Semistructure from Relations

T. H. Merrett

School of Computer Science
McGill University,
3480 University St., Montreal, Que. H3A 2A7, Canada,
Email: tim@cs.mcgill.ca

Abstract

The central characteristics of semistructured data—irregular and implicit structure, and hierarchies navigated by path expressions—are also natural developments of programming with relations. We show the central steps of this argument, starting from nested relations and developing path expressions and mechanisms for irregular structure. With this foundation, only one new operator is needed to generate programmable relations from conventional semistructured data.

Keywords: marked-up text, nested relations, path expressions, recursion, semistructured data

1 Introduction

The issues of representing and processing semistructured data have occupied an increasing proportion of research and practice in databases over the past two decades, culminating in the dominance of XML and of query and update languages for XML. The present paper is iconoclastic, showing that nested relations are sufficient for representing and processing semistructured data. (This is not to say that nobody has attempted to implement, for example, XML, on top of a relational database—the current literature is full of such work. We do claim that, to our knowledge, nobody has *replaced*, for example, XML, by a relational *language*, which is more powerful to boot.) Since flat relations are sufficient for nested relations, this means that the classical capabilities given by the late Ted Codd, augmented only in the most obvious ways needed to make them into a programming language, suffice for semistructured data.

Why is it a good idea to reduce semistructured data querying and updating to the relational model? The first reason is the classic appeal of relations: they are well-defined, mathematical objects, with a well-defined algebra for querying and updating, which also supports general programming. The second reason is the superb adaptation of relations to secondary storage: relations support exactly the abstraction of bulk data needed by SS. The third reason is integration: the same development of the relational model into a general-purpose programming language that enables it to support semistructured data also absorbs a host of other applications, including GIS and systems for spatio-temporal data, inference engines and expert systems, data mining, and systems for data distributed over the Internet, not to mention all the administrative database systems for which the relational model was originally developed (Merrett 2005*a*). Including semistructured data into the repertoire of this one framework is a major step in integrating data-based applications. For example, a genomics database (which is a good application of semistructured data) can be seamlessly linked with a geographical system to enable studies of geo-historical factors in botanical or zoological evolution. And so on.

The paper presents semistructured data processing as an inevitable result of developing a programming language for nested relations. The next section shows how relations, domains and nesting contribute to the construct called paths of attributes; it gives syntactic sugar for these, which is equivalent to the corresponding paths in semistructured data, and it shows how regular expressions on these paths convert to simple recursive operations in the relational formalism; finally, it shows the corresponding implementations of the complementary paths of conditions in selection operators, and paths of updates. (All these ideas require the support of a programming notation which is subtly but importantly different from the query notation in conventional relational database implementations.)

Section 3 introduces a metadata-generating operator, **transpose**, and shows its use in querying and updating the *schema* describing the data. Union types, so central to the flexibility of semistructured data, appear here and support transitions from single to multiple values. Together with a variant, **transposeAll**, the **transpose** operator supports the idea of a wildcard symbol for attributes, thus completing the release of semistructured data from regularity of structure. The **transpose** operator, used recursively, gives a general mechanism for discovering the schema in case it is not explicitly given.

Section 4 wraps up the discussion by bringing in marked-up text, with fully implicit schema, and data on the World Wide Web in HTML format. Here we show the generation, from marked-up text, of the nested relations we have used up to this point, and so complete the link from relations to semistructure.

We scarcely have space in this paper to do more than introduce the basic derivations of semistructured operations from those on nested relations. So we will not here be able to support the above claims for all the other applications. ((Merrett 2005*a*) gives a sketch.) We will even be able to give only the briefest introduction to the extensions to Codd's original formalism needed to interpret flat relations as nested, and that only inline in the next section. (See (Merrett 1999).)

We omit a number of extensions, such as links within semistructured data, recursion in updating, unions resulting from wildcard symbols at leaves and roots, sequencing information from marked-up text, and whole discussions of operators on pure text and of pattern matching with regular expressions: these have all succumbed to our relational approach and will appear elsewhere. We do not show the functional equivalence of nested and flat relations, which can be done simply by implementing the former with the latter.

We also omit from this paper our formulations of all the queries illustrating the classical papers on semistructured data and a couple of important predecessors. These are in the appendix of an exploratory tutorial (Merrett 2005*b*): the graphical query languages G (Cruz, Mendelzon & Wood 1987), G+ (Cruz, Mendelzon & Wood 1988) and GraphLog (Consens & Mendelzon 1990); "A C.elegans Database", ACeDB (Thierry-Mieg & Durbin 1992); relational SGML systems, (Sacks-Davis, Kent, Ramamohanarao, Thom & Zobel 1995) (Atlas) and (Blake, Consens, Kilpeläinen, Larson, Snider & Tompa 1994); Lorel (Quass, Rajaraman, Sagiv, Ullman & Widom 1995) and (Quass et al. 1995); UnQL (Buneman, Davidson & Suciu 1995, Buneman, Davidson, Hillebrand & Suciu 1996, Buneman 1997), and (Suciu 1998); OQL-doc (Abiteboul, Cluet, Christophides, Milo, Moerkotte & Simeon 1997); the book (Abiteboul, Buneman & Suciu 2000); the Web query languages, WebSQL (Mendelzon, Mihaila & Milo 1997) and WebLog (Lakshmanan, Sadri & Subramanian 1996); and finally XQuery itself (Chamberlin 2002). We are indebted to all of this prior work for guidance through the tangle of possible interpretations of "semistructured data".

2 Relations and Path Expressions

Query languages for semistructured data are based on path expressions, first proposed by Mendelzon and his group (e.g., (Consens & Mendelzon 1990)) to simplify and replace recursive coding by regular expression coding. We will need three kinds of paths for semistructured querying and programming: one for relations and attributes, one for conditions, and one for updates.

In this section, we introduce each in turn, together with the relevant underlying relational syntax, which is slightly modified from that used in conventional products in order better to support programming as opposed to mere querying.

2.1 Paths of attributes

To arrive at the hierarchical structure that is one of the characteristics of semistructured data, and at the meaning of the paths that navigate this structure, we need nested relations. To arrive at nested relations, we need a cleaner syntax for the relational algebra and we need to be able to manipulate attributes using the so-called domain algebra. Once this is done, we can write path expressions as syntactic sugar for the underlying ideas, and we can include regular expression operators such as the Kleene star as a simplified way to express recursion.

We will use a family tree example throughout this paper, with variations of form and of content as needed to bring out different aspects in their turn. We start with flat relations to illustrate the programming-language (as opposed to query-language) syntax of the relational algebra. A pair of relations could be used to describe a simple family tree.

Child(*Name, DoB, Pa, Ma*)
Spouse(*Ma, Pa, Wed*)

The projection operator is written

ChildND <− [*Name, DoB*] **in** *Child*;

which also illustrates a *statement* which assigns a value from a *relational expression* to a name. A projection on a single attribute,

[*Name*] **in** *Child*

can omit the square brackets:

Name **in** *Child*

Binary operators, with two relational expressions as operands, include **join** and **union** and are written with the operator infixed:

([*Ma, Pa*] **in** *Child*) **union** [*Ma, Pa*] **in** *Spouse*

This shows the union of two relations which have the same attributes, so **union** here serves as conventional set union.

Operations on attributes can have a simple, "horizontal", form in which two attributes are, say, added together, and aggregate, or "vertical", forms which find, say, the minimum value of an attribute:

let *Oldest* **be red min of** *DoB*;

In this statement, a *virtual* attribute, *Oldest*, is defined as the least date-of-birth. The relation in which this minumum is to be found is not specified: an essential intellectual simplification results from keeping these attribute operations independent of relations. The above statement can be seen as a parameterless function definition, using a *domain expression* on the right-hand side of the defining **be** operator. (The formalism has been mis-named the *domain* algebra instead of the alliterative *attribute* algebra, but the name has stuck.) Other domain expressions allow grouping tuples into equivalence classes, so

equiv min of *DoB* **by** *Pa*

gives the oldest child for each father, and

equiv min of *DoB* **by** *Ma, Pa*

gives the oldest child for each couple.

Nesting relations breaks first normal form, a restriction imposed early in the development of the relational model to avoid the complication of non-"simple" attributes in a relation. The domain algebra makes nesting easy, and indeed an inevitable outcome of the programming language principle that there should be no special-case constructs or data types: since nesting allows relations to be attributes, operating on nested relations must allow the domain algebra to subsume the relational algebra. Thus, in particular, we will find combinations such as **red union** in the domain algebra (and see why it is necessary to give binary relational operators such as **union** and **join** an explicit, infixed syntax).

We shift our family tree example by nesting the *Children* relation within the *Family* relation:

Family(*Ma, Pa, Wed, Children*)

given

Children(*Name, DoB*)

Let's try to extract the *Name* of a child, one level down.

let *ChildN* **be** [*Name*] **in** *Children*;

(note the relational-algebra projection in this domain-algebra statement)

famChildN <− [*ChildN*] **in** *Family*;

This gives a result which is still nested

famChildN(*ChildN*(*Name*))

so we need a means of raising levels if we want a flat relation as the result.

Since it is the naming of *ChildN*, above, that gives a name for the relation in between, it is appropriate to suppose that this relation should disappear if we were to replace *ChildN* in the projection by the (anonymous) expression defining it. This does not quite work because *Family* may have many tuples, and so there would be many *Children* relations. So in addition to anonymity we must also merge all these relations into one before we may raise the level. This gives us level-raising in nested relations without needing new syntax or any explicit operator.

FamChildN <− [**red union of** *ChildN*] **in** *Family*;

(The same idea works if we want to raise all attributes of *Children* to top level:

FamChildren <− [**red union of** *Children*]

in *Family*;)

This construct is a basic path expression, for which we can define syntactic sugar.

Family/*ChildN* ≡ [**red union of** *ChildN*]
in *Family*

(Note that the syntactic sugar covers only the special case of projections on a single attribute. We can write *Family*/*Children*, even though this gives a result with the two attributes, *Name* and *DoB*, but syntactic sugar cannot give, say, *Wed, Name* and *DoB* from *Family*. To get this would require the more general syntax and operators.)

Leaf attributes must be treated a little differently, and require a level-pushing mechanism which is the converse of the level-raising discussed above. This uses a **relation**() operator, which groups its operand attributes into a *singleton* relation (having only one tuple). We can illustrate this with code to construct the above nested relation. Given a single flat relation

Fam(*Ma, Pa, Wed, Name, DoB*)

we group *Name* and *DoB* into a nested relation which we call *Child* and which describes, for each tuple of *Fam*, only the one child in that tuple.

let *Child* **be relation**(*Name, DoB*);

Then we collect all children for each couple,

let *Children* **be equiv union of** *Child*
by *Ma, Pa, Wed*;

and create the nested relation.

Family <− [*Ma, Pa, Wed, Children*] **in** *Fam*;

If we write a leaf attribute at the end of a path expression, we can only translate it into a **union** if it has itself become a nested relation instead of the scalar attribute it actually is because it is a leaf. So this requires the **relation**() operator.

Family/*Ma* ≡ *Ma* **in** *Family* ≡
[**red union of relation**(*Ma*)] **in** *Family*

Because this is a very special use of the **relation**() operator, we will elide it in future, and, when we see a construct such as **red union of** *Ma* we will understand it to mean **red union of relation**(*Ma*).

If we modify the example further, we can illustrate recursively nested relations.

Person(*Name, Family*)
Family(*Conj, Wed, Children*)
Children(*Name, DoB, Family*)

Here *Family* and *Children* are mutually nested. (It is also possible to have a self-nested relation.)

A path expression naming a fixed number of levels can be written as above, and the notion of an *alternative* (using the regular expression operator, "?") is also straightforward, but the Kleene star, "*", needs recursion in the domain algebra.

Person/*Family*/*Children*/*Name* ≡
[**red union of**
[**red union of**
[*Name*] **in**
Children] **in**
Family] **in**
Person

Fixed-Depth Path Expression

Person(/*Family*/*Children*)?/*Name* ≡
Name **in** *Person* **union**
[**red union of**
[**red union of**
[*Name*] **in**
Children] **in**
Family] **in**
Person

Path Expression with Alternative

Person(/*Family*/*Children*)*/*Name* ≡
let *Nom* **be** *Name* **union**
[**red union of**
[**red union of** *Nom*] **in**
Children] **in**
Family;
[**red union of** *Nom*] **in** *Person*

Path Expression with Kleene Star

2.2 Paths of conditions

Paths of conditions require rather less preparation than paths of attributes did. Conditions arise in the selection operation of the relational algebra, so we need a syntax for that. To take selection over into the domain algebra for use with nested relations, we need to be able to write a selection so that it is itself a boolean value which can be nested in further conditions.

Going back to the flat relation, *Child*(*Name, DoB, Pa, Ma*), we write first a boolean-valued projection which states that the relation is not empty.

[ ] **in** *Child* ≡ "something in *Child*"
≡ "there is a child"

This *nullary* projection can be thought of as boolean because the relation in the argument has only two states that can be distinguished when it is projected on no attributes whatsoever: either it is empty or it is not. We thus define the nullary projection to be true if the argument is not empty and false otherwise.

Selection is usually combined with projection, and here we combine it with a nullary projection.

[ ] **where** *Name* = `"Joe"` **in** *Child*
≡ "there is a child, Joe"

Here is the syntactic sugar for a path of conditions, illustrated for our recursively nested example.

Person(*Name, Family*)
Family(*Conj, Wed, Children*)
Children(*Name, DoB, Family*)

DoB **where** *Family*/*Children*/*Name* = `"Mary"`
in *Person* ≡
DoB **where**
([ ] **where**
([ ] **where** *Name* = `"Mary"` **in**
Children) **in**
Family) **in**
Person

Fixed-Depth Condition Path Expression

As with paths of attributes, when we write Kleene stars in the syntactic sugar, the underlying code is recursive. This time, the recursion is conventional.

```
DoB where (Family/Children/)*Name = "Mary"
in Person ≡
func mary is
{  Name = "Mary" or
   ([ ] where
      ([ ] where mary in Children)
   in Family)
};
DoB where mary in Person
```

Fixed-Depth Condition Path Expression with Kleene Star

Note that the syntactic sugar is implemented with **or**. If **and** or **xor** were wanted, it would have to be written explicitly, without sugar. Once again, the sugar is less general than the underlying capabilities.

2.3 Paths for updates

Updates are also easily extended to the data hierarchies found in semistructured data. We need a programming syntax for updating, which allows additions of tuples, and deletions of and changes to specified tuples. Here, we focus on change.

Going back first to the flat relation, *Child*(*Name, DoB, Pa, Ma*), here is an update using a (domain algebra) conditional expression on the tuples of the relation being updated.

```
update Child change DoB <-
   if Name = "Mary" then "1933" else DoB;
```

This changes Mary's birth date to 1933 but leaves everyone else's alone.

In the mutually nested relations

Person(*Name, Family*)
Family(*Conj, Wed, Children*)
Children(*Name, DoB, Family*)

updating a value a fixed number of levels down simply requires the statement following **change** to be itself an **update**, repeatedly if necessary.

```
update Person/Family/Children change DoB <-
  if Name = "Mary" then "1933" else DoB;
≡
update Person change
  update Family change
    update Children change DoB <-
      if Name = "Mary" then "1933" else DoB;
```

Fixed-Depth Update Path Expression

An update specified by a Kleene star uses a recursive procedure, which we do not show here.

3 Irregular and Unknown Structure

As a result of possessing structure which is not always explicitly known, if at all, semistructured data needs the ability to query, and update, this structure. This can also support the further characteristics of semistructured data that its structure can be very irregular, with dynamically typed alternatives at any point in the schema, and that updates can change data types at any such point. Such considerations lead to union types, dynamic typing, and, above all, to metadata, i.e., data describing the schema.

We introduce a metadata operator, **transpose**(), which analyses tuples of a relation and explicitly characterizes each attribute by type. The **transpose** operator becomes interesting when the data types are variable, so we illustrate it with a simple union type.

3.1 Schema query and update

The simplest union type is constructed from base types such as string or integer.

```
domain DoB strg|intg;
```

The specific type for any data value must be stored with the value, and could appear on display as follows.

Child(

Name	*DoB*	*Pa*	*Ma*
Mary	**intg:**1934	Ted	Alice
James	**strg:**1935	Ted	Alice

)

The **transpose**() operator defines new attributes, whose types must have previously been declared.

```
domain att attr;
domain typ type;
domain val any;
let xpose be transpose(att, typ, val);
```

These new types are typical of metadata: **attr** is the metadata type whose values are attributes of the database (and with nesting, attributes can also be relations); **type** is the metadata type whose values are themselves types, which must be either base types or named by the programmer; **any** is the union of all types, clearly needed if an operator is to extract values of any attributes, which **transpose** does. **Transpose** may have any subset of the three arguments shown, and their function in **transpose** is determined not by their names or positions but by their types.

If *xpose* were actualized in the above example it would produce a nested relation in each of the two tuples.

```
transposeChild <-
      [Name, DoB, Pa, Ma, xpose] in Child;
```

transposeChild

(*Name*	*DoB*	*Pa*	*Ma*	*xpose*			)
				(*att*	*typ*	*val*	)
Mary	**intg:** 1934	Ted	Alice	Name	**strg**	**strg:**Mary	
				DoB	**intg**	**intg:**1934	
				Pa	**strg**	**strg:**Ted	
				Ma	**strg**	**strg:**Alice	
James	**strg:** 1935	Ted	Alice	Name	**strg**	**strg:**James	
				DoB	**strg**	**strg:**1935	
				Pa	**strg**	**strg:**Ted	
				Ma	**strg**	**strg:**Alice	

This gives us our first query on the structure of the data: find all integer dates of birth.

```
intgDoB <- where
   xpose/att=quote DoB and xpose/typ=intg
   in Child;
```

The answer is, of course, Mary:

intgDoB

(*Name*	*DoB*	*Pa*	*Ma*)
Mary	**intg:**1934	Ted	Alice

If we can query, we can update.

```
update Child change DoB <- (strg)DoB
  using where
    xpose/att=quote DoB and xpose/typ=intg
    in Child;
```

This uses a cast to change Mary's integer date of birth to a string. Since *DoB* was declared as a type union of integer or string, this is legal; a cast to, say, **real** would not be, unless *DoB* was of type **any** or at least a union involving **real**. It also must **quote** the attribute *DoB* so that the name is used rather than values of *DoB*. (This defines the **quote** operator used above.)

Note that **transpose** operates on only one level. It does not automatically traverse all levels. If we want to do that, we must code the traversal explicitly, as in the recursive case in section 3.4.

3.2 Missing and multiple values

The next aspect of structure irregularity is that values can be absent, single or multiple. A nested relation handles all of these cases without any syntactic intervention (the former by being empty). However, since single values are often represented by scalar attributes without the extra machinery of nesting and subordinate attributes, we now show a couple of ways of updating an originally scalar attribute with (necessarily) a single value to a multi-valued nested relation.

Here is a relation with a scalar value for *Children*, as permitted by the union type declaration of *Children*.

domain *child* **strg**; **domain** *DoB* **intg**;
domain *Name* **strg**;
domain *chiln*(*Name, DoB*);
domain *Children child*|*chiln*
domain *Conj* **strg**; **domain** *Wed* **strg**;
relation *Family*(*Conj, Wed, Children*);

```
Family
(Conj   Wed    Children )
Alice   1932   child:Bernice
```

Now we add two more children. Just to make the point more strongly, these two children have, in addition, known dates of birth.

relation *Chiln*(*DoB, Name*) <−
  {(1934, "Mary"),(1935, "James")};
update *Family*/*Children* **add** *Chiln* ≡
update *Family* **change**
  update *Children* **add** *Chiln*;

```
Family
(Conj   Wed    Children                 )
Alice   1932   child:Bernice
               chiln:
               (DoB           Name)
               1934           Mary
               1935           James
```

The above result is not completely seamless, so let's use an update which replaces attributes instead of using union types.

domain *Conj* **strg**;
domain *Wed* **strg**;
domain *Child* **strg**;
relation *Family*(*Conj, Wed, Child*);

```
Family
(Conj   Wed    Child   )
Alice   1932   Bernice
```

We replace *Child* with a relation, *Children*:

let *Name* **be** *Child*;
let *Children* **be** **relation**(*Name*);
update *Family* **change**
  replace *Child* **with** *Children*;

Now we can add to *Children* the same relation, *Chiln*, we added above.

update *Family*/*Children* **add** *Chiln*

giving *Family*, shown to the right.

```
Family
(Conj   Wed    Children            )
Alice   1932   (Name)
               Bernice
               (DoB       Name)
               1934       Mary
               1935       James
```

We have here also added children with the extra attribute, *DoB*, so the resulting *Children* is a *polymorphic* relation. If *Chiln* had been defined only on *Name*, the resulting *Children* would have been a simple relation on *Name* with three tuples for Bernice, Mary and James.

3.3 Wildcards

Our discussion of regular expressions in section 2.1 did not mention the wildcard symbol, ".", which we are now able to implement. The **transpose** operator, above, detects only leaf attributes, and so was not used in section 3.2 on multiple values. We return to it now, and we will also need an operator, **transposeAll**(), which picks up *all* attributes of the relation it is evaluated in, whether they are scalars (leaves) or more deeply nested relations. Because of this extension, **transposeAll** does not take the argument of type **any** that we called *val* in section 3.1, but it does take the other two types of argument we discussed for **transpose**, with the same meanings.

We find

Person/./*Conj* ≡
[**red union of**
  [**red union of** *Conj*] **in**
.] **in** *Person*

in the relation,

```
Person
(Name   Family         Employer               )
        (Conj   Wed)   (Boss   Conj   Subord)
Ted     Alice   1932   Pete    Alan   Carole
```

Here, *Conj* may be taken to mean *spouse* in *Family* and *partner* in *Employer*. The important issue is that "." refers to a union of the nested attributes, *Family* and *Employer*.

We must find the nonleaf attributes, *Family* and *Employer*, of *Person*. **Transpose** finds only the leaf attributes (*Name* in this case), while the new operator **transposeAll**() applied to *Person* finds all three attributes. So we find the difference.

let *nonleaves* **be**
  transposeAll(*att*) **diff** **transpose**(*att*);

If applied to *Person*, this gives the set of nonleaf attributes of *Person*. Here it is actualized, with the rest of *Person* as above.

```
Person
(Name   Family   Employer)   nonleaves
        (..)     (..)        (att       )
                             Family
                             Employer
```

We next evaluate their names to give relations which we can take the union of (and raise the level so that this union is a direct attribute of *Person*).

let *FE* **be** [**red union of eval** *att*] **in** *nonleaves*;

Now we can implement "." in the above code.

Person/./*Conj* ≡ *Person*/*FE*/*Conj* ≡
[**red union of**
[**red union of** *Conj*] **in**
FE] **in** *Person*

Fixed-Depth Path Expression with Wildcard

The wildcard can now be used in regular expressions together with, say, the Kleene star.

Person//*Name* ≡ *Person*(/.)*/*Name* ≡
let *Nom* **be** *Name* **union**
[**red union of** *Nom*] **in** .;
[**red union of** *Nom*] **in** *Person*

Path Expression with Wildcard and Kleene Star

Wildcards are especially important for semistructured data, in which an implicit schema may not be explicitly known. In such cases, queries using wildcards work despite this ignorance. Furthermore, a Kleene star applied to a wildcard, as above, is now applicable in any nested relation even when the nesting is not recursive.

Simple modifications to the above implementation are needed if the wildcard symbol appears at the end of a path expression and so refers to all the leaf attributes. Lack of space obliges us to omit them.

3.4 Schema discovery

When a schema is not known, for instance because it is completely implicit in a marked-up document as we shall examine in the next section, 4, it becomes useful to be able to extract and represent it explicitly. Suppose we have data conforming to the recursive schema

```
Person
(Name   Family                                )
        (Conj   Children                      )
                (Name   Family              )
                        (Conj   Children)
                                (Name)
```

Here is simple code to discover this schema.

We introduce the domain algebra **self** operator: it returns, as a value for every tuple, the name of the relation it is evaluated in (or the lowest named relation in the levels above, if the relation and any of its immediate ancestors happen to be anonymous). This name is of type **attr**.

let *attrib* **be self**;

Now a recursive use of **transpose** extracts the structure.

let *schema* **be**
transpose(*attrib*) **union** [*attrib, schema*]
in .;
Schema <− [*attrib, schema*] **in** *Person*;

The result is

```
Schema
(attrib  schema                                                  )
         (attrib  schema                                         )
                  (attrib    schema                              )
                             (attrib   schema                    )
                                       (attrib     schema)
                                                   (attrib)
Person   Name
         Family   Conj
                  Children   Name
                             Family    Conj
                                       Children    Name
```

4 Markup and Data on the Web

Now that we can handle all useful operations on irregular and unspecified schemas, via path expressions and the more general underlying formalism which path expressions abbreviate, we can turn to data with fully implicit schema (which may account for a significant part of the size of the data). This data could be XML, but we allow ourselves a little more freedom in this exploratory work, and use a structure we call xML, where $x \in \{$G, SG, HT, X, ..$\}$. Here is an example, with facilities limited to the variants we have discussed in the last two sections.

```
<PERSON>
  <Name>Ted</Name> married
  <FAMILY><Conj>Alice</Conj> in
   <Wed>1932</Wed>. Their children,
    <CHILDREN><Name>Mary</Name>
     (<DoB>1934</DoB>) married
      <FAMILY><Conj>Alex</Conj> in
       <Wed type=intg>1954</Wed>
        (Mary and Alex had
         <CHILDREN label=J>Joe</Children>)
      </FAMILY>
      and <Name>James</Name>
       (<DoB>1935</DoB>) married
      <FAMILY><Conj>Jane</Conj>
        (<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>
```

This example shows nested tags, with text within the tags and with connecting text. There is one example of an "attribute" in the xML (not the relational) sense of the word: `<Wed type=intg>` specifies that the date for this tag is to have integer type (string is the default type for all text). (Further use of xML "attributes", for example, is in HTML, discussed below, or to create links and cross-references, not discussed in this paper.)

We would like to convert this text to the corresponding nested relation, shown in the box on the next page.

To do so, we introduce an operator, **mu2nest**(), which converts the top level of a marked-up text to a relation. (This relation is not nested—yet.)

nestPerson <− **mu2nest**() *PERSON*;

```
nestPerson
(Name   FAMILY                           )
Ted     <FAMILY><Conj>Alice</Conj>
        in ..  </FAMILY>
```

Mu2nest does not need to be recursive, because we can write recursive domain algebra. Here is the full conversion for the example, using mutual recursion.

let *Family* **be** [*Conj, Wed, Children*] **in**
mu2nest() *FAMILY*;
let *Children* **be** [*Name, DoB, Family*] **in**
mu2nest() *CHILDREN*;
Person <− [*Name, Family*] **in**
mu2nest() *PERSON*;

The figure at the top of the next page gives the result.

In full, **mu2nest**() is an attribute generator, as is **transpose**, and it takes up to three arguments. For example,

mu2nest(*content, start, length*)

given

Person									
(*Name*	*Family*								)
	(*Conj*	*Wed*	*Children*						)
			(*Name*	*DoB*	*Family*				)
					(*Conj*	*Wed*	*Children*	)	
							(*Name*	*DoB*)	
Ted	Alice	1932	Mary	1934	Alex	**intg**: 1954	Joe	1956	
			James	1935	Jane	**strg**: 1960	Tom	1961	
							Sue	1962	

domain *content* **any**;
domain *start* **intg**;
domain *length* **intg**;
will generate three attributes: *content*, which contains the content of the tag (usually a string, but we need type **any** to be safe); *start*, which holds the byte position in the text (ignoring all tags) of the first byte of the contents of the tag; and *length*, the number of bytes in the tag contents. (If only one argument of type integer is provided to **mu2nest**, it is taken to be the starting position.) Here is a one-level application of **mu2nest**(c, s, l) (abbreviating *content, start* and *length* in an obvious way because space is limited).
nestPerson <− **mu2nest**(c, s, l) *PERSON*;

nestPerson							
(*c*	*s*	*l*	*Name*			*FAMILY*	)
			(*c*	*s*	*l*)		
Ted .. (1962).	1	203	Ted	1	3	<FAMILY><Conj>Alice </Conj> in .. </FAMILY>	

This construct circumvents the fundamental mismatch that text is ordered while relations are order-independent: the *start* attribute explicitly records position. It also includes the text between tags, which **mu2nest**() did not.

HTML is a special case of mark-up syntax, but one which can be considered to be recursive mark-up. Each anchor tag leads to a complete new HTML document, and this can be thought of as nested within the parent document. Such thinking brings the Web to the same level of abstraction as relations. Relational programming ignores the boundary between primary memory and secondary storage by simply considering all relations to be accessible without special operations that the programmer must invoke to fetch or store. Making the Web a single, enormous, complexly-nested document ignores the boundary between combined RAM and SS on one hand and the Internet on the other.

Here is an example.

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

Here is its schema, generated by **mu2nest**(*content*), with tag A recursively referring, indirectly, to itself.
relation *HTML*(*Head, Body*);
domain *Head*(*Title*);
domain *BodyRel*(*content, A, img*);
domain *Body* **strg**|*BodyRel*;
domain *A*(*content,href,Head,Body,Name,img*);
domain *img*(*src, width*);
All the processing discussed up to here in this paper now applies to the Web.

5 Conclusions

We have developed hierarchies and flexible schemata for relational data so as to support the central characteristics of semistructured data, namely path expressions and irregular structure. We have provided an operator to translate XML-like marked-up text into this representation.

The work is presented here as a development of relational technology with semistructured data as the goal. In fact, only the **mu2nest** operator is specific to semistructured data. The first section, on path expressions, is merely an obvious completion of relational ideas independent of possible application to semistructured data: it is arbitrary to limit attributes or the operations of the domain algebra to scalars, so nested relations are but the liberation from this restriction; it is arbitrary to limit nesting to fixed depth or the domain algebra to being non-recursive, so recursive nesting results; path expressions are the appropriate syntactic shorthand for the expressions in the resulting formalism that are most often used; and given these capabilities, and path expressions, for the projection operator, the selection and update operators also demand attention. We have shown, in boxes, the relational equivalents of each of these types of path expression (but we have not discussed the formal syntactic transformation needed to map path expressions to the underlying implementation).

The second section, on irregular and unknown schemata, introduces unions and hierarchies of types, and these are important capabilities of programming languages in general. The wildcard symbol in path expressions requires unions of unspecified alternatives and is not specifically semistructured. The **transpose** operators were originally developed to generate metadata to implement association-rule data mining, and are here shown to apply to metadata describing semistructured schemata.

Hence, we have shown that semistructured data is a special case of relational data via facilities which would complete relational programming in any case, motivated by general and accepted programming language principles. Path expressions, and hence languages such as XML, are strictly special cases of the the operations provided by a completed relational programming language, as our boxed examples have shown. The only new syntax introduced, apart from the sugar for path expressions and **mu2nest**, is the **relation** operator, **transpose** and **transposeAll**, the wildcard symbol, and **self**.

6 Acknowledgements

We are indebted to the Natural Science and Engineering Research Council of Canada for support under grant OGP0004365. Most of the facilities introduced in this paper have been implemented in our relational system by Zhan Yu (recursive nesting and syntactic sugar) and Fan Guo (metadata). Jiantao Xie and Yu Gu have gone further and implemented the text and pattern-matching facilities not covered in this paper.

References

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

Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G. & Simeon, J. (1997), 'Querying documents in object databases', *Internat. J. on Digital Libraries* **1**(1), 5–19.

Blake, G. E., Consens, M. P., Kilpeläinen, P., Larson, P.-Å., Snider, T. & Tompa, F. W. (1994), Text/relational database management systems: Harmonizing SQL and SGML, *in* W. Litwin & T. Risch, eds, 'First Internat. Conf. on Applications of Databases, ADB', Springer-Verlag, LNCS 819, Vadstena, Sweden, pp. 267–80.

Buneman, O. P. (1997), Semistructured data, *in* 'Proc. 16th Internat. Symp. on Principles of Database Systems PODS', ACM Press, Tucson, Arizona, pp. 117–121.

Buneman, O. P., Davidson, S., Hillebrand, G. & Suciu, D. (1996), A query language and optimization techniques for unstructured data, *in* H. V. Jagadish, ed., 'Proceedings of ACM SIGMOD International Conference on Management of Data', ACM, Montreal, Canada, pp. 505–16.

Buneman, O. P., Davidson, S. & Suciu, D. (1995), Programming constructs for unstructured data, *in* P. Atzeni & V. Tannen, eds, 'Proc. Fifth Internat. Workshop on Database Programming Languages, DBPL-5', Springer-Verlag, Gubbio, Italy, p. 12.

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

Consens, M. P. & Mendelzon, A. O. (1990), Graphlog: a visual formalism for real-life recursion, *in* 'Proc. of the Nineth ACM Symp. on Principles of Database Systems, PODS', ACM Press, Nashville, Tennessee, pp. 404–16.

Cruz, I. F., Mendelzon, A. O. & Wood, P. T. (1987), A graphical query language supporting recursion, *in* U. Dayal & I. L. Traiger, eds, 'Proc. of the ACM Internat. Conf. on Management of Data, SIGMOD'87', ACM Press, San Francisco, pp. 323–30.

Cruz, I. F., Mendelzon, A. O. & Wood, P. T. (1988), G+: Recursive queries without recursion, *in* L. Kershberg, ed., 'Proc. of the Second Internat. Conf. on Expert Database Systems', Tysons Corner, Va., pp. 355–68.

Lakshmanan, L. V. S., Sadri, F. & Subramanian, I. N. (1996), A declarative language for querying and restructuring the WEB, *in* 'Proc. of the Sixth Internat. Workshop on Research Issues in Data Engineering, RIDE-NDS', New Orleans, pp. 12–21.

Mendelzon, A. O., Mihaila, G. A. & Milo, T. (1997), 'Querying the world wide web', *Int. J. on Digital Libraries* **1**(1), 54–67. (Also *Proc. PDIS*, 1996).

Merrett, T. H. (1984), *Relational Information Systems*, Reston Publishing Co., Reston, VA.

Merrett, T. H. (1999), Relational information systems. (revisions of (Merrett 1984)):
Data structures for secondary storage http://www.cs.mcgill.ca/~cs420
Database programming:
http://www.cs.mcgill.ca/~cs612.

Merrett, T. H. (2005*a*), CS++: Reinventing computer science (for secondary storage), Technical Report TR-Ald05-01, School of Computer Science, McGill University, Montreal. URL: www.cs.mcgill.ca/~tim/CS++.pdf (750K).

Merrett, T. H. (2005*b*), A nested relation implementation for semistructured data, Technical Report TR-Ald05-03, School of Computer Science, McGill University, Montreal. URL: www.cs.mcgill.ca/~tim/semistruc/recnest.ps.gz.

Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J. & Widom, J. (1995), Querying semistructured heterogeneous information, *in* T. W. Ling, A. O. Mendelzon & L. Vieille, eds, 'Proc. Fourth Internat. Conf. on Deductive and Object-Oriented Databases, DOOD', Springer-Verlag, LNCS 1013, Singapore, pp. 319–44.

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

Suciu, D. (1998), 'An overview of semistructured data', *SIGACT News* **29**(4), 28–38.

Thierry-Mieg, J. & Durbin, R. (1992), acedb—a C.elegans database: Syntactic definitions for the ACeDB data base manager. www.acedb.org/Cornell/syntax.html.