Relational algebra by way of adjunctions
Gibbons, Jeremy; Henglein, Fritz; Hinze, Ralf; Wu, Nicolas

Publication date:
2016

Document Version
Early version, also known as pre-print

Citation for published version (APA):
Relational Algebra by Way of Adjunctions

Jeremy Gibbons
(joint work with Fritz Henglein, Ralf Hinze, Nicolas Wu)
DBPL, October 2015
1. Summary

- bulk types (sets, bags, lists) are *monads*
- monads have nice *mathematical foundations via adjunctions*
- monads support *comprehensions*
- comprehension syntax provides a *query notation*

```
[ (customer.name, invoice.amount)
 | customer ← customers,
     invoice ← invoices,
     customer.cid = invoice.customer,
     invoice.due ≤ today ]
```

- monad structure explains *selection, projection*
- less obvious how to explain *join*
2. Galois connections

Relating monotonic functions between two ordered sets:

\[(A, \leq) \perp (B, \sqsubseteq) \quad \text{means } f \ b \leq a \iff b \sqsubseteq g \ a\]

For example,

\[(\mathbb{R}, \leq_{\mathbb{R}}) \perp (\mathbb{Z}, \leq_{\mathbb{Z}}) \quad \text{and} \quad (\mathbb{Z}, \leq) \perp (\mathbb{Z}, \leq)\]

“Change of coordinates” can sometimes simplify reasoning; eg rhs gives \(n \times k \leq m \iff n \leq m \div k\), and multiplication is easier to reason about than rounding division.
3. Category theory from ordered sets

A *category* $\mathcal{C}$ consists of

- a set* $|\mathcal{C}|$ of *objects*,
- a set* $\mathcal{C}(X, Y)$ of *arrows* $X \rightarrow Y$ for each $X, Y : |\mathcal{C}|$,
- *identity* arrows $id_X : X \rightarrow X$ for each $X$
- *composition* $f \cdot g : X \rightarrow Z$ of compatible arrows $g : X \rightarrow Y$ and $f : Y \rightarrow Z$,
- such that composition is associative, with identities as units.

Think of a directed graph, with vertices as objects and paths as arrows.

An ordered set $(A, \leq)$ is a degenerate category, with objects $A$ and a unique arrow $a \rightarrow b$ iff $a \leq b$.

```
... → −2 → −1 → 0 → 1 → 2 → ...  
```

Many categorical concepts are generalisations from ordered sets.

*proviso...*
4. Concrete categories

Ordered sets are a *concrete category*: roughly,

- the objects are *sets with additional structure*
- the arrows are *structure-preserving mappings*

Many useful categories are of this form.

For example, the category $\mathbf{CMon}$ has commutative monoids $(M, \otimes, \epsilon)$ as objects, and homomorphisms $h : (M, \otimes, \epsilon) \to (M', \oplus, \epsilon')$ as arrows:

\[
h (m \otimes n) = h m \oplus h n \\
h \epsilon = \epsilon'
\]

Trivially, category $\mathbf{Set}$ has sets as objects, and total functions as arrows.
5. Functors

Categories are themselves structured objects...

A functor $F : C \to D$ is an operation on both objects and arrows, preserving the structure: $F f : F X \to F Y$ when $f : X \to Y$, and

$F id_X = id_{F X}$
$F (f \cdot g) = F f \cdot F g$

For example, forgetful functor $U : \text{CMon} \to \text{Set}$:

$U (M, \otimes, \epsilon) = M$
$U (h : (M, \otimes, \epsilon) \to (M', \oplus, \epsilon')) = h : M \to M'$

Conversely, $\text{Free} : \text{Set} \to \text{CMon}$ generates the free commutative monoid (ie bags) on a set of elements:

$\text{Free } A = (\text{Bag } A, \cup, \emptyset)$
$\text{Free } (f : A \to B) = \text{map } f : \text{Bag } A \to \text{Bag } B$
6. Adjunctions

*Adjunctions* are the categorical generalisation of Galois connections.

Given categories $\mathbf{C}$, $\mathbf{D}$, and functors $L : \mathbf{D} \to \mathbf{C}$ and $R : \mathbf{C} \to \mathbf{D}$, adjunction

$$\begin{array}{c}
\mathbf{C} \perp \mathbf{D} \\
\downarrow L \\
\downarrow R
\end{array}$$

means* $[-] : \mathbf{C}(L X, Y) \cong \mathbf{D}(X, R Y) : [-]$.

A familiar example is given by *currying*:

$$\begin{array}{c}
\mathbf{Set} \perp \mathbf{Set} \\
\downarrow (-)P \\
\downarrow (-)^P
\end{array}$$

with $curry : \mathbf{Set}(X \times P, Y) \cong \mathbf{Set}(X, Y^P) : curry^\circ$

hence definitions and properties of $apply = \text{uncurry} \ id_{Y^P} : Y^P \times P \to Y$.
7. Products and coproducts

\[
\begin{array}{c}
\text{Set} \quad \bot \quad \text{Set}^2 \quad \bot \quad \text{Set} \\
\Delta \quad + \quad \Delta \\
\end{array}
\]

with

\[
\begin{align*}
\text{fork} : \text{Set}^2(\Delta A, (B, C)) & \simeq \text{Set}(A, B \times C) : \text{fork}^* \\
\text{junc}^* : \text{Set}(A + B, C) & \simeq \text{Set}^2((A, B), \Delta C) : \text{junc}
\end{align*}
\]

hence

\[
\begin{align*}
\text{dup} &= \text{fork } \text{id}_{A, A} : \text{Set}(A, A \times A) \\
(fst, snd) &= \text{fork}^* \text{id}_{B \times C} : \text{Set}^2(\Delta (B, C), (B, C))
\end{align*}
\]

give tupling and projection. Dually for sums and injections, and generally for any arity—even zero.
8. Free commutative monoids

Adjunctions often capture embedding/projection pairs:

\[
\text{CMon} \quad \perp \quad \text{Set}
\]

with \([-] : \text{CMon}(\text{Free } A, (M, \otimes, \varepsilon)) \simeq \text{Set}(A, U (M, \otimes, \varepsilon)) : [-]\]

Unit and counit:

\[
single A = [id_{\text{Free } A}] : A \to U (\text{Free } A)
\]
\[
reduce M = [id_M] : \text{Free } (U M) \to M \quad \text{-- for } M = (M, \otimes, \varepsilon)
\]

whence, for \(h : \text{Free } A \to M\) and \(f : A \to U M = M\),

\[
h = reduce M \cdot \text{Free } f \iff U h \cdot single A = f
\]

ie 1-to-1 correspondence between homomorphisms from the free commutative monoid (bags) and their behaviour on singletons.
## 9. Aggregation

Aggregations are bag homomorphisms:

<table>
<thead>
<tr>
<th>aggregation</th>
<th>monoid</th>
<th>action on singletons</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>((\mathbb{N}, 0, +))</td>
<td>({a} \mapsto 1)</td>
</tr>
<tr>
<td>sum</td>
<td>((\mathbb{R}, 0, +))</td>
<td>({a} \mapsto a)</td>
</tr>
<tr>
<td>max</td>
<td>((\mathbb{Z}, \text{minBound}, \text{max}))</td>
<td>({a} \mapsto a)</td>
</tr>
<tr>
<td>min</td>
<td>((\mathbb{Z}, \text{maxBound}, \text{min}))</td>
<td>({a} \mapsto a)</td>
</tr>
<tr>
<td>all</td>
<td>((\mathbb{B}, \text{True}, \land))</td>
<td>({a} \mapsto a)</td>
</tr>
<tr>
<td>any</td>
<td>((\mathbb{B}, \text{False}, \lor))</td>
<td>({a} \mapsto a)</td>
</tr>
</tbody>
</table>

Selection is a homomorphism, to bags, using action

\[
guard : (A \to \mathbb{B}) \to \text{Bag } A \to \text{Bag } A
\]

\[
guard p a = \text{if } p a \text{ then } \{a\} \text{ else } \emptyset
\]

Laws about selections follow from laws of homomorphisms (and of coproducts, since \(\mathbb{B} = 1 + 1\)).
10. Monads

Bags form a monad \((\text{Bag}, \text{union}, \text{single})\) with

\[
\text{Bag} = U \cdot \text{Free}
\]

\[
\text{union} : \text{Bag} (\text{Bag} A) \to \text{Bag} A
\]

\[
\text{single} : A \to \text{Bag} A
\]

which justifies the use of comprehension notation \(\{ f \ a \ b \mid a \leftarrow x, b \leftarrow g \ a \}\).

In fact, for any adjunction \(L \dashv R\) between \(\mathbf{C}\) and \(\mathbf{D}\), we get a monad \((T, \mu, \eta)\) on \(\mathbf{D}\), where

\[
T = R \cdot L
\]

\[
\mu A = R [id_A] L : T (T A) \to T A
\]

\[
\eta A = [id_A] : A \to T A
\]
11. Maps

Database indexes are essentially maps $\text{Map } K V = V^K$. Maps $(-)^K$ from $K$ form a monad (the Reader monad in Haskell), so arise from an adjunction.

The laws of exponents arise from this adjunction, and from those for products and coproducts:

\[
\begin{align*}
\text{Map } 0 V & \simeq 1 \\
\text{Map } 1 V & \simeq V \\
\text{Map } (K_1 + K_2) V & \simeq \text{Map } K_1 V \times \text{Map } K_2 V \\
\text{Map } (K_1 \times K_2) V & \simeq \text{Map } K_1 (\text{Map } K_2 V) \\
\text{Map } K 1 & \simeq 1 \\
\text{Map } K (V_1 \times V_2) & \simeq \text{Map } K V_1 \times \text{Map } K V_2 : \text{merge}
\end{align*}
\]
12. Indexing

Relations are in 1-to-1 correspondence with set-valued functions:

\[
\begin{array}{c}
\text{Rel} \\
\downarrow \quad \downarrow \\
\text{Set}
\end{array}
\quad \rotatebox{90}{$\Rightarrow$} \quad
\begin{array}{c}
\text{Set} \\
\downarrow \quad \downarrow \\
\text{Rel}
\end{array}
\]

where \( J \) embeds, and \( E \ R : A \to \text{Set} B \) for \( R : A \sim B \).

Moreover, the correspondence remains valid for bags:

\[
\text{index} : \text{Bag} (K \times V) \simeq \text{Map} K (\text{Bag} V)
\]

Together, \( \text{index} \) and \( \text{merge} \) give efficient relational joins:

\[
x f \Join g y = \text{flatten} (\text{Map} K \ cp (\text{merge} (\text{groupBy} f \ x, \text{groupBy} g \ y)))
\]

\[
\text{groupBy} : (V \to K) \to \text{Bag} V \to \text{Map} K (\text{Bag} V)
\]

\[
\text{flatten} : \text{Map} K (\text{Bag} V) \to \text{Bag} V
\]
13. Pointed sets and finite maps

Model finite maps $\text{Map}_*$ not as partial functions, but total functions to a pointed codomain $(A, a)$, i.e. a set $A$ with a distinguished element $a : A$.

Pointed sets and point-preserving functions form a category $\text{Set}_*$. There is an adjunction to $\text{Set}$, via

$$
\begin{array}{ccc}
\text{Set}_* & \perp & \text{Set} \\
\text{Maybe} & \rightarrow & \text{U} \\
\rightarrow & & \leftarrow
\end{array}
$$

where $\text{Maybe } A \simeq 1 + A$ adds a point, and $\text{U } (A, a) = A$ discards it.

In particular, $(\text{Bag } A, \emptyset)$ is a pointed set. Moreover, $\text{Bag } f$ is point-preserving, so we get a functor $\text{Bag}_* : \text{Set} \rightarrow \text{Set}_*$.

Indexing remains an isomorphism:

$$
\text{index } : \text{Bag}_* (K \times V) \simeq \text{Map}_* K (\text{Bag}_* V)
$$
14. Graded monads

A catch: finite maps aren’t a monad, because
\[ \eta a = \lambda k \rightarrow a : A \rightarrow \text{Map} \ K \ A \]
in general yields an infinite map.

However, finite maps are a \textit{graded monad}*: for monoid \( (M, \otimes, \epsilon) \),
\[
\begin{align*}
\mu X &: T_m (T_n X) \rightarrow T_{m \otimes n} X \\
\eta X &: X \rightarrow T_{\epsilon} X
\end{align*}
\]
satisfying the usual laws. These too arise from adjunctions*.

We use the monoid \((\mathbb{K}, \times, 1)\) of finite key types under product.
15. Conclusions

- *Monad comprehensions* for database queries
- Structure arising from *adjunctions*
- Equivalences from *universal properties*
- Fitting in *relational joins*, via indexing
- To do: calculating *query optimisations*

Thanks to EPSRC *Unifying Theories of Generic Programming* for funding.