Nested Relational Algebra and Collection Programming

Reminder: The Relational Model

- Relations are sets (multisets) of tuples.

R	A	B
	1	2
	3	4
	3	4

- Schema $R(A, B)$; $sch(R) = \{A, B\}$. Column names (and their data types).

Reminder: Relational Algebra #1

- **Selection $\sigma_\phi(R)$**

  Selection condition ϕ: Boolean combination of expressions $t\theta t'$ where
  - t, t': either a column name from $sch(R)$ or a constant value and
  - θ is one of $=, \neq, <$, and $\leq$.

  Result: Those tuples of R for which ϕ is true.

- **Projection $\pi_{\vec{A}}(R)$**

  Assumption: $\vec{A} \subseteq sch(R)$.
  Result: For each tuple $\vec{a}\vec{b}$ of R in which $\vec{a}$ are the values in the columns $\vec{A}$ and $\vec{b}$ are the values in the remaining columns of R, return $\vec{a}$. The result consists of distinct tuples, i.e., duplicate tuples are removed from the result.

Reminder: Relational Algebra #2

- Relational ("Cartesian", thus C) product $R \times S$
  Assumption: $sch(R) \cap sch(S) = \emptyset$.
  Result: The set of distinct tuples $\vec{r}\vec{s}$ obtained by concatenating tuples $\vec{r}$ from R with tuples $\vec{s}$ from S.
- Column renaming $\rho_{A_1 \ldots A_{ar(R)}}(R)$
  Assumption: $ar(R) = k$.
  Result: If the schema of R is $R(B_1, \ldots, B_{ar(R)})$, rename column B_i to A_i, for each $1 \leq i \leq ar(R)$.

Select-Project-Cartesian product (SPC) queries: relational algebra queries built using the operations σ, π, and $\times$ (and ρ).

- Theta-join: $R \bowtie_\theta S := \sigma_\theta(R \times S)$.

Refresh: Relational Algebra #3

- Union $R \cup S$: union of the tuples in R and S
- Relational difference $R \backslash S$: The tuples of R that are not also in S. (In the case of bag semantics, for each distinct tuple t of R, the multiplicity of t in the result is the multiplicity in R minus the multiplicity in S)

Nested Relations

- First normal form (1NF):
  - Fields are atomic from the point of view of the query language and have no further structure.
  - Classical relational databases consist of 1NF relations.
- Nested ("non-first normal form" (NFNF)) relations: fields do not have to be atomic but may be relations.

Motivation: Values beyond Relations

- Impedance Mismatch – relational DBMS store relations, modern software works with objects (persistency needed!)
- OO data model:
  - Complex values
  - Object identity
  - Behavior: class methods
- Status
  - Very fashionable in the 80s and early 90s.
  - Now most OODBMS companies defunct.
  - But: Many RDBMS are now object-relational DBMS.
  - Difference: Relational at the core, OO functionality through an interface layer.
- Collection Programming

A Query Language for Nested Relations

- Nested relational algebra = relational algebra + nest + unnest
- On (NFNF) relation $R(\vec{A}\vec{B})$, $\text{nest}_{C=(\vec{B})}(R)$ is defined as
  $\{\langle \vec{A}: \vec{x}, C: \{\langle \vec{B}: \vec{y}\rangle \mid \langle \vec{A}: \vec{x}, \vec{B}: \vec{y}\rangle \in R\}\rangle \mid (\exists \vec{y})\langle \vec{A}: \vec{x}, \vec{B}: \vec{y}\rangle \in R\}$.
  Analogous to SQL group-by $\vec{A}$, but no aggregation required.
- Unnest flattens one level of nesting (undoes a nest operation). On NFNF relation $R(\vec{A}C)$ where C is an NFNF column with schema $\vec{B}$, unnest_C is the inverse of $\text{nest}_{C=(\vec{B})}(R)$.

R	A	B
	1	2
	1	3
	1	4
	5	6
	5	7
	8	9

$\xRightarrow{\text{nest}_{C=(B)}}$

$\xLeftarrow{\text{unnest}_C}$

$\text{nest}_{C=(B)}(R)$	A	C
	1	B: 2, 3, 4
	5	B: 6, 7
	8	B: 9

Nested Relational Algebra (NRA) Discussion

- No query examples given here. Why: NRA adds no querying power to relational algebra. Boring.

Theorem (Conservativity w.r.t. relational algebra)

For every NRA query that maps 1NF relations to 1NF relations, there is an equivalent relational algebra query.

- But: power to create and work with nested values: object bases, XML, JSON, etc.
- NRA is a very robust language in a fundamental way – it is the "relational algebra of nested relations".
  - See the conservativity theorem!
  - One can come up with this or an equivalent language in many different ways, analogously to the equivalence class of languages that relational algebra and relational calculus belong to.

Collection Programming

Collection-at-a-time Processing

- Program analysis is very hard to do automatically!
- One can aid the program analyzer by avoiding state (purely functional vs. imperative programming) and using very regular code (foreach rather than loops with counters or iterators).
- Collection-at-a-time processing as a PL abstraction (map , foreach): do work in bulk on many elements of a collection; treat them all in the same way.
  Benefit: Allow a compiler or query optimizer to automatically
  - analyze the code for optimization potential beyond what is feasible for classical code that processes data in a more ad-hoc way.
  - use hardware support for bulk processing (within a core) such as long instruction words and vectorization.
  - parallelize beyond a core (embarassing parallelism as in MapReduce).

Collection-at-a-time Processing

- Collection-at-a-time processing is the key to the success of SQL and DBMS!

  select ... from ... where ...

  Conceptually, everything is done with every tuple of the database (including filtering), and all tuples are treated equal!
- MapReduce
  - "Embarassing parallelism" through map !
  - Collection.map(f): apply f to every element of the collection, and return the result.
  - This abstraction comes from (higher-order) functional programming. Not at all a new idea!

Higher-order Languages

- We will study higher-order typed functional languages.
- Databases are nested "complex values" built from base types, tuples and collections (e.g., sets). Relations are a special case.
- Queries are functions mapping between complex values.
- Higher-order functions: If f is a query, then map(f) – apply f to each member of a collection – is a (higher-order) query.

$$\text{map}(f)(\{a_1, \ldots, a_n\}) = \{f(a_1), \ldots, f(a_n)\}$$

- This is called structural recursion on such collections.
- We will not use recursion beyond this; our language fragment is not Turing-complete.

Motivation: Modern Collection Programming

- Collections in functional languages: OcaML, Haskell, Scala, ...
- Scala trait Traversable.
  - Need to implement

    $$def\ foreach[U](f:\ Elem => U)$$

  - Implements map, flatten, ...; folds, reduce; aggregations.
  - Ancestor of rich collections library: Lists, vectors, associative arrays, ...
  - Spark: implements just yet another Scala collection class (which automatically distributes computation).
- Microsoft LINQ: C#, F#, Visual Basic
- Java for-comprehensions on collections. Java 8 is functional.
- A foundation for map-reduce programming!

Writing a query with Scala collections

Self-join on binary relation R(A,B) $= \{(1,2),(2,3),(3,3)\}$ in SQL:

```
select r1.A, r2.B from R r1, R r2 where r1.B = r2.A
```

The same in Scala (using lists to represent relations):

```
scala> val R = List((1,2),(2,3),(3,3))
R: List[(Int, Int)] = List((1,2), (2,3), (3,3))

scala> val RxR = R.map(x => R.map(y =>
           (x._1, x._2, y._1, y._2))).flatten
RxR: List[(Int, Int, Int, Int)] = List((1,2,1,2),
   (1,2,2,3), (1,2,3,3), (2,3,1,2), (2,3,2,3),
   (2,3,3,3), (3,3,1,2), (3,3,2,3), (3,3,3,3))

scala> val q = RxR.filter(x => x._2==x._3)
                  .map(x => (x._1, x._4))
q: List[(Int, Int)] = List((1,3), (2,3), (3,3))
```

RxR is the relational self-product $R \times R$, filter performs selection, and the final map performs projection.

The same in OCAML

```
# let r = [(1,2); (2,3); (3,3)];;
val r : (int * int) list = [(1, 2); (2, 3); (3, 3)]

# List.map
    (fun ((x, _), (_, y)) -> (x,y))
    (List.filter
      (fun ((_, x), (y, _)) -> x=y)
      (List.flatten (List.map (fun x ->
            (List.map (fun y -> (x, y)) r)) r)));;
- : (int * int) list = [(1, 3); (2, 3); (3, 3)]

# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.flatten;;
- : 'a list list -> 'a list = <fun>
# List.filter;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Note: In Scala, we used OO, here we don't (even though it exists in OCAML). List is a module. Our lists are just values, so we say e.g. `List.map f l` rather than `l.map f` as in Scala.

What does it take to write a query?

- On collections: (singleton) collection construction, collection concatenation/union, map ("zoom" into a collection), flatten (the closest we get to collection destruction)
- On tuples: tuple construction, destruction (projection)
- Conditionals (equality)
- A filter operation (this one is actually redundant, but that is not obvious).
- Anything else?

The power of variables used nonlocally

- We missed one important thing since it is not a function to call, but should be:
  - It gives substantial power, and it comes at a cost.
  - Supporting it in an interpreter is nontrivial ($\Rightarrow$ closures).
  - It allows us to nest operations (such as map) in ways that make parallelization nontrivial and are not supported in e.g. MapReduce: need to eliminate it.
- This is the ability to use variables arbitrarily deeply in expressions.
- For instance, x and R are used nonlocally inside expression

```
R.map(x => R.map(y => (x, y)))
```

How do you implement this in MapReduce?

pairWith

We can eliminate this problem by introducing the following new operation pairWith which encapsulates, once and for all, nonlocal variable uses:

```
scala> def pairWith[T,U](x:T, Y:Traversable[U]) =
           Y.map(y => (x,y))
pairWith: [T, U](x: T, Y: Traversable[U])Traversable[(T, U)]

scala> pairWith("Hi", List(1,2))
res1: Traversable[(String, Int)] = List((Hi,1), (Hi,2))
```

Now we can rewrite

```
R.map(x => R.map(y => (x, y)))
```

as

```
pairWith(R, R).map(t => pairWith(t._2, t._1))
```

pairWith

Using pairWith, we can write any relational algebra query using just one variable. Here is the previous query example again, adapted:

```
pairWith(R, R).map(x => pairWith(x._2, x._1)).flatten
  .map(x => (x._1._1, x._1._2, x._2._1, x._2._2))
  .filter(x => x._2 == x._3).map(x => (x._1, x._4))
```

- Since there is just one variable, we don't need a name for it.
- This leads us to a variable-free (but higher-order) algebra called monad algebra.
- Note also: We eliminated the nesting of map-jobs. This is how you do this in MapReduce, where you can't nest map-phases.

```
scala> val R = List((1,2),(2,3),(3,3))
R: List[(Int, Int)] = List((1,2), (2,3), (3,3))

scala> val v1 = pairWith(R, R)
v1: Traversable[(List[(Int, Int)], (Int, Int))] =
   List((List((1,2), (2,3), (3,3)),(1,2)),
        (List((1,2), (2,3), (3,3)),(2,3)),
        (List((1,2), (2,3), (3,3)),(3,3)))

scala> val v2 = v1.map(x => pairWith(x._2, x._1)).flatten
v2: Traversable[((Int, Int), (Int, Int))] =
   List(((1,2),(1,2)), ((1,2),(2,3)), ((1,2),(3,3)),
        ((2,3),(1,2)), ((2,3),(2,3)), ((2,3),(3,3)),
        ((3,3),(1,2)), ((3,3),(2,3)), ((3,3),(3,3)))

scala> val v3 = v2.map(x => (x._1._1, x._1._2, x._2._1, x._2._2))
v3: Traversable[(Int, Int, Int, Int)] =
   List((1,2,1,2), (1,2,2,3), (1,2,3,3), (2,3,1,2),
    (2,3,2,3), (2,3,3,3), (3,3,1,2), (3,3,2,3), (3,3,3,3))

scala> val q = v3.filter(x => x._2 == x._3)
                 .map(x => (x._1, x._4))
q: Traversable[(Int, Int)] = List((1,3), (2,3), (3,3))
```

Monad Algebra

Complex Values and Types

- Complex values constructed from sets, tuples, and atomic values from a single-sorted domain.
- Types are terms of the grammar

$$\tau ::= \mathsf{Dom} \mid \{\tau\} \mid \langle A_1 : \tau_1, \ldots, A_k : \tau_k \rangle$$

  where $k \geq 0$ and $A_1, \ldots, A_k$ are called attribute values.
- The collection type (here, sets) could be replaced by multisets or lists and there will be a language analogous to the one defined next such that most results carry over.

The language $\mathcal{M}$ (monad algebra)

- A query language on complex values consisting of expressions built from the following operations:
  - Collection operations: Singleton construction, map, flatten
  - Tuple operation: Tuple construction, Tuple destruction (projection)
  - Pushing a value (a tuple) into a collection: pairwith.
- $\mathcal{M}$ is a strongly typed language: applying an operation to a value of the wrong type leads to a type error.
- $\mathcal{M}$ is variable free – a query expression is assembled from primitive operations simply by composition. There is nothing else.

Operations of $\mathcal{M}$ (monad algebra)

1. identity

$$\mathsf{id} : x \mapsto x \qquad \tau \to \tau$$

2. composition[1]

$$f \circ g : x \mapsto g(f(x)) \qquad \frac{f : \tau \to \tau',\ g : \tau' \to \tau''}{f \circ g : \tau \to \tau''}$$

3. constants from $\mathsf{Dom} \cup \{\emptyset, \langle\rangle\}$ ($\langle\rangle$ is the nullary tuple)
4. singleton set construction

$$\mathsf{sng} : x \mapsto \{x\} \qquad \tau \to \{\tau\}$$

[1] Convention: $(f \circ g)(x) = g(f(x))$, not $f(g(x))$.

Operations of $\mathcal{M}$ (monad algebra)

5. application of a function to every member of a set

$$\mathsf{map}(f) : X \mapsto \{f(x) \mid x \in X\} \qquad \frac{f : \tau \to \tau'}{\mathsf{map}(f) : \{\tau\} \to \{\tau'\}}$$

6. unnesting sets of sets:

$$\mathsf{flatten} : X \mapsto \bigcup X \qquad \{\{\tau\}\} \to \{\tau\}$$

Example

$$\mathsf{map}(\mathsf{flatten}) : \{\{\{\tau\}\}\} \to \{\{\tau\}\}$$

$$[\![\mathsf{map}(\mathsf{flatten})]\!](\{\{\{1,2\},\{3\}\},\{\{4,5\},\{5,6\}\}\}) := \{\{1,2,3\},\{4,5,6\}\}$$

We use $\mathsf{flatmap}(f)$ as a shortcut for $\mathsf{map}(f) \circ \mathsf{flatten}$.

Operations of $\mathcal{M}$ (monad algebra)

7. pairing

$$\text{pairwith}_{A_1} : \langle A_1 : X_1, A_2 : x_2, \ldots, A_n : x_n \rangle \mapsto \{\langle A_1 : x_1, A_2 : x_2, \ldots, A_n : x_n \rangle \mid x_1 \in X_1\}$$

$$\langle A_1 : \{\tau_1\}, A_2 : \tau_2, \ldots, A_n : \tau_n \rangle \to \{\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle\}$$

(pairwith$_{A_i}$ for $i > 1$ is defined analogously.)

Example

$$[\![\text{pairwith}_{A_1}]\!](\langle A_1 : \{1, 2\}, A_2 : \{3, 4\}\rangle) := \{\langle A_1 : 1, A_2 : \{3, 4\}\rangle, \langle A_1 : 2, A_2 : \{3, 4\}\rangle\}$$

Operations of $\mathcal{M}$ (monad algebra)

8. tuple formation

$$\langle A_1 : f_1, \ldots, A_n : f_n \rangle : x \mapsto \langle A_1 : f_1(x), \ldots, A_n : f_n(x) \rangle$$

$$\frac{f_1 : \tau \to \tau_1, \ldots, f_n : \tau \to \tau_n}{\langle A_1 : f_1, \ldots, A_n : f_n \rangle : \tau \to \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle}$$

Example

$$\begin{array}{rcl} \langle A_1 : \mathrm{id}, A_2 : 3 \rangle & : & \tau \to \langle A_1 : \tau, A_2 : 3 \rangle \\ [\![\langle A_1 : \mathrm{id}, A_2 : 3 \rangle]\!] (\langle A_1 : 1, A_2 : 2 \rangle) & := & \langle A_1 : \langle A_1 : 1, A_2 : 2 \rangle, A_2 : 3 \rangle \end{array}$$

Operations of $\mathcal{M}$ (monad algebra)

9. projection

$$\pi_{A_i} : \langle A_1 : x_1, \ldots, A_i : x_i, \ldots, A_n : x_n \rangle \mapsto x_i$$

$$\pi_{A_i} : \langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle \to \tau_i$$

Example

$$[\![\langle A_1 : \pi_2, A_2 : \pi_1, A_3 : \pi_1 \circ \mathsf{sng} \rangle]\!](\langle A_1 : 1, A_2 : 2 \rangle) := \langle A_1 : 2, A_2 : 1, A_3 : \{1\} \rangle$$

Observe that projection π is applied to tuples rather than to sets of tuples as in relational algebra.

Example

The relational algebra expression π_{AB} corresponds to the expression $\mathsf{map}(\langle A : \pi_A, B : \pi_B \rangle)$ in $\mathcal{M}$.

Example: Products (Cartesian and Relational Algebra)

Example

The Cartesian product $f \times g$ can be defined in $\mathcal{M}$ as

$$\langle 1 : f, 2 : g \rangle \circ \text{pairwith}_1 \circ \text{flatmap}(\text{pairwith}_2).$$

Observe the difference from the product of relational algebra. For instance, the query id $\times$ id on a set of pairs S computes

$$\{\langle \langle x_1, x_2 \rangle, \langle x_3, x_4 \rangle \rangle \mid \langle x_1, x_2 \rangle, \langle x_3, x_4 \rangle \in S\}$$

rather than

$$\{\langle x_1, x_2, x_3, x_4 \rangle \mid \langle x_1, x_2 \rangle, \langle x_3, x_4 \rangle \in S\}.$$

□

Booleans

- Boolean queries ("predicates"): queries that produce values of type $\{\langle\rangle\}$.
- This type has *two* possible values: $\{\langle\rangle\}$ ("true") and $\emptyset$ ("false").
- The logical conjunction $\gamma \wedge \delta$ of two predicates γ and δ can be computed as $\gamma \times \delta$ (the relational algebra product defined on the previous slide).
- Negation $\neg\gamma$ can be computed as $\langle 1 : \mathrm{id}, 2 : \emptyset\rangle \circ =$.
- (Logical disjuction $\gamma \vee \delta$ is redundant, by De Morgan's law.)

Positive Monad Algebra

- By positive monad algebra $\mathcal{M}_{\cup}$, we denote $\mathcal{M}$ extended by the set union operation $\cup$.
- This language has a number of nice properties [5, 1], but it is known to be incomplete as a practical query language because it cannot yet express a value equality predicate

$$(A_i = A_j) : \langle A_1 : \tau_1, \ldots, A_k : \tau_k \rangle \to \{\langle\rangle\}.$$

Equality

- Equality of atomic values, $(A_i =_{atomic} A_j)$ with $\tau_i = \tau_j = \mathrm{Dom}$ is defined as

$$\langle A_1 : x_1, \ldots, A_k : x_k \rangle \mapsto \{\langle\rangle \mid x_i = x_j\}.$$

- "Deep" equality of arbitrary complex values: $=_{deep}$.

Definition ($=_{deep}$)

- $=_{atomic}$ on atomic values,
- inductively holds on tuple values $\langle A_1 : x_1, \ldots, A_k : x_k \rangle$, $\langle A_1 : y_1, \ldots, A_k : y_k \rangle$ of the same type iff $x_1 =_{deep} y_1 \wedge \cdots \wedge x_k =_{deep} y_k$, and
- on set values X, Y of the same type iff for all $x \in X$ there is a $y \in Y$ such that $x =_{deep} y$ and vice versa.

We will often use just $=$ for $=_{deep}$.

Full Monad Algebra

If we extend $\mathcal{M}_\cup$ by any nonempty subset of the operations

- deep value equality ($A =_{deep} B$),
- testing set membership ($A \in B$) or containment ($A \subseteq B$),
- selection $\sigma_{A=B}$,
- set difference "$-$",
- set intersection $\cap$, or
- nesting (as in nested relational algebra),

we always get the same expressive power.[2] We will call any one of these extended languages full monad algebra.

Theorem ([5])

$\mathcal{M}_\cup[=_{deep}] \equiv \mathcal{M}_\cup[\sigma] \equiv \mathcal{M}_\cup[-] \equiv \mathcal{M}_\cup[\cap] \equiv \mathcal{M}_\cup[\subseteq] \equiv \mathcal{M}_\cup[\in] \equiv \mathcal{M}_\cup[nest]$.

[2] No analogous statement can be made about flat relational algebra.

Example

Given a Boolean predicate γ, selection σ_γ can be expressed as

$$\mathsf{flatmap}\big(\langle 1 : \mathsf{id}, 2 : \gamma\rangle \circ \mathsf{pairwith}_2 \circ \mathsf{map}(\pi_1)\big).$$

Example

Predicate $(A \subseteq B)$ can be expressed in $\mathcal{M}_\cup[=_{deep}, \sigma]$ as

$$\langle A : \pi_A, A' : \pi_A \cap \pi_B\rangle \circ (A =_{deep} A')$$

where

$$f \cap g := (f \times g) \circ \sigma_{1=2} \circ \mathsf{map}(\pi_1).$$

A predicate $(f \subseteq g)$ can be expressed as

$$\langle 1 : f, 2 : g\rangle \circ (1 \subseteq 2).$$

□

Example

Predicate $\neq$ can be expressed in $\mathcal{M}_{\cup}[=_{deep}]$ as $=_{deep} \circ \neg$.

Example

Predicate $\subset$ can be expressed in $\mathcal{M}_{\cup}[=_{deep}, \subseteq]$ as $\subseteq \wedge \neq$.

Example

Given a complex value of type $\langle R : \{\tau\}, S : \{\tau\}\rangle$, difference $R - S$ can be implemented in $\mathcal{M}_{\cup}[\sigma]$ as

$$\text{pairwith}_R \circ \text{map}\big(\langle R : \pi_R, S_R : \text{pairwith}_S \circ \sigma_{R=S}\rangle\big) \circ \sigma_{S=\emptyset} \circ \text{map}(\pi_R).$$

Idea: Compute, for each r of R, the set S_R of elements in S that are equal to r. Select those elements r of R for which S_R is empty.

	$\langle R : \{1, 2\}, S : \{2, 4\}\rangle$
pairwith_R	$\{\langle R : 1, S : \{2, 4\}\rangle, \langle R : 2, S : \{2, 4\}\rangle\}$
$\text{map}(\dots)$	$\{\langle R : 1, S_R : \emptyset\rangle, \langle R : 2, S_R : \{\langle R : 2, S : 2\rangle\}\rangle\}$
$\sigma_{S=\emptyset}$	$\{\langle R : 1, S_R : \emptyset\rangle\}$
$\text{map}(\pi_R)$	$\{1\}$

The Conservativity Theorem

Conservativity of Full Monad Algebra

- Full monad algebra is a conservative extension of relational algebra.
- By a flat relational database, we denote a relational database in the classical sense.
- k-ary flat relation: type $\{\langle A_1 : \mathrm{Dom}, \ldots, A_k : \mathrm{Dom}\rangle\}$.

Theorem ([4])

A mapping from a (flat) relational database to a (flat) relation is expressible in $\mathcal{M}_{\cup}[=_{deep}]$ if and only if it is expressible in relational algebra.

Corollary

All relational calculus queries can be expressed in $\mathcal{M}_{\cup}[=_{deep}]$.

Conservativity of Full Monad Algebra

We have already seen that relational algebra can be mapped to full monad algebra. Other direction (conservativity):

- Initially, tuples are their own key values (cf. [2] Fig. 5; Thm. 5.2).
  - Copy columns to save a separate key for each tuple. $R(A, B)$ represented by $R(K_A, K_B, A, B)$.
  - On projection, leave keys unchanged. Schema of $\pi_A(R)$ is $R(K_A, K_B, A)$.
- Have proper representation of empty sets:
  1. special symbol (subsequently requires case distinctions),
  2. domain-to-bool map to capture set membership (see Koch, ICDT 2009), or
  3. separate tables for nested sets (used next).
- Implementation of monad algebra operation in relational algebra on flat representation straightforward.

Conservativity of Full Monad Algebra

Example:

R	$R.K_A$	$R.A$
	1	1
	2	2

S	$S.K_A$	$S.A$
	2	2
	4	4

The result of pairwith_R is represented by table R plus

$\text{pairwith}_R.S$	$R.K_A$	$S.K_A$	$S.A$
	1	2	2
	1	4	4
	2	2	2
	2	4	4

Category-theoretic Roots

Monads

- Monad algebra is named after monads (a concept from category theory).
- The query language is a monad with tensorial strength.
- This governs the algebraic equivalences that hold in the language and can be used for query rewriting and optimization.
- "Tensorial strength" refers to the support for tuples.
- The monad commutative diagrams (rewrite rules) focus on constructing, mapping into, and unnesting collection types.
- These diagrams can be looked up in standard category theory textbooks such as Mac Lane [3].

Monads, ctd.

- Monad: monoid (map, sng, flatten) in category of endofunctors map($\cdot$).
- Two natural transformations sng (the 1-element) and flatten (the binary operation of the monoid).
- Essentially, the core monad diagrams dictate
  1. On types $\{\tau\}$,

     $$\text{sng} \circ \text{flatten} = \text{map}(\text{sng}) \circ \text{flatten} = \text{id}$$

     ($\approx$ existence of the neutral element)
  2. On types $\{\{\{\tau\}\}\}$,

     $$\text{flatten} \circ \text{flatten} = \text{map}(\text{flatten}) \circ \text{flatten}$$

     ($\approx$ associativity)
- But monads are much more abstract, and sets plus the natural transformations sng and flatten are just an example!
- The diagrams get much more interesting and useful for monads with tensorial strength, see [1]!

Monad Algebra with Powerset

Limits of Expressive Power

- Conservativity: e.g., transitive closure is not expressible in full monad algebra.
- We will consider an extension using a powerset operation

The Powerset Operation

- Powerset operation

$$\text{pow} : \{\tau\} \to \{\{\tau\}\} \qquad \text{pow} : S \mapsto 2^S$$

Example

$[\![\text{pow}]\!](\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

- $\mathcal{M}_\cup[=_{deep}, \text{pow}]$ is extremely expressive: can express all ELEMENTARY queries.
- ELEMENTARY is the complexity class of all problems solvable in time $O(2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}})$, for any fixed tower of twoes.
- Can do much more than e.g. datalog, but this is not necessarily an advantage. For example, transitive closure is expressible but needs the powerset operation – very inefficient.

The Powerset Operation

Example

Encode 3-colorability on graphs (V, E).

$$\langle V : \pi_V, E : \pi_E, R : \pi_V \circ \mathsf{pow}, G : \pi_V \circ \mathsf{pow}, B : \pi_V \circ \mathsf{pow} \rangle \circ \mathsf{pairwith}_R \circ \mathsf{flatmap}(\mathsf{pairwith}_G) \circ \mathsf{flatmap}(\mathsf{pairwith}_B) \circ \mathsf{flatmap}(f_{3col})$$

f_{3col} is the $\mathcal{M}_\cup[=_{deep}]$ version of a first-order formula

$$\phi_{3col} := \big(\forall x\ V(x) \Rightarrow \big((R(x) \vee G(x) \vee B(x)) \wedge \neg\phi_{\neq}(x,x)\big)\big) \wedge \forall x \forall y\ E(x,y) \Rightarrow \phi_{\neq}(x,y)$$

with

$$\phi_{\neq}(x,y) := \bigvee_{P,Q \in \{R,G,B\}, P \neq Q} P(x) \wedge Q(y)$$

expressing that given a tuple $\langle V, E, R, G, B \rangle$, (R, G, B) is a 3-coloring of the graph (V, E).

XQuery

XQuery

- Collection type: lists. XML nodes encoded as (label, child list) pairs.
- Incomplete XQuery to monad algebra mapping:

$$\begin{aligned} \langle a \rangle f \langle /a \rangle &\mapsto \langle \mathit{Label} : a, \mathit{Children} : f \rangle \\ \text{for } \$x \text{ in } f \text{ return } g &\mapsto f \circ \text{flatmap}(g) \\ \$x/a &\mapsto \pi_{\mathit{Children}} \circ \sigma_{\mathit{Label}=a} \\ \text{if } f \text{ then } g &\mapsto \text{sng} \circ \sigma_f \circ \text{flatmap}(g) \end{aligned}$$

- Note: always create a set to handle the else case; has to be flattened out later.
- This is quite central and explicit to the formal semantics of XQuery, not just a hack in my mapping to monad algebra: cannot nest lists in XQuery!
- Things are not quite as simple: the mapping for for-loops does not make the environment (variables introduced in superexpressions) available to g. Correct encoding in [2].

Exercises

Exercises

1. Write ϕ_{3col} in full monad algebra. (Note: this is a lot of work!)
2. Show that *nest* can be implemented in $\mathcal{M}_{\cup}[\sigma]$.
3. Given a complex value of type $\langle R : \{\tau\}, S : \{\tau\}\rangle$, show that difference $R - S$ can be implemented in $\mathcal{M}_{\cup}[\in]$.

Exercises

4. What does the query

$$\pi_E \circ \mathsf{nest}_{T'=(T)} \circ (\mathsf{id} \times \mathsf{id}) \circ \mathsf{flatmap}((\pi_1 \circ \pi_{T'}) \subset (\pi_2 \circ \pi_{T'}))$$

do on a graph given as a complex value of type

$$\langle V : \{\mathsf{Dom}\}, E : \{\langle F : \mathsf{Dom}, T : \mathsf{Dom}\rangle\}\rangle?$$

Write this query also in FO.

Exercises

5. Express the graph reachability problem in directed graphs in full monad algebra with powerset. Graph reachability is the question of deciding whether there is a path from a given start node to a given end node.
   The problem instances are given as tuples

   $$\langle V : \{\mathrm{Dom}\}, E : \{\langle v_1 : \mathrm{Dom}, v_2 : \mathrm{Dom}\rangle\}, \mathit{start} : \mathrm{Dom}, \mathit{end} : \mathrm{Dom}\rangle.$$

   with (V, E) the graph and $\mathit{start}, \mathit{end} \in V$ the start and end node.
   It is ok to give part of the solution as a first-order formula.

Solution of Exercise 4

The query checks whether there are two nodes v, w such that v has an arrow to all nodes that w has an arrow to, plus at least one other node.

$$\exists x \exists y \; (\forall z \; E(y,z) \Rightarrow E(x,z)) \wedge \exists z' \; E(x,z') \wedge \neg E(y,z')$$

(We do not need to require $x \neq y$ because x and y do not have the same number of neighbours and cannot be identical.)

Solution of Exercise 5

- Datalog program for computing set R of nodes reachable from *start*:

$$\begin{aligned} R(\mathit{start}) &\leftarrow . \\ R(y) &\leftarrow R(x), E(x,y). \end{aligned}$$

- Minimal model semantics of datalog (true in the minimal model = true in the intersection of all models): for all sets R,

$$(R(\mathit{start}) \wedge \forall x \forall y\ R(x) \wedge E(x,y) \Rightarrow R(y)) \Rightarrow R(\mathit{end})$$

- Check that there does not exist a set of nodes R with

$$\phi := (R(\mathit{start}) \wedge \forall x \forall y\ R(x) \wedge E(x,y) \Rightarrow R(y)) \wedge \neg R(\mathit{end})$$

- This is FO! Thus the $\mathcal{M}_\cup[=, \mathrm{pow}]$ query is

$$\langle R : \pi_V \circ \mathrm{pow}, E : \pi_E, \mathit{start} : \pi_{\mathit{start}}, \mathit{end} : \pi_{\mathit{end}} \rangle \circ$$
$$\mathrm{pairwith}_R \circ \mathrm{flatmap}(\phi) \circ \neg$$

Bibliography


P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong.
"Principles of Programming with Complex Objects and Collection Types".
Theor. Comput. Sci., **149**(1):3–48, 1995.


C. Koch.
"On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values".
ACM Transactions on Database Systems, **31**(4), 2006.


S. Mac Lane.
Categories for the Working Mathematician.
Springer Graduate Texts in Mathematics, 1998.


J. Paredaens and D. Van Gucht.
"Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions".
In *Proc. PODS*, pages 29–38, 1988.


V. Tannen, P. Buneman, and L. Wong.
"Naturally Embedded Query Languages".
In *Proc. of the 4th International Conference on Database Theory (ICDT)*, pages 140–154, 1992.