Lectures 1 and 2: Generalising Relational Algebra and Programming with Collection Types

Peter Buneman

August 2002

Generic Programming Summer School

Outline

Lectures 1&2 Establish the connection between traditional database query languages (relational algebra, datalog, f.o. logic) and functional programming paradigms (structural recursion, monads)

Lectures 3&4 Development of languages for semistructured data and XML

Background: Relational Database Query Languages

Relational databases have dominated the “market” for 20 years. Why?

- The relational model is a simple abstraction of data, and relational algebra is the interface. Often called a “logical” model.
- The relational algebra is simple and its operations can be efficiently implemented.
- There is a close relationship between relational algebra and first-order logic – clear semantics.
- Relational algebra allows rewriting – optimization.
- Relational decomposition may help in transaction processing.

The relational model and algebra

Munros:

MId	MName	Lat	Long	Height	Rating
1	The Saddle	57.167	5.384	1010	4
2	Ladhar Bheinn	57.067	5.750	1020	4
3	Schiehallion	56.667	4.098	1083	2.5
4	Ben Nevis	56.780	5.002	1343	1.5

Hikers:

HId	HName	Skill	Age
123	Edmund	EXP	80
214	Arnold	BEG	25
313	Bridget	EXP	33
212	James	MED	27

Climbs:

HId	MId	Date	Time
123	1	10/10/88	5
123	3	11/08/87	2.5
313	1	12/08/89	4
214	2	08/07/92	7
313	2	06/07/94	5

The Schema

```
CREATE TABLE Hikers (    HId      INTEGER,
                         HName    CHAR(30),
                         Skill    CHAR(3),
                         Age      INTEGER,
                         PRIMARY KEY (HId) )

CREATE TABLE Climbs (    HId      INTEGER,
                         MId      INTEGER,
                         Date     DATE,
                         Time     INTEGER,
                         PRIMARY KEY (HId, MId),
                         FOREIGN KEY (HId) REFERENCES Hikers(HId),
                         FOREIGN KEY (MId) REFERENCES Munros(MId)  )
```

Updates that violate key constraints are rejected.

Relational databases are in first normal form. The entries in a table are “atomic” types.

Schemas consists of types and constraints. Without the key and inclusion (foreign key) constraints, the schema looks much like a record (struct) type declaration.

Relational Algebra

Six operations all, of which are functions that create tables from existing tables.

Three operations – union, difference, and selection – are familiar operations on sets.

$$\begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 4 & 5 \end{array} \cup \begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 5 & 6 \end{array} = \begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 4 & 5 \\ 5 & 6 \end{array}$$

$$\begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 4 & 5 \end{array} \setminus \begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 5 & 6 \end{array} = \begin{array}{c|c} A & B \\ \hline 4 & 5 \end{array}$$

$$\sigma_{A=1 \vee B<A} \left(\begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 4 & 5 \\ 4 & 1 \end{array} \right) = \begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 4 & 1 \end{array}$$

Projection and Product

projection extracts columns:

$$\pi_{\{A,C\}}\left(\begin{array}{c|c|c} A & B & C \\ \hline 1 & 3 & 6 \\ 4 & 5 & 7 \\ 4 & 1 & 7 \end{array}\right) = \begin{array}{c|c} A & C \\ \hline 1 & 6 \\ 4 & 7 \end{array}$$

product (not quite cartesian product):

$$\begin{array}{c|c} A & B \\ \hline 1 & 3 \\ 2 & 3 \end{array} \times \begin{array}{c|c} C & D \\ \hline 3 & 31 \\ 4 & 41 \\ 5 & 51 \end{array} = \begin{array}{c|c|c|c} A & B & C & D \\ \hline 1 & 3 & 3 & 31 \\ 2 & 3 & 3 & 31 \\ 1 & 3 & 4 & 41 \\ 2 & 3 & 4 & 41 \\ 1 & 3 & 5 & 51 \\ 2 & 3 & 5 & 51 \end{array}$$

Also common

natural join: columns with the same label are identified.

renaming (column relabelling)

Why Relational Algebra?

- Efficient implementation of individual operations, especially join.
- Query rewriting

$$\sigma_{C \wedge D}(R) \times S = \sigma_C(R) \times S \cup \sigma_D(R) \times S$$

- “Equivalence” with f.o. logic:

$$\{x \mid \forall y(\exists z(R(y,z) \to R(x,z)))\} = \pi_A(R) \setminus \pi_A(\pi_A(R) \times \pi_B(R) \setminus R)$$

- But $\{x \mid \neg(R(x,x))\} = \quad ???$

Why not relational algebra?

- First normal form condition is annoying. Why not freely combine set and tuple (record) types?
- Why not lists and multisets? (Some multiset operations are available in SQL)

Example – Swissprot (one entry)

```
ID   11SB_CUCMA     STANDARD;      PRT;   480 AA.
AC   P13744;
DT   01-JAN-1990 (REL. 13, CREATED)
DT   01-JAN-1990 (REL. 13, LAST SEQUENCE UPDATE)
DT   01-NOV-1990 (REL. 16, LAST ANNOTATION UPDATE)
DE   11S GLOBULIN BETA SUBUNIT PRECURSOR.
OS   CUCURBITA MAXIMA (PUMPKIN) (WINTER SQUASH).
OC   EUKARYOTA; PLANTA; EMBRYOPHYTA; ANGIOSPERMAE; DICOTYLEDONEAE;
OC   VIOLALES; CUCURBITACEAE.
RN   [1]
RP   SEQUENCE FROM N.A.
RC   STRAIN=CV. KUROKAWA AMAKURI NANKIN;
RX   MEDLINE; 88166744.
RA   HAYASHI M., MORI H., NISHIMURA M., AKAZAWA T., HARA-NISHIMURA I.;
RL   EUR. J. BIOCHEM. 172:627-632(1988).
RN   [2]
RP   SEQUENCE OF 22-30 AND 297-302.
RA   OHMIYA M., HARA I., MASTUBARA H.;
RL   PLANT CELL PHYSIOL. 21:157-167(1980).
CC   -!- FUNCTION: THIS IS A SEED STORAGE PROTEIN.
CC   -!- SUBUNIT: HEXAMER; EACH SUBUNIT IS COMPOSED OF AN ACIDIC AND A
CC       BASIC CHAIN DERIVED FROM A SINGLE PRECURSOR AND LINKED BY A
CC       DISULFIDE BOND.
CC   -!- SIMILARITY: TO OTHER 11S SEED STORAGE PROTEINS (GLOBULINS).
```

Swissprot continued

```
DR   EMBL; M36407; G167492; -.
DR   PIR; S00366; FWPU1B.
DR   PROSITE; PS00305; 11S_SEED_STORAGE; 1.
KW   SEED STORAGE PROTEIN; SIGNAL.
FT   SIGNAL        1     21
FT   CHAIN        22    480       11S GLOBULIN BETA SUBUNIT.
FT   CHAIN        22    296       GAMMA CHAIN (ACIDIC).
FT   CHAIN       297    480       DELTA CHAIN (BASIC).
FT   MOD_RES      22     22       PYRROLIDONE CARBOXYLIC ACID.
FT   DISULFID    124    303       INTERCHAIN (GAMMA-DELTA) (POTENTIAL).
FT   CONFLICT     27     27       S -> E (IN REF. 2).
FT   CONFLICT     30     30       E -> S (IN REF. 2).
SQ   SEQUENCE   480 AA;  54625 MW;  D515DD6E CRC32;
     MARSSLFTFL CLAVFINGCL SQIEQQSPWE FQGSEVWQQH RYQSPRACRL ENLRAQDPVR
     RAEAEAIFTE VWDQDNDEFQ CAGVNMIRHT IRPKGLLLPG FSNAPKLIFV AQGFGIRGIA
     IPGCAETYQT DLRRSQSAGS AFKDQHQKIR PFREGDLLVV PAGVSHWMYN RGQSDLVLIV
     FADTRNVANQ IDPYLRKFYL AGRPEQVERG VEEWERSSRK GSSGEKSGNI FSGFADEFLE
     EAFQIDGGLV RKLKGEDDER DRIVQVDEDF EVLLPEKDEE ERSRGRYIES ESESENGLEE
     TICTLRLKQN IGRSVRADVF NPRGGRISTA NYHTLPILRQ VRLSAERGVL YSNAMVAPHY
     TVNSHSVMYA TRGNARVQVV DNFGQSVFDG EVREGQVLMI PQNFVVIKRA SDRGFEWIAF
     KTNDNAITNL LAGRVSQMRM LPLGVLSNMY RISREEAQRL KYGQQEMRVL SPGRSQGRRE
//
```

If fully normalized, Swissprot requires 20 or more tables.

Artificial identifiers need to be introduced

Most queries require joins

Lists are needed (and arrays?)

Before leaving relational algebra

It has been well studied, but there are still some interesting open issues to do with finding "generic" types for the operations (natural join and relabelling are the problems)

Didier Rémy, Typing Record Concatenation for Free. In Nineteenth Annual Symposium on Principles Of Programming Languages, pages 166–176, 1992.

Peter Buneman and Atsushi Ohori. Polymoprhism and Type Inference in Database Programming. ACM Transactions on Database Systems, March 1996.

Jan Van den Bussche, Emmanuel Waller: Type Inference in the Polymorphic Relational Algebra. PODS 1999 : 80-90

From structural recursion to database queries.

A relation can be taken as a set of records. An alternative approach is therefore to study the operations associated with these two data types.

The operations for records are well-known.

Record construction: $[\text{Name} = \text{"Joe"},\ \text{SS\#} = 123456789,\ \text{Dept} = \text{"Sales"}]$

Record decomposition (field selection): $r.\text{Dept}$

Together with the equations:

- $[\ldots, l_i = x, \ldots].l_i = x$
- $[l_1 = r.l_1, \ldots, l_n = r.l_n] = r$

For sets, the problem is more subtle.

Two choices of primitives for set construction are

"Insert presentation":	Empty set:	$\{\}$
	Insertion:	$x \nearrow S$
"Union presentation":	Empty set:	$\{\}$
	Singelton:	$\{x\}$
	Union:	$S_1 \cup S_2$

Primitives to decompose sets should follow the construction primitives, by structural recursion.

Example: using the "insert presentation", find the maximum of a set of natural numbers:

$$\begin{array}{lll} fun\ \ \mathit{set\_max}(\{\}) & = & 0 \\ \mid\ \ \mathit{set\_max}(x \nearrow S) & = & \mathit{max}(x, \mathit{set\_max}(S)) \end{array}$$

$$\mathit{set\_max} : \{\mathit{nat}\} \longrightarrow \mathit{nat}$$

Example: counting a set using the "union presentation" (?)

$$\begin{array}{lll} fun\ \ \mathit{count}(\{\}) & = & 0 \\ \mid\ \ \mathit{count}\{x\} & = & 1 \\ \mid\ \ \mathit{count}(S_1 \cup S_2) & = & \mathit{count}(S_1) + \mathit{count}(S_2) \end{array}$$

$$\mathit{count} : \{\tau\} \longrightarrow \mathit{nat}$$

From which

$$1 \;=\; \mathit{count}(\{x\}) \;=\; \mathit{count}(\{x\} \cup \{x\}) \;=$$
$$\mathit{count}(\{x\}) + \mathit{count}(\{x\}) \;=\; 1 + 1 \;=\; 2 \text{ !!!}$$

Conditions that ensure well-defined programs on sets

For the “insert presentation”:

$$\begin{array}{llcll} fun & g(\{\}) & = & e & g : \{\sigma\} \to \tau \\ \mid & g(x \nearrow S) & = & i(x, g(S)) & \text{when } e : \tau \text{ and } i : \sigma \times \tau \to \tau \end{array}$$

Notation: $g = sri(i, e)$.

This is well defined when i is commutative [$i(x, i(y, S)) = i(y, i(x, S))$] and idempotent [$i(x, (x, S)) = i(x, S)$].

For the “union presentation”:

$$\begin{array}{llcll} fun & h(\{\}) & = & e & h : \{\sigma\} \to \tau \\ \mid & h(\{x\}) & = & f(x) & \text{when } e : \tau,\ f : \sigma \to \tau, \\ \mid & h(c_1 \cup c_2) & = & u(h(c_1), h(c_2)) & \text{and } u : \tau \times \tau \to \tau \end{array}$$

Notation: $h = sru(u, f, e)$

This is well defined if (τ, u, e) is a commutative, idempotent monoid.

Examples

Use *sru*(u, f, e) for “union representation” structural recursion.

$$\begin{aligned}
\textit{map}\ f &= \textit{sru}(\cup, \lambda x.\{x\},) \\[2em]
\textit{flatten} &= \textit{sru}(\cup, \textit{id},) \\[2em]
\textit{pairwith}(x, S) &= \textit{map}\ (\lambda y.(x, y))\ S \\[2em]
\textit{cartprod}(S_1, S_2) &= \textit{flatten}(\textit{map}\ (\lambda x.\textit{pairwith}(x, S_2))\ S_1) \\[2em]
\textit{powerset} &= \textit{sru}((\lambda p.\textit{map} \cup (\textit{cartprod}\ p)), (\lambda x.\{x\}), \{\{\}\})
\end{aligned}$$

All these generalize to bags and lists.

A Natural Fragment of Structural Recursion

This limited form of structural recursion is always well-defined:

$$\begin{array}{llcl} fun & h(\{\}) & = & \{\} \\ | & h(\{x\}) & = & f(x) \\ | & h(c_1 \cup c_2) & = & h(c_1) \cup h(c_2) \end{array}$$

Call this $\mathit{ext}(f)$. Equivalently, $\mathit{ext}(f) = \mathit{sru}(\cup, f, \{\})$

We can Build a language using:

- For sets: $\{\}$, $\{x\}$, $S_1 \cup S_2$, $\mathit{ext}(f)S$
- For records: Formation and field selection.
- Lambda abstraction, but only over variables that represent complex objects. I.e. no “higher order” abstraction.

To simplify things, use pairs rather than records (which can be simulated by nesting pairs). Our complex-object types are given by:

$$\tau ::= b \mid \mathit{unit} \mid \tau \times \tau \mid \{\tau\}$$

where b ranges over base types, and *unit* is the “nullary” product, inhabited only by $()$. Note that this allows nested sets.

We have seen how cartesian product can be implemented with these primitives. So can relational projection:

$$\Pi_i\ R = \mathit{map}\ (\pi_i)\ R$$

Using $\{\}$ and $\{()\}$, the two values of type, $\{unit\}$, to represent *false* and *true*, respectively, we can implement selection.

$$\mathit{select}\ (p)\ S = \mathit{flatten}(\mathit{map}\ (\lambda x.\Pi_1(\mathit{cartprod}(\{x\},\ p\ x)))\ S)$$

We have all the operations of the relational algebra except *difference*.

A calculus – $\mathcal{MC}$

Variables and constants

$$\frac{}{x : \mathit{Type}(x)} \qquad \frac{}{c : \mathit{Type}(c)} \qquad \frac{}{p : \mathit{DType}(p) \longrightarrow \mathit{CType}(p)}$$

Abstraction and application

$$\frac{e : \tau}{\lambda x.e : \mathit{Type}(x) \longrightarrow \tau} \qquad \frac{e_1 : \sigma \longrightarrow \tau \quad e_2 : \sigma}{e_1 e_2 : \tau}$$

Pairing

$$\frac{e_1 : \sigma \quad e_2 : \tau}{(e_1, e_2) : \sigma \times \tau} \qquad \frac{e : \sigma \times \tau}{\pi_1 e : \sigma \qquad \pi_2 e : \tau} \qquad \frac{}{() : \mathit{unit}}$$

Sets

$$\frac{e : \tau}{\{e\} : \{\tau\}} \qquad \frac{e : \sigma \longrightarrow \{\tau\}}{\mathit{ext}(e) : \{\sigma\} \longrightarrow \{\tau\}} \qquad \frac{}{\{\}_\tau : \{\tau\}} \qquad \frac{e_1 : \{\tau\} \quad e_2 : \{\tau\}}{e_1 \cup e_2 : \{\tau\}}$$

c ranges over primitive constants with o-type $\mathit{Type}(c)$

p ranges over primitive functions with type $\mathit{DType}(p) \longrightarrow \mathit{CType}(p)$

Σ – The signature of primitive constants and functions.

$\mathcal{MC}(\Sigma)$ – the language over this signature.

A Monad "Algebra" – $\mathcal{MA}(\Sigma)$

$$\frac{}{Kc : \textit{unit} \to \textit{Type}(c)} \qquad \frac{}{p : \textit{DType}(p) \to \textit{CType}(p)}$$

$$\frac{f : \sigma \to \tau \quad g : \tau \to \upsilon}{g \circ f : \sigma \to \upsilon} \qquad \frac{}{\textit{id}_\sigma : \sigma \to \sigma}$$

$$\frac{f_1 : \sigma \to \tau_1 \quad f_2 : \sigma \to \tau_2}{(f_1, f_2) : \sigma \to (\tau_1 \times \tau_2)} \qquad \frac{}{\textit{fst}_{\sigma,\tau} : \sigma \times \tau \to \sigma} \qquad \frac{}{\textit{snd}_{\sigma,\tau} : \sigma \times \tau \to \tau}$$

$$\frac{f : \sigma \to \tau}{\textit{map}\ f : \{\sigma\} \to \{\tau\}} \qquad \frac{}{\mathrm{sng}_\tau : \tau \to \{\tau\}} \qquad \frac{}{\mathrm{flatten}_\tau : \{\{\tau\}\} \to \{\tau\}}$$

$$\frac{}{\rho 2_{\sigma,\tau} : \sigma \times \{\tau\} \to \{\sigma \times \tau\}} \qquad \frac{}{t_\tau : \tau \to \textit{unit}}$$

$$\frac{}{K\{\} : \textit{unit} \to \{\tau\}} \qquad \frac{}{\textit{union} : \{\tau\} \times \{\tau\} \to \{\tau\}}$$

$$\left[\frac{}{Kx : \textit{unit} \to \textit{Type}(x)}\right]$$

Theorem. For any signature Σ, $\mathcal{MC}(\Sigma) \simeq \mathcal{MA}(\Sigma)$ [$\stackrel{\mathrm{def}}{=}$ $\mathcal{M}(\Sigma)$]

Theorem. Set intersection is not definable in $\mathcal{M}()$.

Theorem. For any signature Σ,

$$\mathcal{M}(\cap, \Sigma) \simeq \mathcal{M}(=, \Sigma) \simeq \mathcal{M}(\text{difference}, \Sigma) \simeq$$

$$\mathcal{M}(\subseteq, \Sigma) \simeq \mathcal{M}(\in, \Sigma) \simeq \mathcal{M}(\text{nest}, \Sigma)$$

Theorem. Every query expressible in $\mathcal{M}(=)$ can be computed in polynomial time with respect to input size. Hence powerset $\notin \mathcal{M}(=)$

Claim. $\mathcal{M}(=, \Sigma)$ is the “right” nested relational algebra.

Theorem (Wong; Paredaens& Van Gucht). $\mathcal{M}(=)$ is a conservative extension of flat relational algebra. Hence parity, transitive closure $\notin \mathcal{M}(=)$

Let us use $\mathcal{NRA}$ for $\mathcal{MA}(=)$

Further use of Structural Recursion

It is easy to define $R_1 \circ R_2$, the composition of R_1 and R_2 in $\mathcal{NRA}$

Defining $i : (\alpha \times \alpha) \times \{\alpha \times \alpha\} \rightarrow \{\alpha \times \alpha\}$ as

$$i(r, T) = \{r\} \cup T \cup \{r\} \circ T \cup T \circ \{r\} \cup T \circ \{r\} \circ T$$

gives us transitive closure:

$$\begin{array}{lllll} \textit{fun} & \textit{TC}(\{\}) & = & \{\} \\ \mid & \textit{TC}(s \nearrow R) & = & i(s, \textit{TC}(R)) \end{array}$$

We have to check that i satisfies the idempotence and commutativity conditions for this form of structural recursion.

Warshall's algorithm can be defined in a similar fashion. With some extra manipulation, efficient implementations of these algorithms can be derived.

Powerset

We have seen that *powerset* is definable with *sru*

The Abiteboul and Beeri algebra ($\mathcal{A}\&\mathcal{B}$)is obtained by adding powerset operator to a nested relational calculus.

It can express equal cardinality, parity and transitive closure.

Let C be a signature of object types, i.e. no functions.

Theorem. $\mathcal{A}\&\mathcal{B}(C) \simeq \mathcal{M}(=, powerset, C) \simeq \mathcal{SR}(=, C)$

The proof of this relies on our well-definedness conditions for $\mathcal{SR}(C)$

However,

Theorem. There are (very simple) signatures Σ for which $SR(\Sigma)$ cannot be polymorphically translated into $\mathcal{A}\&\mathcal{B}(\Sigma)$.

Also, transitive closure, expressed in $\mathcal{A}\&\mathcal{B}(C)$, requires the use of *powerset*. Any algorithm to express transitive closure in $\mathcal{A}\&\mathcal{B}$ requires exponential space [Suciu&Paredaens, PODS'94]

Connections with Other Languages

Over complex objects fixpoints (inflationary, partial) can compute powerset.

However we can restrict the expressive power of a fixpoint operator by *bounding* its output.

$$\frac{f : \{\sigma\} \to \{\sigma\} \qquad B : \{\sigma\}}{\textit{bfix}(f, B) : \{\sigma\} \to \{\sigma\}}$$

$$\textit{bfix}(f, B) = \textit{fix}(g) \text{ where } g(S) = f(S) \cap B$$

$\mathcal{NRA} + \textit{bfix}$ is conservative over $\textit{FO} + \textit{fix}$ (inflationary Datalog).

Hence $NRA + \textit{bfix}$ cannot compute parity.

- $\mathcal{NRA}$ + rational arithmetic + aggregate summation ($\overset{\text{def}}{=}$ NRA_Q) is conservative over its first-order fragment (Libkin & Wong).
- The following languages
  - NRA_Q + transitive closure + linear order
  - NRA_Q + bounded fixed point + linear order (inflationary or partial semantics)

  are conservative over their respective first-order fragments (Libkin & Wong).
- NRA_Q + powerset + linear order is conservative over its second-order fragment (Libkin & Wong).

Bag Languages

Nested Bag Algebra is defined in the same way as NRA, but bag semantics are used.

$\mathcal{BQL} \overset{\text{def}}{=}$ Nested Bag Algebra + monus + unique

Results for bag languages:

1. $\mathcal{BQL}$ + (insert) structural recursion expresses exactly the class of all primitive recursive functions (Libkin & Wong).

2. $\mathcal{BQL}$ + powerbag expresses exactly the class of all Kalmar-elementary functions (Libkin & Wong).

Comprehensions

Wadler has shown a nice connection between “comprehensions” and the operations of NRA.

Comprehensions “look like” Zermello-Fraenkel set notation. They look even more like practical database query langauges.

They can be interpreted for sets, bags, lists and, ...

They can be used with ML-style pattern matching, and better

They can be transformed into NRA using rewrite rules such as

$$\{e' \mid x \leftarrow e \ldots\} \quad \rightsquigarrow \quad \textit{ext}(\lambda x.\{e' \mid \ldots\})\, e$$

$$\{e' \mid\} \quad \rightsquigarrow \quad \{e\}$$

Optimizations arise systematically from categorical descriptions, and are best exploited using the syntax of comprehensions [Wong, PhD thesis]. Examples ($\mu = $ flatten):

For all collections:

- $\mu\{e_1 \mid x \leftarrow \{e_2\}\} \rightsquigarrow e_1[e_2/x]$
- $\mu\{\{x\} \mid x \leftarrow S\} \rightsquigarrow S$
- $\mu\{e_1 \mid x \leftarrow \mu\{e_2 \mid y \leftarrow e_3\}\} \rightsquigarrow \mu\{\mu\{e_1 \mid x \leftarrow e_2\} \mid y \leftarrow e_3\}$
  (vertical loop fusion)

For sets and bags:

- $\mu\{e_1 \mid x \leftarrow e\} \cup \mu\{e_2 \mid x \leftarrow e\} \rightsquigarrow \mu\{e_1 \cup e_2 \mid x \leftarrow S\}$
  (horizontal loop fusion)

From this last equation one can derive that $\{\tau + \sigma\} \cong \{\tau\} \times \{\sigma\}$. This is I believe,one of the reasons for the usefulness of relational databases.

An application – NCBI's GenBank

GenBank, the most comprehensive source of biosequence information, is distributed in ASN.1 (Abstract Syntax Notation) format.

This is a "structured file"; it is not a database.

ASN.1 terminology	Standard terminology	Our notation
sequence of	list	$[\tau]$
set of	set	$\{\tau\}$
sequence	record	$(l_1 : \tau_1 \ldots l_n : \tau_n \;)$
set	tuple ??	$\tau_1 * \ldots * \tau_n$
choice	variant	$<< \; l_1 : \tau_1 \ldots l_n : \tau_n \; >>$

An ASN.1 type (part of GenBank):

```
[(em:Date, cit:Cit-art, gene:{string}, ...)]
    where Cit-art = (title:  string, authors:  Auth-list, ...
          Auth-list = [(name:string,...)]
```

A sample query:

```
{[title = x.cit.title, gene = x.gene]|
            \x <- Medline-data;
            x.em.year = 1989;
            [name = "J.Doe", ...]  <- x.cit.authors}
```

c.f. SQL - (as it should be!)

```
    SELECT title = x.cit.title, gene = x.gene
    FROM   Medline-data x
    WHERE  x.em.year = 1989
    AND    "J.Doe" IN
           SELECT Name
           FROM x.cit.authors
```

Another example, involving variants:

```
{[abstract = x.abstract, volume = v]|
     \x <- Medline-data;
     x.em.year = 1989;
     <<journal = [title = [name = "J.Irrep.Res", ...],
                        imprint = [vol = \v,...].  ...]>
         <- x.cit.from}
```

Further Reading

Serge Abiteboul, Richard Hull and Victor Vianu, *Foundations of Databases.* Addison-Wesley, 1995.

Peter Buneman, Shamim A. Naqvi, Val Tannen, Limsoon Wong: Principles of Programming with Complex Objects and Collection Types. TCS 149(1): 3-48 (1995)

L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In ACM SIGMOD International Conference on Management of Data, San Jose, California, pp 47-58, May 1995.

The Penn web site: http://db.cis.upenn.edu

R. G. G. Cattell *at al.*, The Object Data Standard 3.0. Morgan Kaufmann (2000)

Lectures 3 and 4:
From Semistructured Data to XML

Peter Buneman

August 22, 2002

Generic Programming Summer School

Motivation

Some data really is unstructured. Examples:

- The World-Wide Web
- Data exchange formats
- ACeDB – a database used by biologists.

Motivation – the Web

Why do we want to treat the Web as a database?

- To maintain integrity
- To query based on *structure* (as opposed to content)
- To introduce some “organization”.

But the Web has no structure. The best we can say is that it is an enormous graph.

Motivation – Data Formats

Much (probably most) of the world's data is in data formats. These are formats defined for the interchange and archiving of data.

Data formats vary in generality. ASN.1 and XDR are quite general. Scientific data formats tend to be "fixed schema" (NetCDF is an exception.)

The textual representation given by data formats is sometimes not immediately translatable into a standard relational/object-oriented representation.

Some examples of structured text and data formats

```
Identification_Information:
     Citation:
          Citation_Information:
               Originator: OL-A, Air Force Combat Climatology Center (AFCCC)
               Originator: Air Force Global Weather Central (AFGWC) (comp)
               Publication_Date: 19960621
               Title: PIBAL - Upper Air Pilot Balloon Observations (PIBAL)
               Publication_Information:
                    Publication_Place: ASHEVILLE, NC
                    Publisher: OL-A, AFCCC
     Description:
          Abstract: The PIBAL database includes rawinsonde, pilot   ...
     Spatial_Domain:
          Bounding_Coordinates:
               West_Bounding_Coordinate: -180.0000000000
               East_Bounding_Coordinate: 180.0000000000
               North_Bounding_Coordinate: 90.0000000000
               South_Bounding_Coordinate: -90.0000000000
          Stratum:
               Stratum_Keyword_Thesaurus: None
               Stratum_Keyword: Troposphere
               Stratum_Keyword: Stratosphere
               Stratum_Keyword: Mesophere
```

Another example: ACeDB

ACeDB (A C. elegans Database) is popular with biologists for its flexibility and its ability to accommodate missing data.

An ACeDB schema (with some liberties):

```
person name firstname unique string              — at most one first name
            lastname unique string               — at most one last name
            tel int                              — several numbers

book authors person                              — means set of persons
     title unique string                         — at most one title
     chapter-headings int unique string          — an array of strings
...
```

Some ACeDB data

```
ASmith person name firstname "Alan"          --- ASmith is key/OID
                   lastname "Smith"

LH17.23.15  book authors ASmith
                         JDoe
                 title "A very brief history of time"
                 chapter-headings  1    "The Beginning"
                                   2     "The Middle"
                                   3     "The End"

GK12.23.45 book authors "K. Ludwig"
           ...
```

ACeDB continued

An ACeDB type is an infinite tree, and an instance as a finite subtree of the type.

In fact ACeDB has a parameterized type list. list(int) stands for int int int ... – an infinitely branching, infinitely deep, tree

An example of an instance of list(int):

2	3	2	4
	2	5	
	4	1	
4	2	7	
3	1		
	2		

Although ACeDB has a schema (and might not be regarded as semistructured) the schema only places rather weak “outer bounds” on the data.

A format for data exchange – Tsimmis

The Object Exchange Model provides a syntax for describing objects.

It describes a flexible data structure in which many other conventional data structures may be represented.

$\langle$bib, set, $\{\mathit{doc}_1, \mathit{doc}_2 \ldots \mathit{doc}_n\}\rangle$

doc_1 : $\langle$doc,set, $\{\mathit{au}_1, \mathit{top}_1, \mathit{cn}_1\}\rangle$

au_1 : $\langle$authors, set, $\{\mathit{au}_1^1\}\rangle$

au_1^1 : $\langle$author-ln, string, “Ullman”$\rangle$

top_1 : $\langle$topic, string, “Databases”$\rangle$

cn_1 : $\langle$local-call#, integer, 25$\rangle$

doc_2 : $\ldots$

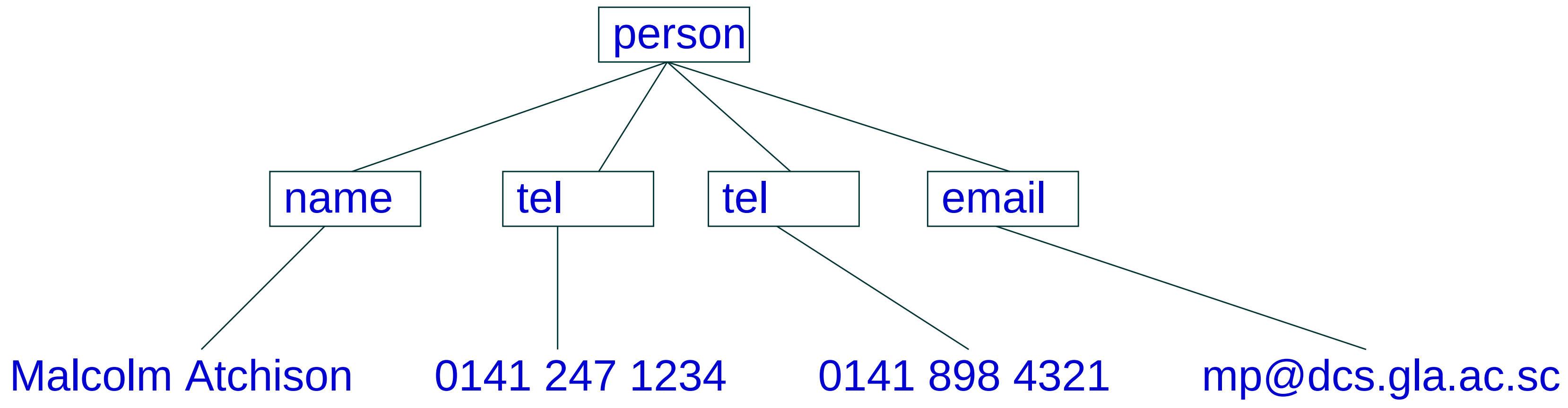
doc_3 : $\ldots$

$\ldots$

The general form is *oid* : $\langle$*label*, *type-indicator*, *value*$\rangle$. Note that records and sets are represented in the same way.

XML

```
⟨person⟩
      ⟨name⟩ Malcolm Atchison ⟨/name⟩
      ⟨tel⟩ 0141 247 1234 ⟨/tel⟩
      ⟨tel⟩ 0141 898 4321 ⟨/tel⟩
      ⟨email⟩ mp@dcs.gla.ac.sc ⟨/email⟩
⟨/person⟩
```



In XML the (horizontal) order of nodes is important.

Motivation – Browsing

To query a database one needs to understand the schema.

However schemas have opaque terminology and the user may want to start by querying the data with little or no knowledge of the schema.

- Where in the database is the string `"Casablanca"` to be found?
- Are there integers in the database greater than 2^{16}?
- What objects in the database have an attribute name that starts with `"act"`

While extensions to relational query languages have been proposed for such queries, there is no *generic* technique for interpreting them.

What is the model for semistructured data?

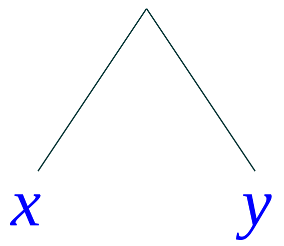
- A familiar representation for semistructured (unstructured) data?
- An attempt at a definition.
- Semistructured data as a labelled graph.
- A syntax for data.
- Examples.

Lisp – A language for unstructured data?

Lisp (basic Lisp) has one data structure that is used to represent a variety of data types.

Lisp has a syntax for building values, but has no separate syntax of types.

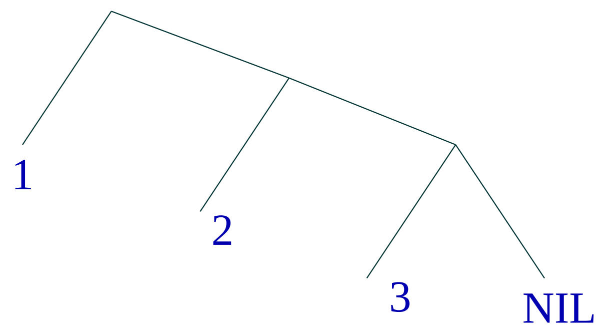
The basic constructor is CONS, which forms a tuple of its two arguments. The CONS of x and y is written (CONS x y) and can be depicted as a tree:



A variety of data structures, lists, trees, records, functions, may be represented using this constructor.

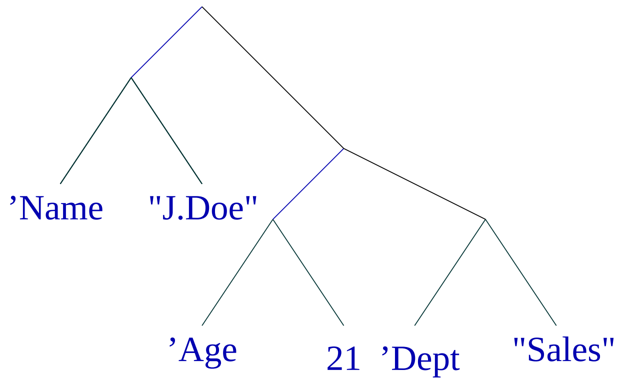
There are a number of extensions to Lisp (CLOS, LOOPS) and a “struct” definition in Common Lisp that add a syntax for types.

Representing Data in Lisp



A List

(CONS 1 (CONS 2 (CONS 3 (CONS NIL))))



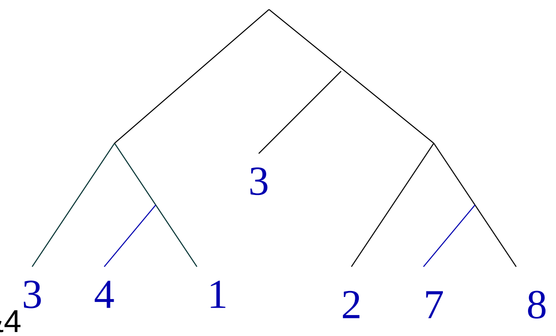
A Record

(CONS (CONS ’Name ”Joe”)

(CONS (CONS ’Age 21)

(CONS ’Dept ”Sales”)))

A Binary Tree (data at internal nodes)



(CONS (CONS 3 (CONS 4 1)

(CONS 3 (CONS 2 (CONS 7 9)))))

Describing Lisp Data

A Lisp value has a simple description. It is one of:

- a *number*, written 1,2,3 ...,
- a *string*, written ”cat”, ”dog”, ...,
- a *symbol*, written ’Name, ’Age, ...,
- NIL, or
- a pair of values, written (CONS x y)

This can be summarized in the type equation

$$\tau = \textit{number} \mid \textit{string} \mid \textit{symbol} \mid NIL \mid \tau \times \tau$$

A Definition of Semistructured Data?

As a partial definition, a semistructured data model is a syntax for data with *no separate syntax for types*.

That is, no schema language or data definition language.

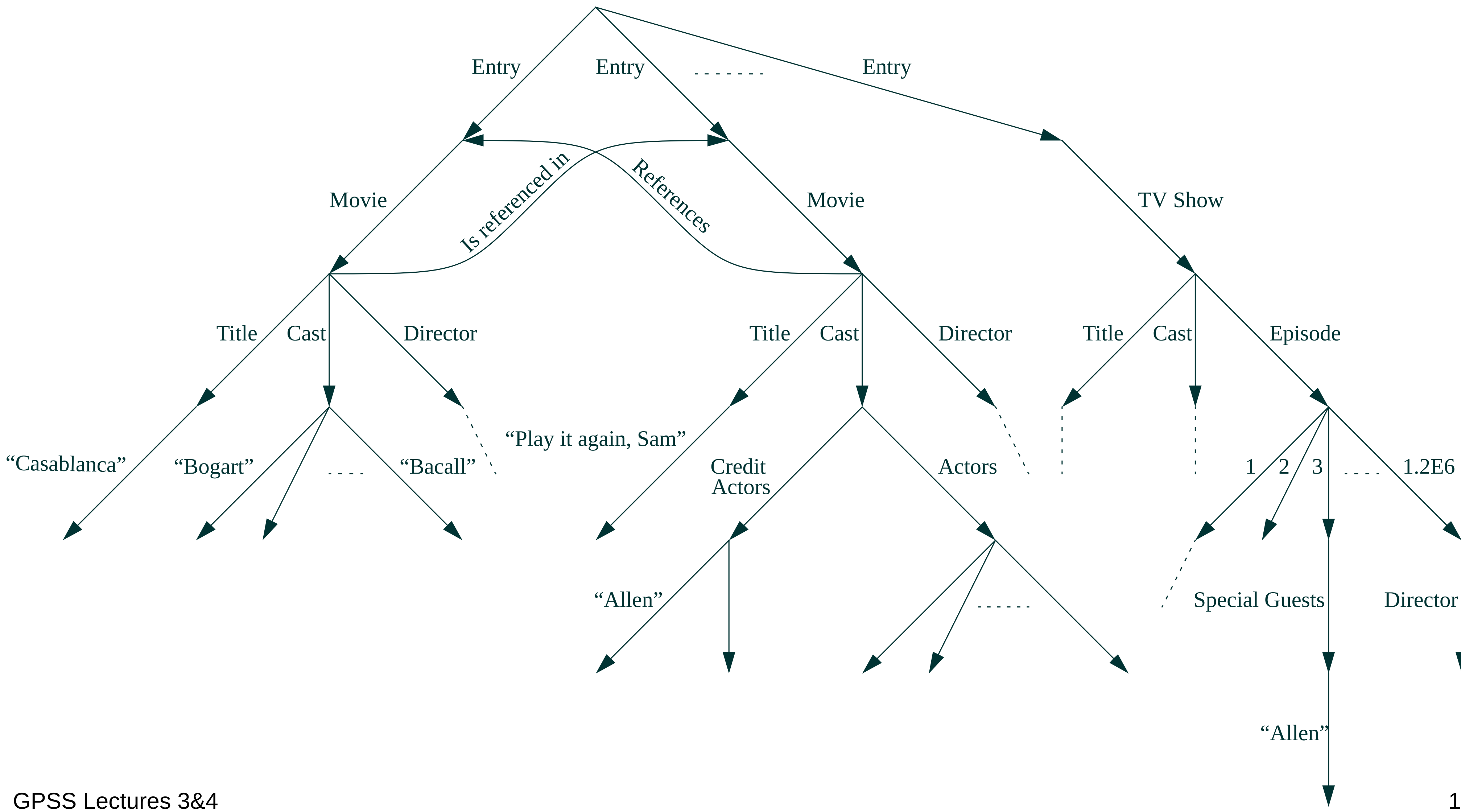
“Self describing” might be a better term, but this is used for data formats (e.g. ASN.1) that do have a syntax for types.

The Lisp data model is too “low-level” Coding a relational database as a Lisp value is possible (and often done) but the coding does not suggest any natural language for such values.

We would like a set type (or some collection type) to be explicit in our model.

Semistructured data is usually “mostly structured”. We are typically trying to capture data that has only minor deviations from relational / nested relational / object-oriented data. For example...

A Semistructured Movie Database



Semistructured Data as a Labeled Graph

We want to put data (base types) *int, string, video, audio* into our graph. We also want *symbols*. The names we use for attributes, relation names etc.

- Labels (symbols and data) on edges only (UnQL):

  $\textit{type label} = \textit{int} \mid \textit{string} \mid \ldots \mid \textit{symbol}$

  $\textit{type tree} = \textit{set}\left(\textit{label} \times \textit{tree}\right)$

- Symbols on edges, data at leaves (Lorel):

  $\textit{type base} = \textit{int} \mid \textit{string} \mid \ldots$

  $\textit{type tree} = \textit{base} \mid \textit{set}\left(\textit{symbol} \times \textit{tree}\right)$

- Symbols on edges, data on levaes nodes. (Simplified XML)

  $\textit{type base} = \textit{string}$

  $\textit{type tree} = \textit{label} \times \textit{list(tree)}$

- Object identities at nodes – to be discussed later.

What are the differences between these models?

1. Labels (symbols and data) on edges only.
2. Symbols on edges, data at leaves.
3. Data on edges and (all) nodes.

It is easy to define mappings between any two of these.

Having data on edges makes for nice representations of arrays (see ACeDB)

(3) has the mild disadvantage that taking a union of two graphs cannot be performed just by gluing together their roots.

Caution. There is a great distance between defining semistructured data as untyped or “schema-less” and adopting one of these models. There are all sorts of other models that may prove equally interesting.

I shall (not quite arbitrarily) adopt (1)

A Syntax for Data

The type definition almost determines a syntax for data. Here are some of the details.

- Usual syntax for numbers
- ”cat”, ”dog”, etc. for strings
- Unquoted strings Age, Name, etc. for symbols (drop the Lisp “quote”).
- $\{l_1 : t_1, \ldots, l_n : t_n\}$ – for a tree whose out-edges are $l_1, l_2, \ldots, l_n$ connected to trees $t_1, t_2, \ldots, t_n$.
- Shorthand l for $l : \{\}$ (terminal leaves).

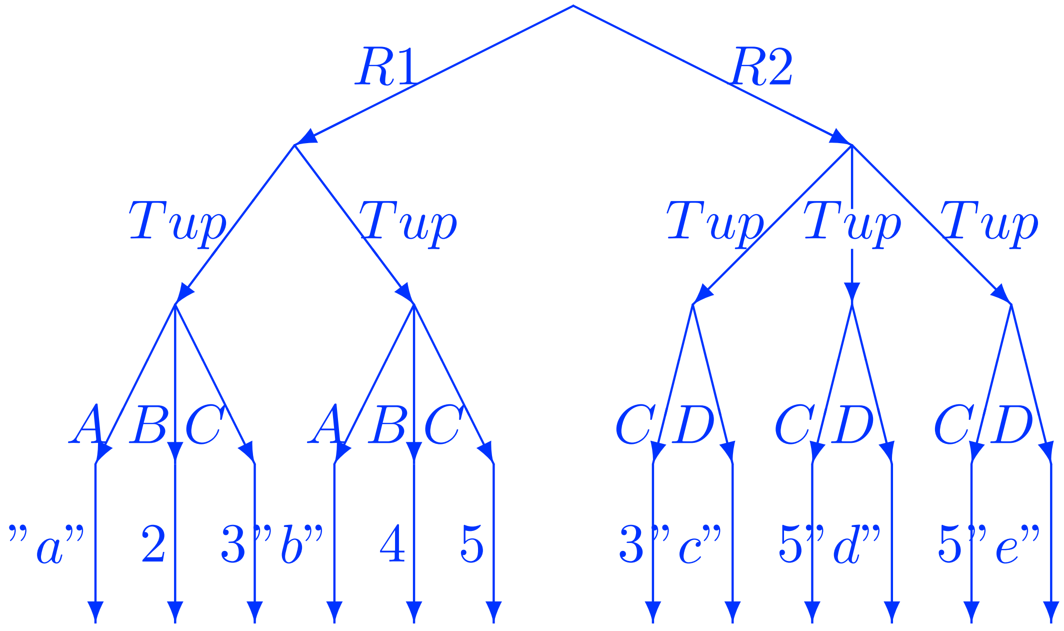
Example: Representing Relational Data

$R1$

A	B	C
$"a"$	2	3
$"b"$	4	5

$R2$

C	D
3	$"c"$
5	$"d"$
5	$"e"$



$$\{R1 : \{Tup : \{A : "a", B : 2, C : 3\}, Tup : \{A : "b", B : 4, C : 5\}\},$$
$$R2 : \{Tup : \{C : 3, D : "c"\}, Tup : \{C : 5, D : "d"\}, Tup : \{C : 5, D : "e"\}\}\}$$

Querying Semistructured Data

There are (at least) three approaches to this problem

- Add arbitrary features to SQL or to your favorite query language. This is the least likely to produce coherent results and may end up being the least useful.
- Find some principled approach to programs that are based on the *type* of the data.
- Represent the graph (or whatever the structure is) as appropriate predicates and use some variety of datalog on that structure.

The “Graph Datalog” approach

I shall not cover this approach in detail. Some remarks later

Please see references to WebSQL and WebLog.

The general approach is to represent a graph by two relations whose schemas are:

$\mathit{Node}(\mathit{oid}, \mathit{data})$	For nodes. *oid* is the node identifier *data* is the data at that node.
$\mathit{Edge}(\mathit{oid}, \mathit{label}, \mathit{oid})$	For edges. *label* carries edge information (may be the same as *da*

We can only expect a query to produce results on that part of the graph *reachable from the root*.

The “Extend SQL approach”

Having criticized this, it is the one I shall adopt (initially)!

In fact it is an attempt to extend the philosophy of OQL and comprehension syntax to these new structures.

It is the approach taken in the design of UnQL and also of Lorel.

In UnQL the syntax of the language is an *extension* of the syntax of the data.

Queries – in UnQL

$$\begin{array}{ll}\text{select} & t \\ \text{where} & R1 : \backslash t \leftarrow DB\end{array}$$

“Compute the union of all trees t such that DB contains an edge $R1 : t$ emanating from the root.”

There is only one such edge; this query returns the set of tuples in $R1$. The result is:

$$\begin{array}{ll}\{ & Tup : \{A : \text{"}a\text{"}, B : 2, C : 3\}, \\ & Tup : \{A : \text{"}b\text{"}, B : 4, C : 5\}\}\end{array}$$

- This is *not* SQL (No “from” clause).
- The form $(R1 : \backslash t) \leftarrow DB$ is a generator. Parentheses show grouping.
- $R1 : \backslash t$ is a *pattern*
- Introduction of a variable is explicit ($\backslash x$). There are other approaches.

A heterogeneous result

$$\text{select } t$$
$$\text{where } \backslash l : \backslash t \leftarrow DB$$

The result is the union of all tuples in both relations—a heterogeneous set that cannot be described by a single relation.

- The label variable $\backslash l$ is used to match any edge emanating from the root.
- In UnQL variables may be *label* variables or *tree* variables.

A join

$$\begin{array}{ll} \textsf{select} & \{Tup : \{A : x, D : z\}\} \\ \textsf{where} & R1 : Tup : \{A : \backslash x, C : \backslash y\} \leftarrow DB, \\ & R2 : Tup : \{C : y, D : \backslash z\} \leftarrow DB \end{array}$$

We join $R1$ and $R2$ on their common attribute C and then project onto A and D.

- $R1 : Tup : \{A : \backslash x, C : \backslash y\}$ is a tree pattern.
- Note that the variable y is bound in the pattern of one generator and then used as a constant in the pattern of the second.

A group-by

$$
\begin{aligned}
\text{select } \{x : (\quad & \text{select } y \\
& \text{where } R2 : Tup : \{C : x, D : \backslash y\} \leftarrow DB \\
&)\} \\
\text{where } R2 : Tup : C : \backslash x : \{\} \leftarrow DB &
\end{aligned}
$$

A group-by operation on $R2$ along the C column.

- $\backslash x : \{\}$ binds x to an edge label rather than a tree.
- In contrast, $\backslash y$ ranges over trees.
- The result is $\{3 : \{"c"\}, 5 : \{"d", "e"\}\}$.

At the movies – A

$$\begin{array}{ll} \textsf{select} & \{Tup : \{Title : x, Cast : y\}\} \\ \textsf{where} & Entry : \_ : \{Title : \backslash x, Cast : \backslash y\} \leftarrow DB \end{array}$$

The titles and casts of all movies.

- The “wildcard” symbol $\_$ matches any edge label.
- The result is a set of tuples of trees.

At the movies – B

$$\begin{array}{ll} \textsf{select} & \{Tup : \{Actor : x, Title : y\}\} \\ \textsf{where} & Entry : Movie : \{Title : \backslash y, Cast : \backslash z\} \leftarrow DB, \\ & \backslash x : \{\} \leftarrow z \textit{ union } (\textsf{select } u \textsf{ where } \_ : \backslash u \leftarrow z), \\ & isstring(x) \end{array}$$

A binary relation consisting of actress/actor and title tuples for movies.

- We assume that the names we want will be found immediately below the $Cast$ edge or one step further down.
- Note the use of a condition.

More on Types

Recall our recursive equation

$$\textit{type tree} = \textit{set}\big(\textit{label} \ \times \textit{tree}\big)$$

The type *set* is itself recursive, and can be constructed from

- The empty set $\{\}$
- The singleton set $\{l : t\}$
- The union of sets t_1 *union* t_2

This decomposition suggests certain natural forms of programming via *structural recursion*. The general form is

$$\begin{aligned} f(\{\}) &= e \\ f(\{l : t\}) &= s(l, t) \\ f(t_1 \ \textit{union} \ t_2) &= u(f(t_1), f(t_2)) \end{aligned}$$

where e, s, u are “simpler” functions.

However, a special case of this form gives us some interesting results:

$$\begin{aligned} f(\{\}) &= \{\} \\ f(\{l : t\}) &= s(l, t) \\ f(t_1 \textit{ union } t_2) &= f(t_1) \textit{ union } f(t_2) \end{aligned}$$

This restricted form of structural recursion is determined by the function s and defines a function $\textit{ext}(s)$ whose meaning is (informally)

$$\textit{ext}(s)\{l_1 : t_1, l_2 : t_2, \ldots l_n : t_n\} = s(l_1, t_1) \textit{ union } s(l_2, t_2) \textit{ union } \ldots \textit{ union } s(l_n, t_n)$$

I.e., apply s to each member of the tree (taken as a set) and union together the results:

$$\begin{aligned} f(\{\}) &= \{\} \\ f(\{l : t\}) &= \text{if } l = \text{R1 then } t \text{ else } \{\} \\ f(t_1 \textit{ union } t_2) &= f(t_1) \textit{ union } f(t_2) \end{aligned}$$

This is our first query that selects a relation from the database.

Some Basic Results

We can build a language *EXT* in which the *only* “computation” on sets is given by *ext*. The other things we need are:

- For sets: empty set, $\{\}$, singleton, $\{l : t\}$, and union, $(t_1 \textit{ union } t_2)$
- Decomposition of $l : t$ (pattern matching).
- A conditional expression $\mathsf{if} \ldots \mathsf{then} \ldots \mathsf{else} \ldots$
- Equality on labels, an emptiness test, predicates on labels e.g., $\textit{isstring}(l)$.

EXT has some important properties:

- The select . . . where . . . language, as informally described to this point, can be implemented with *EXT*.
- On the "natural" encoding of relations as trees, (nested) relational queries can be implemented in *EXT*.
- Queries in *EXT* that take (nested) relations as inputs and produce (nested) relations as output can be implemented in (nested) relational algebra.
- I.e. *EXT* is a natural extension of (nested) relational algebra.

“Deep” structural recursion

We could try to generalize the recursive function that defined *EXT* to

- a definition of the form

$$\begin{aligned} f(\{\}) &= \{\} \\ f(\{l : t\}) &= s(l, f(t)) \\ f(t_1 \textit{ union } t_2) &= f(t_1) \textit{ union } f(t_2) \end{aligned}$$

- or possibly

$$\begin{aligned} f(\{\}) &= \{\} \\ f(\{l : t\}) &= s(l, t, f(t)) \\ f(t_1 \textit{ union } t_2) &= f(t_1) \textit{ union } f(t_2) \end{aligned}$$

  In which the function f is called on subtrees.

Consider special cases of this:

$$\begin{aligned}
\textit{strings}(\{\}) &= \{\} \\
\textit{strings}(\{l : t\}) &= (\text{if } \textit{isstring}(l) \text{ then } \{l\} \text{ else } \{\})\ \textit{union strings}(l) \\
\textit{strings}(t_1 \textit{ union } t_2) &= \textit{strings}(t_1)\ \textit{union strings}(t_2)
\end{aligned}$$

$$\begin{aligned}
\textit{paths}(\{\}) &= \{\} \\
\textit{paths}(\{l : t\}) &= \{l\}\ \textit{union} \text{ select } \{l : t\} \text{ where } \backslash t \leftarrow \textit{paths}(t) \\
\textit{paths}(t_1 \textit{ union } t_2) &= \textit{paths}(t_1)\ \textit{union paths}(t_2)
\end{aligned}$$

On *trees* they are both well defined when considered as equations or as programs.

On *cyclic structures* the first has a well-defined solution, but as a program it would recurse indefinitely.

On *cyclic structures* the second does not have a finite solution as data. What kind of restriction do we need to avoid this, and how do we implement the well-defined cases?

Going Deep

Let’s try to resolve the issue again by “adding features”!!!

$$\begin{array}{l}\textsf{select } \{l\} \\ \textsf{where } \_* : \backslash l : \_ \leftarrow DB, isstring(l)\end{array}$$

Find all the strings in the database

- The $\_*$ is a “repeated wildcard” that matches any path.

The use of a leading $\_*$ is so common that we shall use a special abbreviation $p \twoheadleftarrow t$ for $\_* : p \leftarrow t$. So:

$$\begin{array}{l}\textsf{select } \{l\} \\ \textsf{where } \backslash l : \_ \twoheadleftarrow DB, isstring(l)\end{array}$$

Doubly deep

$$\begin{array}{ll} \textsf{select} & \{Movie : x\} \\ \textsf{where} & Movie : \backslash x \twoheadleftarrow DB, \\ & \text{''}Bogart\text{''} : \_ \twoheadleftarrow x, \\ & \text{''}Bacall\text{''} : \_ \twoheadleftarrow x \end{array}$$

We use consecutive “deep” generators to find all the movies involving “Bogart” and “Bacall”:

The error corrected

$$\begin{array}{ll}\textsf{select} & \{Movie : x\} \\ \textsf{where} & Movie : \backslash x \twoheadleftarrow DB, \\ & [\hat{}Movie]* : "Bogart" : \_ \leftarrow x, \\ & [\hat{}Movie]* : "Bacall" : \_ \leftarrow x\end{array}$$

Following `grep`, the pattern $[\hat{}\textit{Movie}]*$ matches any path that does not contain the label *Movie*.

Arbitrary regular expressions may be used on labels.

A “deep” version of *EXT*

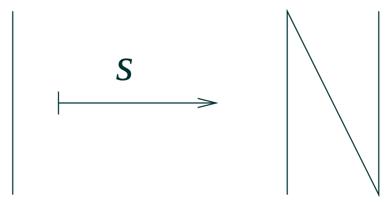
Recall the definition of *ext*:

$$\mathit{ext}(s)\{l_1 : t_1, l_2 : t_2, \ldots l_n : t_n\} = s(l_1, t_1) \text{ union } s(l_2, t_2) \text{ union } \ldots \text{ union } s(l_n, t_n)$$

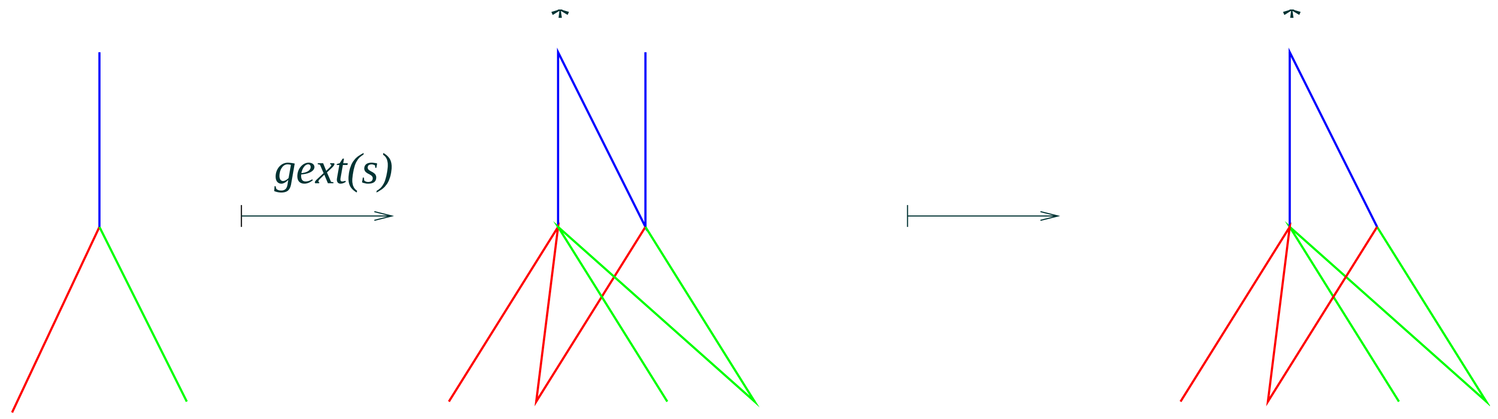
Read as “replace each element x in a set by $s(x)$ and ‘glue together’ the results”

We are going to generalize this operation to graphs, but it is easier to descibe the syntax with pictures:

Suppose our function s acts on individual edges to produce a graph with n inputs and n outputs



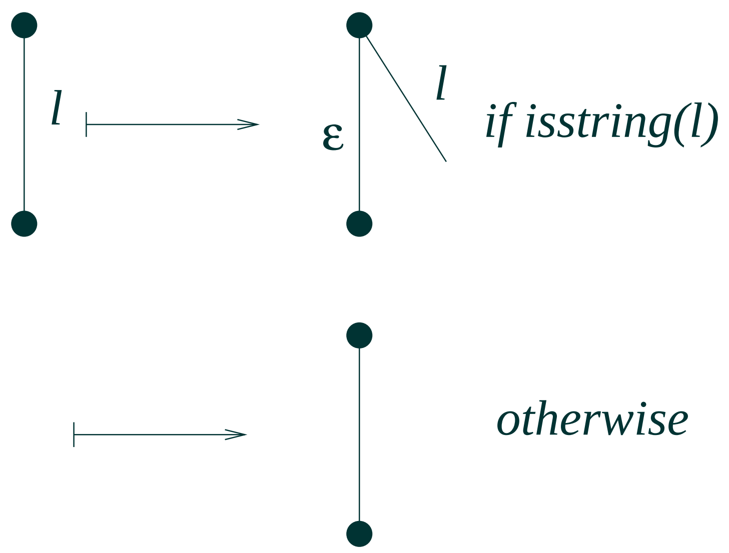
Apply this funtion *in parallel* to each edge of the input tree and glue together corresponding inputs and outputs.

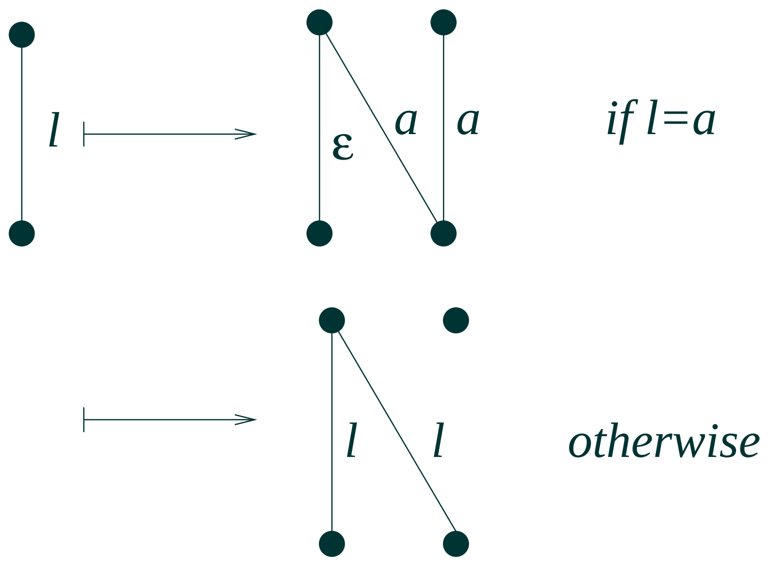


By default the top left vertex of the new graph is chosen as the new root.

(The function does not have to preserve the shape of the graph, but the number of inputs and outputs must be the same in all cases.)

Some Examples of gext

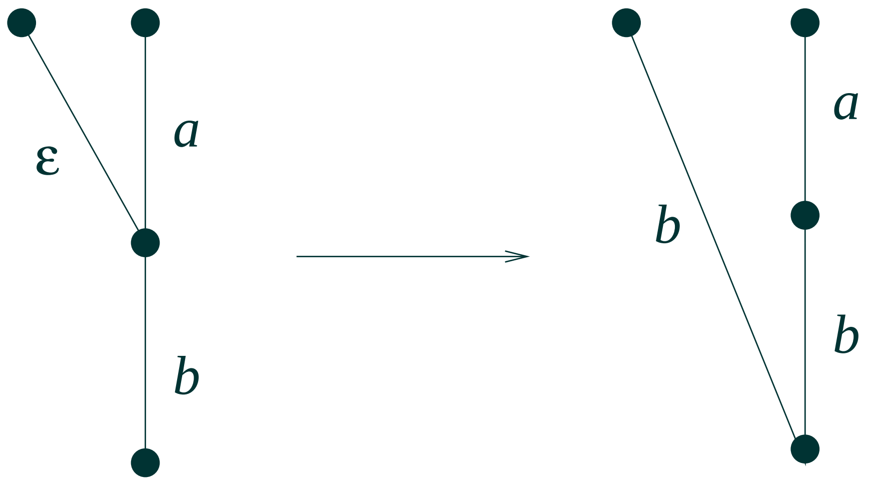




All the strings in a tree

The union of all the trees at the ends of $a*$ paths

ε-edges represent unions. The operation on graphs is to eliminate them by rewriting:



Elimination of ε-edges is similar to transitive closure.

Results concerning *GEXT*

GEXT is, by analogy with *EXT*, the language obtained by using *gext* to compute with graphs.

GEXT is (fairly obviously) well defined for cyclic structures.

GEXT can also be used to implement “deep” select . . . where . . . fragment of UnQL with arbitrary regular expressions on paths.

GEXT Can also be use to *transform* a graph. E.g. to correct the egregious mistake in the cast of “Casablanca”.

*H*owever the extent to which *GEXT* can modify a graph is limited. It cannot, for example, add the reverse of every edge to a graph.

GEXT allows similar optimizations in the “vertical” dimension to the “horizontal” optimizations of *EXT* – many of the relational algebra optimizations.

Conclusions and Prospects

The select . . . where . . . fragment of UnQL and Lorel have very similar syntax.

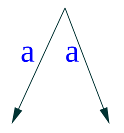
Lorel has some additional constructs for dealing with object identity.

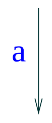
This raises an interesting question of what various languages can “observe” about a graph.

UnQL observes graphs up to bisimulation. If two graphs are bisimilar, UnQL queries will produce the same ouptut. If they are not bisimilar, there is an UnQL query that distinguishes them.

Separating Pairs

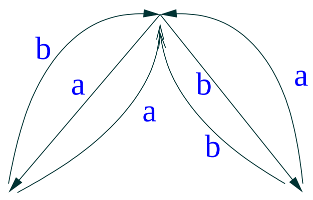
Graph isomorphism

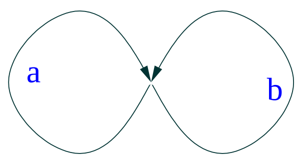




Distingushed by graph datalog with node equality.

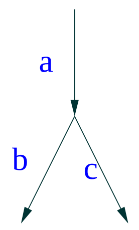
First-order Equivalence

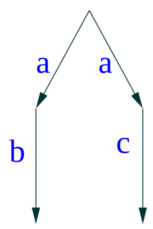




Distinguished by graph datalog.

Bisimilarity





Distinguished by UnQL

Lots more to do ...

Is the model right? What about lists rather than sets for building trees? Not so easy to write "nicely behaved" programs on cyclic data.

Is semistructured data a good idea? Why not get the structure right in the first place? (But existing data models do not accommodate structures like ACeDB.)

Schemas. See Suciu and Goldman for ideas on how schemas can be used for optimization. These (respectively) use similarity and NDFSA equivalence to define schemas.

Browsing There ought to be some principles here. Semistructured data is a good model for browsing, but we need to convey the structure to the user at the same time.

Finding structure How do we extract/infer structure from semistructured data?

Conversion standards? There is more than one way of representing even a relational database as semistructured data. Which is “right”?

Creating semi-structured data How do we rapidly parse/extract semistructured data from text formats?

Co-existence of structured and semistructured data

Our languages ought to allow us to handle both types (structured and semistructured) of data in the same framework.

Our implementations ought to make efficient use of structure when it exists. They should allow both forms to coexist.

We should not have to use semistructured data just because our languages or implementations are weak in representing structure.

XML – the reality

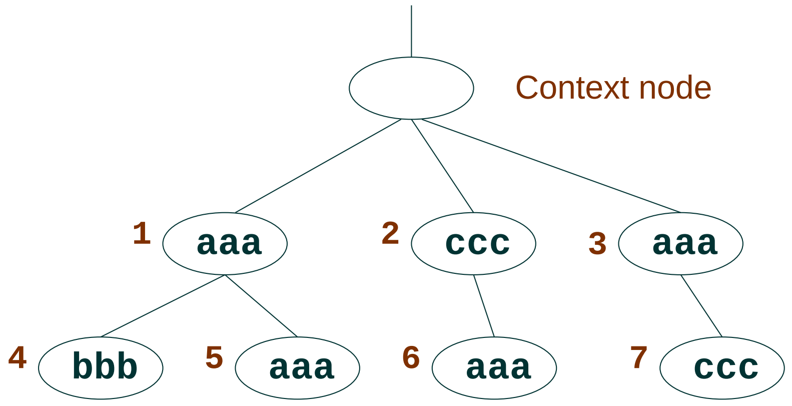
A series of prototype query languages, UnQL, Lorel, XML-QL, . . . led to the present state of affairs, XQuery. This consists of two parts.

- XPath – a language for identifying sets of nodes in an XML tree.
- XQuery – Comprehension syntax surrounding XPath

The problem is that XPath has a life of its own, and does not have any primcipled basis in, e.g., some algebra.

XPath

Navigation is remarkably like navigating a unix-style directory.



All paths start from some context node.

`aaa`	all the child nodes of the context node labeled `aaa` {1,3}
`aaa/bbb`	all the `bbb` children of `aaa` children of the context node {4}
`*/aaa`	all the `aaa` children of *any* child of the context node {5,6}.
`.`	the context node
`/`	the root node

XPath- child axis navigation (cont)

`/doc`	all the `doc` children of the root
`./aaa`	all the `aaa` children of the context node (equivalent to `aaa`)
`text()`	all the text children of the context node
`node()`	all the children of the context node (includes text and attribute nodes)
`..`	parent of the context node
`.//`	the context node and all its descendants
`//`	the root node and all its descendants
`//para`	all the para nodes in the document
`//text()`	all the text nodes in the document
`@font`	the font attribute node of the context node

Predicates

`[2]`	the second child node of the context node
`chapter[5]`	the fifth chapter child of the context node
`[last()]`	the last child node of the context node
`person[tel="12345"]`	the `person` children of the context node that have or more `tel` children whose string-value is `"1234"` string-value is the concatenation of all the text on des dant text nodes)
`person[.//name = "Joe"]`	the person children of the context node that have in descendants a firstname element with string-value `"Jo`

From the XPath specification (`$x` is a variable – see later):

> NOTE: If `$x` is bound to a node set then `$x = "foo"` does not mean the same as `not ($x != "foo")`.

Unions of Path Expressions

- `employee | consultant` – the union of the `employee` and `consultant` nodes that are children of the context node
- For some reason `person/(employee|consultant)` – as in general regular expressions – is not allowed
- However `person/node()[boolean(employee|consultant)]` is allowed!!

From the XPath specification:

The boolean function converts its argument to a boolean as follows:

- a number is true if and only if it is neither positive or negative zero nor NaN
- a node-set is true if and only if it is non-empty
- a string is true if and only if its length is non-zero
- an object of a type other than the four basic types is converted to a boolean in a way that is dependent on that type.

A Query in XPath

```
SELECT age FROM employee WHERE name = "Joe"
```

We can write an XPath expression:

```
//employee[name="Joe"]/age
```

Find all the `employee` nodes under the root. If there is at least one `name` child node whose string-value is `"Joe"`, return the set of all `age` children of the `employee` node.

Or maybe

```
//employee[//name="Joe"]/age
```

Find all the `employee` nodes under the root. If there is at least one `name` *descendant* node whose string-value is `"Joe"`, return the set of all `age` *descendant* nodes of the `employee` node.

N.B. This returns a set of nodes, not XML

Why isn’t XPath a query language?

It doesn’t return XML – just a set of nodes.

It cant do complex queries invoking joins.

We’ll turn to XQery shortly, but there’s a bit more on XPath.

XPath – navigation axes

In Xpath there are several navigation axes. The *full* syntax of XPath specifies an axis after the `/`. E.g.,

`ancestor::employee`: all the `employee` nodes *directly above* the context node

`following-sibling::age`: all the `age` nodes that are *siblings* of the context node and to the *right* of it.

`following-sibling::employee/descendant::age`: all the `age` nodes *somewhere below* any `employee` node that is a *sibling* of the context node and to the *right* of it.

`/descendant::name/ancestor::employee`: Same as `//name/ancestor::employee` or `//employee[boolean(.//name)]`

So XPath consists of a series of navigation steps. Each step is of the form: *axis*::*node test*[*predicate list*]

Navigation steps can be concatenated with a `/`

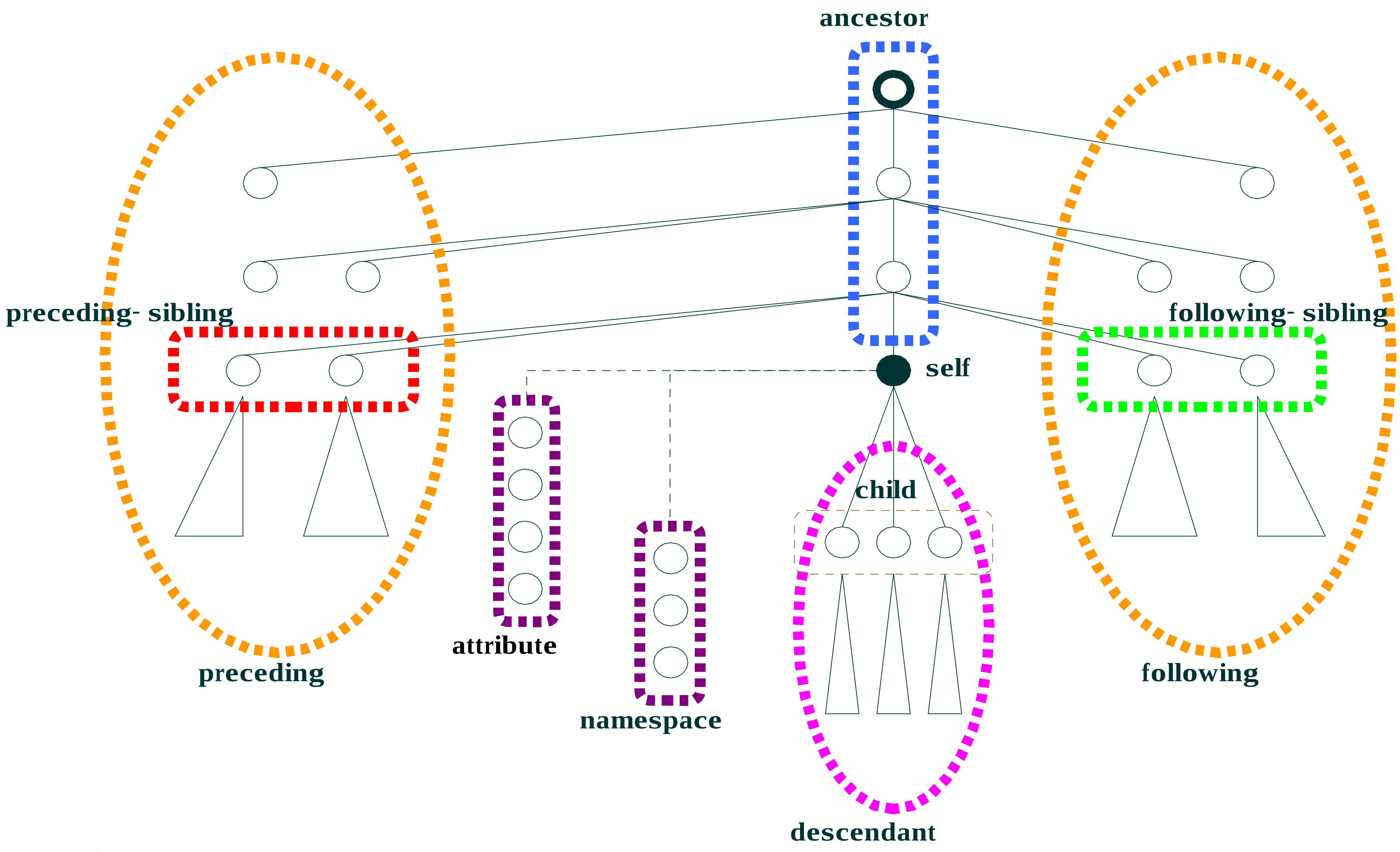
If the path starts with `/` or `//`, start at root. Otherwise start at context node.

The following are abbreviations/shortcuts.

- no axis means child
- `//` means `/descendant-or-self::`

The full list of axes is: `ancestor`, `ancestor-or-self`, `attribute`, `child`, `descendant`, `descendant-or-self`, `following`, `following-sibling`, `namespace`, `parent`, `preceding`, `preceding-sibling`, `self`.

The XPath axes



XQuery

XPath is central to XQuery. In addition to XPath, XQuery provides:

- XML “glue” that turns XPath node sets back into XML.
- Variables that communicate between XPath and XQuery.
- It is “reverse” comprehension syntax, so that you can do things like joins, aggregates and more sophisticated conditions than those in XPath.

A simple query. The {...} embeds XPath expressions in XML. (XPath in orange):

```
⟨answer⟩{document("bib.xml")//title}⟨/answer⟩
```

produces:

```
⟨answer⟩
   ⟨title⟩...⟨/title⟩
   ⟨title⟩...⟨/title⟩
   ...
⟨/answer⟩
```

“Select-Project” in XQuery

```
for $x in document("payroll.xml")//employee
where $x/age = "25"
return $x/name
```

- `$x` gets bound to each node in the set of nodes produced by the XPath expression `document("payroll.xml")//employee`.
- `$x/age` produces a *set* of nodes. As in XPath, `$x/age` = `"25"` is true if at least one element in `$x/age` has string value `"25"`.

Join in XQuery

```
⟨results⟩
    for $x in document("payroll.xml")//employee
        $d in document("organization.xml")//department
   where value-equals($x/DeptId, $d/DeptId)
   return ⟨result⟩{$x/name}{$x/name}⟨/result⟩
⟨/results⟩
```

What happens if a department has two names, or an employee has two names, or both?

Group by

```
⟨answer⟩
   for $a in distinct-values(document("payroll.xml")//employee/age)
   return
      ⟨age-group⟩
         { $a }
         {
            for $e in document("payroll.xml")//employee
            where value-equals($a, $e/age)
            return $a/name
         }
      ⟨/age-group⟩
⟨/answer⟩
```

Examples from XQuery

Use of aggregate functions

List each publisher and the average price of their books.

```
for $p in distinct(document("bib.xml")//publisher)
let $a := avg(document("bib.xml")//book[publisher = $p]/pric
return
   ⟨publisher⟩
      ⟨name⟩{$p/text()}⟨/name⟩
      ⟨avgprice⟩{$a}⟨/avgprice⟩
   ⟨/publisher⟩
```

`let` binds a new variable.

Examples from XQuery (cont)

List the publishers who have published more than 100 books.

```
⟨big-publishers⟩
   {
      for $p in distinct(document("bib.xml")//publisher)
      let $b := document("bib.xml")//book[publisher = $p]
      where count($b) > 100
      return $p
   }
⟨/big-publishers⟩
```

Note that `let` binds to a set – it does not cause another iteration.

Document Type Descriptors

XML has gained acceptance as a standard for data interchange. There are now hundreds of published DTDs. DTDs are described in the XML standard and in most XML tutorials.

- A Document Type Descriptor (DTD) constrains the structure of an XML document.
- There is some relationship between a DTD and a schema, but it is not close – hence the need for additional “typing” systems, such as XML-Schema.
- The unlike an E-R diagram, a DTD is a syntactic specification. Its connection with any conceptual model may be quite remote.

Example: The Address Book

`⟨person⟩`	
`    ⟨name⟩ MacNiel, John ⟨/name⟩`	must exist
`    ⟨greet⟩ Dr.  John MacNiel ⟨/greet⟩`	optional
`    ⟨addr⟩ 1234 Huron Street ⟨/addr⟩`	as many address lines as needed
`    ⟨addr⟩ Rome, OH 98765 ⟨/addr⟩`	
`    ⟨tel⟩ (321) 786 2543 ⟨/tel⟩`	0 or more tel and faxes in any order
`    ⟨fax⟩ (123) 456 7890 ⟨/fax⟩`	
`    ⟨tel⟩ (321) 198 7654 ⟨/tel⟩`	
`    ⟨email⟩ jm@abc.com ⟨/email⟩`	0 or more email addresses
`⟨/person⟩`	

Specifying the Structure

`name`	to specify a `name` element
`greet?`	to specify an optional (0 or 1) `greet` elements
`name,greet?`	to specify a name followed by an optional `greet`
`addr*`	to specify 0 or more `address` lines
`tel | fax`	a `tel` or a `fax` element
`(tel | fax)*`	0 or more repeats of `tel` or `fax`
`email*`	0 or more email elements

Specifying the structure (cont)

So the whole structure of a person entry is specified by

```
name, greet?, addr*, (tel | fax)*, email*
```

This is a regular expression in slightly unusual syntax. Why is it important?

A DTD for the address book

```
⟨!DOCTYPE addrbooktype [
   ⟨!ELEMENT addressbook (person*)⟩
   ⟨!ELEMENT person (name, greet?, addr*, (fax|tel)*, email*
   ⟨!ELEMENT name (#PCDATA)⟩
   ⟨!ELEMENT greet (#PCDATA)⟩
   ⟨!ELEMENT addr (#PCDATA)⟩
   ⟨!ELEMENT tel (#PCDATA)⟩
   ⟨!ELEMENT fax (#PCDATA)⟩
   ⟨!ELEMENT email (#PCDATA)⟩
]⟩
```

XDuce - a Typed XML programming Language

- DTDs (and XML-Schema) constrain the tags and order of subelements.
- For most query languages DTDs and XML-Schema do not act as stype type systems.
- Validation $\neq$ typechecking. Incorrect queries yield empty answers.
- An exception is XDuce ...

Yet another syntax..

```
⟨addrbook⟩
   ⟨name⟩Jane Dee⟨/name⟩
   ⟨addr⟩NYC⟨/addr⟩
   ⟨tel⟩213 1234⟨/tel⟩
   ⟨tel⟩213 7654⟨/tel⟩
   ⟨name⟩John Doe⟨/name⟩
   ⟨addr⟩Neasden⟨/addr⟩
   ⟨tel⟩745 0011⟨/tel⟩
⟨/addrbook⟩
```

$\longrightarrow$

```
addrbook[
   name["Jane Dee"],
   addr["NYC"],
   tel["213 1234"],
   tel["213 7654"],
   name["John Doe"],
   addr["Neasden"],
   tel["745 0011"] ]
```

Also for the types...

```
⟨!ELEMENT addrbook (name, addr, tel*)*⟩
⟨!ELEMENT name (#PCDATA)⟩
⟨!ELEMENT addr (#PCDATA)⟩
⟨!ELEMENT tel (#PCDATA)⟩
```

$\longrightarrow$

```
type Addrbook = addrbook[(Name,Addr,Tel*)*]
type Name =     name[Str]
type Addr =     addr[Str]
type Tel =      tel[Str]
```

Subtyping

Types denote sequences of values, e.g.

```
tel["1234"],tel["2345"] :   Tel*
```

Subtyping is derived from containment of regular expressions and denotes “sub-forests”, e.g.

```
Tel <: Tel*
Name, Addr <:  Name, Addr, Tel*
addrbook[Name,Addr,Name,Addr,Tel],addrbook[(Name,Addr)*]
            <:  Addrbook
```

XDuce types are more general than DTDs. Example: `a[b[c[Str]],b[d[Str]]]`

Pattern Matching and Functions

```
fun mkAddrList: (Name,Addr,Tel*)* -> (Name, Addr)* =
  name[n:Str],addr[a:Str],tels:Tel*,rest:(Name,Addr,Tel*)*
    ->name[n],addr[a],mkAddrList(rest)
| () -> ()


fun mkTelList (Name,Addr,Tel*)* -> (Name, Tel)* =
  name[n:Str],addr[a:Str],tels:[t:Tel,restT:Tel*],
    rest:(Name,Addr,Tel*)
    -> name[n], tel[t], mkTelList(name[n],addr:[a],tels[rest
| name[n:Str], addr[a:Str], rest:(Name,Addr,Tel*)*
    -> mkTelList(rest)
| () -> ()
```

About XDuce

- It is a full programming language. Substantial applications (e.g. an XML Schema validator) have been written in it.
- Subtyping and type equivalence are non-trivial.
- “Width” (record) subtyping has also been added. But this may need further work.
- The expected “type-safety” theorems hold.

The future

- The big question is whether we can store large quantities of (typed?) XML and query them efficiently – as we can for relational databases.
- To what extent can we type-check X-Query?
- What is the “right” way of combining regular expression types with familiar (record, variant,...) data types?
- Can we make XDuce a higher-order language? Can we add parametric polymorphism?
- Can we find an optimisable “algebra”?
- DTDs and (worse) XML-Schema are complicated, and there are no clean underlying principles. Can we find something that is close and clean?
- Similarly for XPath.

The list is endless ...

Bibliography

Serge Abiteboul, Peter Buneman and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

Peter Buneman, Mary Fernandez and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal 9(1), 75-110, 2000.

Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom and Janet L. Weiner. The Lorel Query Language for Semistructured Data. Journal of Digital Libraries, volume 1:1, 1997.

Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu. XML-QL: A Query Language for XML. `http://www.w3.org/TR/NOTE-xml-ql`

Mary Fernandez, Jerome Simeon, Philip Wadler. An Algebra for XML Query. FST TCS, Delhi, December 2000.

Haruo Hosoya, Benjamin C. Pierce. XDuce: A Typed XML Processing Language. Int'l Workshop on the Web and Databases (WebDB) 2000

And, of course, `http://www.w3.org/XML/`

$\langle$/lecture$\rangle$