)

A movie database is introduced in [6] and used in [5]

relation *DB*(*Entry*);
domain *Entry*(*Movie*|*TVShow*);
domain *Movie*(*Title,Cast,Director,Refs,RefBy*);
domain *TVShow*(*Title,Cast,Episodes*);
domain *Cast*(*Actors,CreditActors*);
domain *Actors*(*Actor*);
domain *CreditActors*(*Actor*);
domain *Episodes*(*Episode,SpecialGuests,Director*);
domain *SpecialGuests*(*Actor*);
domain *Refs Movie*;
domain *RefBy Movie*;

and every datum is a **string**, except *Episode* which is **integer**.

Here are selected queries from these two papers.

Example 3.5 Give the titles and casts of all entries—movies, TV shows, etc.
[*Title,Cast*] **in** *DB/Entry/*.

Example 3.6 Give actress/actor and title tuples for movies.
[*Title,Cast/Actors*] **in** *DB/Entry/Movie*

Example 3.7 Find the set of all strings in the database.
let *typval* **be transpose**(*type,value*) **union** *typval*;
value **where** *typval/type*=**string in** *DB*;

Example 3.8 Find all movies involving "Bogart" and "Bacall".
relation *BogBac*(*Actor*) <− {(`"Bogart"`),(`"Bacall"`)};
where */*Actors* **sup** *BogBac* **in** *DB/Movie*

Example 3.9 Find all movies involving "Bogart" and "Bacall", avoiding the cycles possibly generated by *Refs* and *RefBy*: do not follow paths containing *Movie* more than once.
where [ˆ*Movie*]*/*Actors* **sup** *BogBac* **in** *DB/Movie*

Example 4.1 Replace all *SpecialGuest* labels with a *Featuring* label.
let *Featuring* **be** *SpecialGuest*;
update *DB*/* **change insert** *Featuring* **remove** *SpecialGuest*;

Example 4.2 For TV shows, replace all *SpecialGuest* labels with a *Featuring* label.
let *Featuring* **be** *SpecialGuest*;
update *DB*/*/*TVShow*/* **change insert** *Featuring* **remove** *SpecialGuest*;

- Where in the database is the string "Casablanca" to be found?
  let *attrCasa* **be** (*attr* **in grep**(*attr*) `"Casablanca"` **in** .) **union** *attrCasa*;
  attrCasa/attr **in** *DB*;

- Are there integers in the database greater than 2^{16}?
  let *typval* **be transpose**(*type,value*) **union** *typval*;
  [] **where** *type*=**integer and** (*integer*)*value*>`2^16` **in** *DB/typval*

- What objects in the database have an attribute name that starts with "Act"?
  let *attribs* **be transpose**(*attr*) **union** *attribs*;
  where grep `"^Act"` **in** *attribs* **in** *DB*

The queries from [6] that we did not answer above pertain to an emulation of flat relations in the labelled tree model. This database can also be emulated by nested relations (although there is little motivation to do so). A literal interpretation of the emulation is

relation *DB*(*R1,R2*);
domain *R1*(*tup*);
domain *R2*(*tup*);
domain *tup*(A, B, C);
domain *tup*(C, D);

where we use polymorphism for *tup*. Given the application, we can simplify:

relation *DB*(*R1,R2*);
domain *R1*(A, B, C);
domain *R2*(C, D);

The queries are

Example 3.1 Compute the union of all trees t such that *DB* contains an edge *R1*$\Rightarrow t$ emanating from the root.
DB/R1

Example 3.2 Find the union of all tuples in both relations.
DB/R1 **union** *DB/R2*
Here, polymorphism allows the heterogeneous result to be a single, if awkward, relation. Note that UnQL does not need to name *R1* and *R2*.

Example 3.3 Join *R1* and *R2* on their common attribute C and then project onto A and D.
[[A, D] **in** (*R1* **join** *R2*)] **in** *DB*

Example 3.4 Perform a group-by operation on *R2* along the C column.
let *Dset* **be equiv union of** D **by** C;
[[*C,Dset*] **in** *R2*] **in** *DB*

Finally, [20] uses a bibliographic database, which we capture as

relation *Bib*(*paper*|*book*);
domain *paper*(*authors,title,year,pages,references*);
domain *book*(*authors,title,publisher,references*);
domain *references*(*reference*);
domain *authors*(*author*);
domain *author* **string**|(*firstname,lastname*);
domain *pages*(*first,last*);
domain *reference Bib*;

The queries are

- Return the titles of papers written since 1995.
  title **where** *year*>`1995` **in** *Bib/paper*;

- Find all papers referenced directly or indirectly by Ullman:.
  references/reference/paper **where** *authors/author/lastname*?=`"Ullman"`
  in *Bib/*.

- Retrieve all integers in the database.
  let *typval* **be transpose**(*type,value*) **union** *typval*;
  value **where** *typval/type*=**integer in** *Bib*;

- Convert all integers to strings in books, leaving integers untouched in other publications.
  let *attr′* **be** *attr* **cat** *S*;
  let *atTyVal* **be transpose**(*attr,type,value*) **union** *atTyVal*;

A.6 OQL-doc

The O_2 document database in [2] can be put into nested relational form (without constraints, and without hiding information as private—which, together with instantiation and other aspects of full object orientation, requires procedural mechanisms and is beyond the scope of this paper).

relation *Articles*(*title,authors,affil,abstract,sections,acknowl,status,cites*);
domain *authors*(*author,seq*);
domain *sections*(*seq,title,bodies,subsectns*);
domain *subsectns*(*seq,title,bodies,subsectns*);
domain *bodies*(*seq,figure*|*paragr*);
domain *figure*(*picture,caption,label*);
domain *paragr*(*reflabel*);
domain *cites*(*citation*);
domain *citation Articles*;

Note that we have added *seq* attributes to *authors, sections, subsectns* and *bodies* to distinguish first, second, etc. authors, since relations are not ordered. We have also made the *subsectns* attribute recursive, beyond the O_2 specification. We do not follow up the specification of *label* and *reflabel* as lists of Objects, because this is not explained or used in the OQL-doc paper.

We *have* added a recursive *cites*(*citation*) attribute in order to answer query 6, below.

Q1 Find the title and the first author of articles having a section with the title containing "SGML" and "OODB".
[**red union of** (*title* **join where** *seq*=`1` **in** *authors*)] **where**
[] **in grep** `"SGML"` **in** *section* **and** [] **in grep** `"OODB"` **in** *section*
in *Articles*

Q2′ Find the subsections of articles containing the sentence "complex object". (Nonrecursive.)
[*section/subsectn*] **in grep** `"complex object"` **in** *Articles*

Q2 Find the subsections, containing the sentence "complex object", of articles. (Nonrecursive.)
grep `"complex object"` **in** *Articles/section/subsectn*

Q2+ Find the subsections, containing the sentence "complex object", of articles. (Recursive.)
grep `"complex object"` **in** *Articles/section/subsectn*+

Q3 Find all titles in *Articles*.
*/*title* **in** *Articles*

Q3+ Find all the paths titles are on in *Articles*. (See the end of section 1.3.)
let *path* **be self**/*attr*;
let *path′* **be self**/*path*;
let *paths* **be** *path* **in**
((*path* **where** *attr*=**quote** *title* **in transpose**(*attr*))
[*path* **union** *path′*] (*path′* **in** [**red union of** *paths*] **in** .));
[**red union of** *paths*] **in** *Articles*

Q3++ Find all titles in *Articles* and the paths they are on.

```
let path be self/attr;
let path′ be self/path;
let paths be [title,path] in
        ((title join (path where attr=quote title in transpose(attr)))
        [title,path union title,path′] ([title,path′] in [red union of paths]
        in .));
[red union of paths] in Articles
```

Q3+++ Find titles containing “OODB” in *Articles* and the paths they are on.

```
let path be self/attr;
let path′ be self/path;
let paths be [title,path] in (((grep "OODB" in title) join
        (path where attr=quote title in transpose(attr)))
        [title,path union title,path′] ([title,path′] in [red union of paths]
        in .));
[red union of paths] in Articles
```

Q4 Find the structural differences between my new and old articles.

```
let path be self/attr;
let path′ be self/path;
let paths be [path] in ((transpose(attr))
        [path union path′] (path′ in [red union of paths] in .));
([red union of paths] in my_article) diff
        [red union of paths] in my_old_article
```

Q5 Find the attributes defined in articles whose value contains the string “final”.

```
attr in grep(attr) "final" in Articles/*
```

Q6 Return pairs of documents referencing each other, given that the title of the first document contains the word “digital”. (We suppose that the title and the set of authors suffice to identify a document, to keep things brief: we do not have object identity, but we can test two relations for equality using the join written “=”. Hint: start by thinking of this problem for the flat binary relation *cites*(*article,citation*); the query, and even its extension to arbitrary depths of reference, then becomes trivial in Graphlog, for instance.)

```
let ctitle be cites/citation/title;
let cauthors be cites/citation/authors;
let title′ be title;
let authors′ be authors;
let ctitle′ be ctitle;
let cauthors′ be cauthors;
selfcite2 <− [title,authors,ctitle,cauthors]
        where title=ctitle′ and authors=cauthors′ in
        ((where ([] in grep "[Dd]igital" in title) in Articles)
        [ctitle,cauthors join title′,authors′]
        [title′,authors′,ctitle′,cauthors′] in Article
```

A.7 From Relations to Semistructured Data and XML

Abiteboul, Buneman, and Suciu [1] offer interesting examples starting in Chapter 4 and based on the following bibliographic database (we show, with them, data for the initial queries, and we add some data to illustrate further queries, but do not cover all the queries we discuss below).

DB(	*biblio*						
	(*book*						)
	(*authors*	*date*	*title*			)	
	(*author*	)					
	`Roux`	`1976`	`Database Systems`				
	`Combalusier`						
	`Smith`	`1999`	`Database Systems`				
	(*paper*						)
	(*authors*	*title*	*year*	*refers_to*		)	
	(*author*	)		(*authors*	*cite*)		
				(*author*	)		
	`Suciu`	`Semi-Databases`	`2001`	`Roux`	`RC76`		
				`Combalusier`			
				`Smith`	`Sm77`		
				`Smith`	`Sm99`		
				`Suciu`	`Su01`		
	`Jones`	`Smith's DB Work`	`2000`	`Smith`	`Sm77`		
				`Smith`	`Sm99`		

This can be formulated as the nested relations

domain *authors*(*author*);
domain *book*(*authors,date,title*);
domain *refers_to*(*authors,cite*);
domain *paper*(*authors,title,year,refers_to*);
domain *biblio book*|*paper*;
relation *DB*(*biblio*);

Here are the queries, identified by their numbers where provided and by letters otherwise.

Q1 Find the set of book authors.
DB/*biblio*/*book*/*authors*

Q2 Find documents written by Smith.
let *row* **be** *biblio*/.;
row **where** *authors*/*author*=`"Smith"` **in** *DB*

Q2 ′ Find documents about databases.
where ([] **in grep** `"[dD]atabase"` **in** *title*) **in** *DB*/*biblio*

Q3 Find the set of book authors, retaining the distinction between books.
let *row* **be** *authors* **in** *biblio*/*book*;
DB/*row*

Q4 Find books including Roux as an author, and split into separate entries for each author.
let *row* **be** *authors* **union** *title*;
row **where** *authors*/*author*=`"Roux"` **in** *DB*/*biblio*/*book*

Qa Find authors who are referred to at least twice in some paper with "Database" in the title. [We use a QT-selector again here.]
let *multcite* **be** *author* **quant** (#>1) *cite* **in red union of** (*authors* **join** *cite*);
refers_to/*multcite*/*author* **where** [] **in grep** `"Database"` **in** *paper*) **in** *DB*/*biblio*

Q5 Find authors of 1997 papers.
authors/*author* **where** *year*=`1997` **in** *DB*/*biblio*/*paper*

Q6 Find titles of papers written after 1989.
title where year>`1989` **in** *DB*/*biblio*/*paper*

Qb Find all occurrences of “Shakespear”.[5]

```
[author,attr,val] in grep "Shakespear" in DB/biblio/.*
```

Qc Create *publication*(*type,title*) where type is “book” or “paper”, for documents after 1989.

```
let type be "book";
B <− [type,title] in DB/biblio/book;
let type be "paper";
P <− [type,title] in DB/biblio/paper;
B union P
```

(This solution can be made more general: see the next query.)

Qd *Group papers under their year of publication.* (The objective is to convert data, namely the values of *year*, to attributes. This was also the objective of the previous query, but we now do it generally, using attribute metadata, casting to type **attribute**, and the metadata operator, **eval**.)

```
let titles be equiv union of relation(authors,title,refers_to) by year;
let (attr)eval year be titles;
next <− [year,titles] in DB/biblio/paper;
[(attr)year in next] in next
```

Qe *Find the path from the root to the string* `"[Ss]emi"`.This requires us to include a **grep** operation in the recursion given to find all paths in section 3. Note that this code will produce `DB/biblio/paper/title`, but, should there be a reference back to Suciu’s paper by some other paper, the code will not provide the extended additional path, say, `DB/biblio/paper/title/ref/paper/title`: see the discussion in the book [1].

```
let path be self/attr;
let spath be ([path,attr] in transpose(attr)) comp
        grep(attr) "[Ss]emi" in .;
let path′ be self/spath;
let paths be [path] in (([path] in transpose(attr)) [path union path′]
        [path′] in [red union of paths] in .);
[red union of paths] in company
```

Qf Find books published by Morgan Kaufmann.

```
answer <− [author,title] where publisher/name="Morgan Kaufmann"
        in DB/biblio/book;
```

Qg *Find title and price of all books* (assuming price is optional: this makes no difference to the nested relational formulation).

```
let booktitle be title;
let bookprice be price;
result <− [booktitle,bookprice] in DB/biblio/book;
```

Qh For all authors, group under each the titles he/she published.

```
let titles be equiv union of relation(title) by author;
let authTitl be authors join title;
result <− [author,titles] in DBbiblio/./authTitl;
```

Qi *Find all titles in French.* This is a query on attributes of XML tags. We treat attributes on a par with nested tags, that is, as attributes. So suppose the XML is `<book language="French">` for Roux & Combalusier, and that other books may or may not have languages specified.

```
title where language="French" in DB/biblio/book
```

[5]The mispelling is deliberate; Abiteboul, Buneman and Suciu themselves sometimes used it.

Qj *Find all authors who have published at least two books.* We use *authTitl* from query Qh.

result <− *author* **quant** (#>1)*title* **in** *DB/biblio/book/authTitl*;

Qk Find all publications published in 1995 for which Smith is either an author or an editor.

title **where** *date*=`"1995"` **and**
(*editor*=`"Smith"` **or** *authors/author*=`"Smith"`) **in** *DBbiblio/book*;

Ql Using the recursively nested *part*(*name,brand,part*), find the name of every part element that contains a brand name equal to "Ford", regardless of the nesting level of that part.

let nameFord be *name* **where** *brand*=`"Ford"` **in** *part*;
(*nameFord*)* **in** *part*;

Qm *Retrieve all persons whose last name precedes the first name.* Since relations have no implicit ordering, we must suppose that the data was extracted from marked-up text, and we have, as in section 7.2, also extracted an explicit *Pos* attribute. Then the relation is *person*(*firstname,fpos,lastname,lpos*) and the result is easy.

where *lpos*<*fpos* **in** *person*

Qn *Reverse the order of all authors in a publication.* Again, we suppose the data has been extracted, together with postions, from marked-up text. Roux and Combalusier might appear in the string `J.P. Roux and T.H. Combalusier`, starting at position 122, and so the relation *authors* for this work in *DB/biblio/book* would show

author	*pos*
`Roux`	`127`
`Combalusier`	`141`

We swap the positions (this code will reverse any number of authors, but not necessarily swap any positions exactly) and then actualize.

let *revpos* **be** (**red** + **of** *pos*) − *pos*;
pub <− [*author,revpos*] **in** *DB/biblio/book/authors*;

A.8 Web Query Languages

We reformulate queries given for WebSQL [16] and WebLog [15] in terms of the hypertext in section 7.2.

WebSQL represents hypertext as two flat relations, *Document*(*url,title,text,type,length,modif*) and *Anchor*(*base,href,label*), where the attribute names make their functions pretty self-evident: *type, length* and *modif* are from the HTTP headers mentioned at the end of section 7.2; *href* and *label* appear in the anchor tag `<A HREF=`*href*`>`*label*`</A>`, and *base* connects *Anchor* to *Document.* Here are the queries in terms of the recursively nested relation in section 7.2. Note that we assume, as do the WebSQL authors, that the search is of all documents reachable from a set to which we have access through, in our case, *HTML.*

Ex.1 Find all HTML documents about "hypertext".

where *A/format*=`"html"` **in grep** `"hypertext"` **in** *HTML/Body*(*/A/Body*)*

Ex.2 Find all links to applets from documents about "Java".

[*A.content, href*] **where** ([] **in grep** `"applet"` **in** *A.content* **in** *A*) **in**
where [] **in grep** `"Java"` **in** *HTML/Body*(*/A/Body*)*

Ex.3 Starting from the Department of Computer Science home page, find all documents that are linked through paths of length two or less containing only local links. Keep only documents containing the string "database" in the title. [We have modified the definition of *local* to take two steps, through both *Body* and *A.*]

```
let local be grep "^[^/]" in href in Body/A;
where ([] in grep "database" in Head/Title) in (local)?(/local)?
      where href="www.cs.toronto.edu" in HTML
```

Ex.4 Find all documents mentioning "Computer Science" and all documents that are linked to them through paths of length two or less containing only local links.

```
CS <− grep "Computer Science" in HTML(/Body/A)*;
CS union CS(/local)?(/local)?
```

Weblog is based on Prolog via Datalog and SchemaLog. It is thus structured as a sequence of Horn clauses, with the usual semantics of universal quantification of variables defined by the clase and existential quantification of variables linking the defining predicates. The examples given all refer to *leyurl*, so we define it beforehand.

```
let leyurl be "http://www.informatik.uni-trier.de/~ley/db/index.html";
```

Some of the WebLog queries claim to generate html results, but the paper does not show what they look like, and so we do not attempt to duplicate this result structuring.

Q.1 We are interested in collecting all citations (hyperlinks) referring to HTML documents, that occur in the Database Systems & Logic Programming page. We would also like this collection to contain the title of the document the citation refers to. (One level of traversal.)

```
let title be "all citations";
let hlinks be red union of relation(href, Title);
[title,hlinks] in [hlinks] in [href,Body/A/Head/Title] where
      href=leyurl and [] in grep "\.html$" in Body/A/href) in HTML;
```

Q.2 Find all documents in the Ley server that have information relating to 'Interoperability'. [To match this query, we introduce a relation *synonym*(*source*,*syn*), which contains a dictionary of synonyms: WebLog, of course, introduces a predicate for this. We also reuse *local* from above. Our formulation of this query follows all paths all of whose documents refer to interoperability.]

```
let Ley be where href=leyurl in HTML;
let Interop be grep(source where syn="Interoperability" in synonym)
      in local;
Ley(/Interop)*/href;
```

Q.3 Suppose we know that a paper on Coral has appeared in the VLDB Journal. We do not know which year this paper appeared, but would like to find this information. [We suppose that the HTML Body contains a non-HTML tag, `<year>`, which, of course, becomes an attribute.]

```
Body.content/year where [] grep "VLDB Journal" in Body.content and
      lowercase(../Head/Title)="coral" in Ley/Body(/A/Body)*
```

Q.4 We would like to compile the citations of all CFP's of conferences having 'interoperability' as a topic of interest. We would also like to include information on the submission deadline in this compilation. [We make the same rather special assumptions about the structure of Calls for Papers as do the WebLog authors. The WebLog query includes the *synonym* predicate applied to `"submit"`, which we can do, but it is easier to read just `"papers due"`.]

```
let submit be grep(;x) "papers due:  \x" in Body.content;
let submitDate be "submission date:  " cat submit/x;
let title be "allcfps";
cfp <− [title,../href,submitDate] in grep "Conference in Ley/Body(/A/Body)*;
```

Q.5 Suppose we would like to restructure the newly generated cfp further in such a way that all conferences having a deadline in the same week are grouped together in a page. [We implement the WebLog *dates2weeks* predicate using a relation *months*(*m#*,*#days*) giving the number of days in each month, and we suppose the date format to be yy/mm/dd, which we convert to yy/ddd.]

let *monthend* **be fun** + **of** *#days* **order** *m#*;
let *monthbase* **be if** *m#*=1 **then** 0 **else fun pred of** *monthend*;
let *m#* **be**(**intg**) *x*;
let *day* **be**(**intg**) *y*;
let *submitmd* **be** [*m#,day*] **in grep**(;*x*, *y*) `".*/\x/\y"` **in** *submitDate*;
let *yd* **be** *monthbase* + *day*;
let *ddd* **be** *yd* **in** (*submitmd* **comp** [*monthbase,m#*] **in** *months*);
let *julian* **be** *ddd*/*yd*;
let *week* **be** *julian* **mod** 7;
[*week,href,title,submitDate*] **in** *cfp*;

A.9 XQuery

XQuery queries XML data and, like SQL before it, has become a de facto standard. Here are the queries from [8], formulated for nested relations.

Except for query Q5, the example database is the on-line auction represented by the two nested relations

```
items.xml
(items                                                         )
(item                                                  status)
(itemno    seller   description   reserve-price   end-date)

bids.xml
(bids                                          )
(bid                                         )
(itemno    bidder    bid-amount    bid-date)
```

Q1. *List the descriptions of all items offered for sale by Smith.*
description **where** *seller*=`"Smith"` **in** *items.xml/items/item*

Q2. *List all description elements found in the document items.xml.*
items.xml//description

Q3. *Find the status attribute of the item that is the parent of a given description.*
status **where** *item/description*=*descr* **in** *items.xml//*

Qa. Items with reserve-price > 1000.
where *reserve-price* > 1000 **in** *items.xml/items/item*
Relational semantics dictate that a scalar attribute such as *reserve-price* is either null or a singleton, but never multiple. For multiplicity, we need a non-scalar, i.e., nested, attribute such as *reserve-prices*(*reserve-price*). With this, we can accommodate all the syntactic variations discussed on p.603 of the paper. For instance, *item*[*reserve-price* > 1000] in XQuery is
where *reserve-prices/singleton* & *reserve-prices/reserve-price* > 1000
  in *items.xml/items/item*
and *item*[*reserve-price* **gt** 1000] in XQuery is
let *singleton* **be** 1 = **red** + **of** 1;
where *reserve-prices/reserve-price* > 1000 **in** *items.xml/items/item*
Furthermore, we can find, for example, any item node with exactly four reserve-price child nodes whose value is greater than 1000:
let *count1000* **be red** + **of if** *reserve-price* > 1000 **then** 1 **else** 0;
where *reserve-prices/count1000* = 4 **in** *items.xml/items/item*

Qb. Fifth item
where *seq* = 5 **in** *items.xml/items/item*
Note that relations abstract over ordering, so a *seq* attribute must be put in explicitly. This is done automatically by a generator such as **mu2nest** in section 7.2.

Qc. Items containing a reserve-price
let *count* **be red** + **of** 1;
where *reserve-prices/count* > 0 **in** *items.xml/items/item*
which tests the nested attribute *reserve-prices*, from above, for non-emptiness. (We can also find items containing, say, 6 reserve-prices.)

Qd. Node identity: $node1 is $node2, $node1 isnot $node2
Relational semantics do not distinguish identity from value, so identity information must be explicitly represented as an extra attribute. The paper does not give an example for us to work.

Qe. Order: $node1 < $node2
let *startJoe* **be red** + **of if** *seller/content* + `"Joe"` **then** *start* **else** 0;
let *startSue* **be red** + **of if** *seller/content* + `"Sue"` **then** *start* **else** 0;
where (*seller/content* = `"Joe"` **or** *seller/content* = `"Sue"`) **and**
startJoe < *startSue* **in** *items.xml/items/item*
using the *content* and *start* attributes generated by **mu2nest**.

Qf. Not function: item[not(reserve-price)]
where not [] **in relation**(*reserve-price*) **in** *items.xml/items/item*

Qg. Namespace: auction = "items.xml/items"
let *auction* **be quote** *items.xml/items*;
where *reserve-price* > 1000 **in** *auction/item*;
using the metadata **quote** operator.

Qh. Constructor: constants
relation *highbids*(*status,itemno,bid-amount*) <− {(`"pending",4871,250.00`)};

Qi. Constructor: evaluated (1)
let *maxbid* **be red max of if** *itemno* = *i* **then** *bid-amount* **else** −1;
let *status* **be** *s*;
let *itemno* **be** *i*;
highbid <− [*status,itemno,bid-amount*] **in** [*maxbids*] **in** *bids*;
where *i* and *s* are parameters, which might be arguments to a procedure containing the above code.

Qj. Constructor: evaluated (2)
highbid <− [*status,itemno,bid-amount*] **in** *b*;

Qk. Computed element constructor
This would use metadata and casting as illustrated for the next example.

Ql. Computed attribute constructor
let (**attr**) **if** *sex* = `"M"` **then** `"father"` **else** `"mother"` **be** *name*;
Father and mother attributes are mutually exclusive: polymorphism takes care of the "null values" of the absent one when the other is present.

Q4. *For each item that has more than ten bids, generate a popular-item element containing the item number, description, and bid-count.*
let *countBids* **be equiv max of par** + **of** 1 **order** *bidder* **by** *itemno*
by *itemno*;
popular-item <− ([*itemno,description*] **in** *items.xml/items/item*) **join**
[*itemno,countBids*] **where** *countBids* > 10 **in** *bids.xml/bids/bid*;
Note also that the relational and domain algebras abstract over looping, so we do not need the **for** construct.

Q5. *Given a sequence of emp elements, replace their salary, commission, and bonus subelements with a new pay element containing the sum of the values of the original elements, and order the resulting sequence in ascending order by the value of the pay element.*

> **let** *pay* **be** *salary* + *commission* + *bonus*;
> **let** *empNP* **be** [*name,pay*] **in** *emp*;
> **update** *emps* **change** *emp* <− *empNP*;

assuming the database

> *emps*(*emp*(*name,salary,commission,bonus*))

Again, ordering is not explicit in relations. A **fun** .. **order** *pay* expression in the domain algebra suffices to order by *pay* when needed.

Q6. *Construct a new element named recent-large-bids, containing copies of all the bid elements in the document bids.xml that have a bid-amount of more than 1000 and a bid-date after January 1, 2002.*

> **relation** *cutoff*(*year,month,day*) <− {`2002,1,1`};
> *recent-large-bids* <− **where** *bid-amount* > 1000 **and**
>   *dateGT*(*bid-date,cutoff*) **in** *bids.xml/bids/bid*;

using the boolean *dateGT* method of an abstract data type for dates (construction not shown here), since date comparisons are not built-in to the language.

Qm. Items with no bids.

> *items.xml//item* **diff** *bids.xml//bid*

Qn. Union of a/b, a/c.

> **let** *bc* **be** *b* **union** *c*;
> *a/bc/d*

Qo. Conditional expression.

> **let** *price* **be if** *discounted* **then** *wholesale* **else** *retail*;

Q7. *Generate a report containing the status of bids for various items. Label each bid with a status "OK", "too small", or "too late". Enclose the report in an element called bid-status-report.*

> **let** *status* **be if** *dateGT*(*bid-date,ebd-date*) **then** `"too late"` **else**
>   **if** *bid-amount* < *reserve-price* **then** `"too small"` **else** `"OK"`;
> *ans1* <− [*itemno,bidder,bid-amount,status*] **in** (*bids.xml//bid* **join** *items.xml//item*);
> **let** *bid* **be equiv union of relation**(*bidder,bid-amount,status*) **by** *itemno*;
> *bid-status-report* <− [*itemno,bid*] **in** *ans1*;

Qp. Existential quantification.

> **let** *some* **be red or of** *n* > 10; *rel/some*

Qq. Universal quantification.

> **let** *every* **be red and of** *n* > 10; *rel/every*

Q8. *Find the items in items.xml for which all the bids received were more than twice the reserve price. Return copies of all these item elements, enclosed in a new element called underpriced-items.*

> *underpriced-items* <− [*itemno,seller,description,reserve-price,end-date*] **where**
>   **equiv and of** (*bid-amount* > 2×*reserve-price*) **by** *itemno*
>   **in** (*bids.xml//bid* **join** *items.xml//item*);

Qr. Function to find largest bid-amount recorded for a given item.

```
comp highbid(item#,maxbid) is
{       maxbid <− [red max of bid-amount] where itemno=item#
   in bids.xml//bid
};
highbid["1234"];
```

Qs. Function to return an element, or a default value if it is missing.

```
comp defaulted(attr,default,result) is
{       result <− if [] in attr then attr else default
};
```

Qt. Recursive function to determine depth of hierarchy.

```
comp depth(rel,d) is
{       let xpose be transpose(attrib);
        d <− if not [] in rel then 1 else
           1 + red max of depth[ [attrib] in rel/xpose ]
};
```

Qu. Treat billing address differently according to type.

```
let a be customer/billing-address;
let result be if typeof(a)=USAddress then a/state else
              if typeof(a)=CanadaAddress then a/province else
              if typeof(a)=JapanAddress then a/prefecture else "unknown";
```

B Glossary

<−	(relational operator) 1.1.4.
/	(attribute path constructor) 2.
|	(union type constructor) 5.
coex	(semistructured link label) 5.
comp	(relational operator) 1.1.2.
$\mathcal{DC}$	(null value) 1.1.5.
equiv	(domain algebra operator) 1.2; (nested attributes) 1.3.
grep	(*pos,attr*) 1.1.3; (*val,type*) 1.1.5. (Not recursive: 2.)
igrep	(<*pattern*>, *pos1,pos2,val1,val2*) 7.1.
is	(relational operator) 1.1.4.
join	(relational operator) 1.1.2.
lowercase	(_ string operator) 1.1.3.
metadata	1.1.3
mu2nest	(marked-up text operator) 7.