Module Theory and Query Processing

Fritz Henglein and Mikkel Kragh Mathiesen
DIKU, University of Copenhagen

8th Workshop on Mathematically Structured Functional Programming (MSFP)
2020-09-01

Triangle queries

- How many reference triangles are there on Wikipedia?
  - A references B, which references C, which references A.

Experiment (Mathiesen, 2016):

- Input: 335730 reference pairs between Wikipedia pages.
- MySQL: SQL join query, in-memory database, query optimization, indexing
- Haskell: 3 pairwise join functions applied (A with B, B with C, C with A), no preprocessing

Implementation	Execution time (sec)
MySQL	6540
Haskell	

Triangle queries

- How many reference triangles are there on Wikipedia?
  - A references B, which references C, which references A.

Experiment (Mathiesen, 2016):

- Input: 335730 reference pairs between Wikipedia pages.
- MySQL: SQL join query, IMDB, query optimization, indexing
- Haskell: 3 pairwise join functions applied (A with B, B with C, C with A), no preprocessing

Implementation	Execution time (sec)
MySQL	6540
Haskell	4

Strategy

- Consider a classic problem, say query processing
- Forsake the old ways (relational algebra, SQL, etc.)
- Take an algebraic approach (modules)
- Sprinkle category theory on top
- ...
- Profit: generalise previous results, generate new results

Modules

A module $\mathcal{V}$ over commutative ring $\mathcal{K}$ consists of

- A set $|\mathcal{V}|$.
- An element $0_{\mathcal{V}} : |\mathcal{V}|$
- An operation $+ : |\mathcal{V}| \times |\mathcal{V}| \to |\mathcal{V}|$
- An operation $\cdot : |\mathcal{K}| \times |\mathcal{V}| \to |\mathcal{V}|$

such that

- $0_{\mathcal{V}} + x = x$ (zero identity)
- $(x + y) + z = x + (y + z)$ (associativity)
- $x + y = y + x$ (commutativity)
- $1_{\mathcal{K}} \cdot x = x$ (scalar identity)
- $(\alpha\beta) \cdot x = \alpha \cdot (\beta \cdot x)$ (associativity)
- $(\alpha + \beta) \cdot x = \alpha \cdot x + \beta \cdot x$ (distributivity)
- $\alpha \cdot (x + y) = \alpha \cdot x + \alpha \cdot y$ (distributivity)

Linear Maps

A *linear* map $f : \mathcal{U} \to \mathcal{V}$ respects the module structure:

$$f(x+y) = f(x) + f(y)$$
$$f(\alpha x) = \alpha f(x)$$

A *bilinear* map $f : \mathcal{U}_1 \times \mathcal{U}_2 \to \mathcal{V}$ is linear in each argument:

$$f(x_1 + x_2, y) = f(x_1, y) + f(x_2, y)$$
$$f(x, y_1 + y_2) = f(x, y_1) + f(x, y_2)$$
$$f(\alpha x, y) = \alpha f(x, y)$$
$$f(x, \alpha y) = \alpha f(x, y)$$

Modules over $\mathcal{K}$ with linear maps form a category.

Basic Modules

- The trivial module $\{0\}$ with only a zero element.
- The ring $\mathcal{K}$ is a module.
- Linear maps $\mathcal{U} \to \mathcal{V}$ form a module with pointwise operations.

Coproducts: Universal property

$$\begin{tikzcd} \coprod_{i:I} \mathcal{V}_i \arrow[d, dashed, "\mathbf{case}\langle i.c_i \rangle"'] & \mathcal{V}_j \arrow[l, "\mathbf{inj}_j"'] \arrow[dl, "c_j"] \\ \mathcal{W} & \end{tikzcd}$$

Write:

- $\mathcal{V}_1 \oplus \mathcal{V}_2 = \coprod_{i:\{1,2\}} \mathcal{V}_i$,
- $x_1 \oplus x_2 = \mathbf{inj}_1(x_1) + \mathbf{inj}_2(x_2)$.

Coproducts: Natural Isomorphisms

$$
\begin{aligned}
\coprod_{0} \mathcal{V} &\cong \{0\} \\
\coprod_{1} \mathcal{V} &\cong \mathcal{K} \\
\coprod_{I+J} \mathcal{V} &\cong \coprod_{I} \mathcal{V} \oplus \coprod_{J} \mathcal{V} \\
\coprod_{I \times J} \mathcal{V} &\cong \coprod_{I} \coprod_{J} \mathcal{V}
\end{aligned}
$$

This is precisely the structure of generic tries.

Polysets: Universal property

Let $\mathcal{K} = \mathbb{Z}$.

$$\begin{tikzcd} {|\mathbf{P}[B]|} \arrow[d, dashed, "{|\mathbf{ext}\langle b.f(b)\rangle|}"'] & B \arrow[l, "{[\cdot]}"'] \arrow[dl, "f"] \\ {|\mathcal{W}|} & \end{tikzcd}$$

We have $\mathbf{P}[B] \cong \coprod_B \mathbb{Z}$.

Polysets: Programming

- Elements are polysets: *finite sets*

$$\{b_1^{(k_1)}, \ldots, b_m^{(k_m)}\} = k_1 \cdot [b_1] + \ldots + k_m \cdot [b_m]$$

  where $b_1, \ldots, b_m \in B$ and each element carries a *multiplicity* $0 \neq k_i \in \mathbb{Z}$.
    - All unlisted $b \in B$ implicitly have multiplicity 0.
- Application of $f = \mathbf{ext}\langle b.v_b \rangle$ to polyset:

$$f(k_1 \cdot [b_1] + \ldots + k_m \cdot [b_m]) \quad = \quad k_1 \cdot v_{b_1} + \ldots + k_m \cdot v_{b_m}$$

Tensor Products (Property)

$$\begin{tikzcd} \mathcal{U}\otimes\mathcal{V} \arrow[d, dashed, "\mathbf{uncurry}\langle f\rangle"'] & \mathcal{U}\times\mathcal{V} \arrow[l, "\otimes"'] \arrow[dl, "f"] \\ \mathcal{W} & \end{tikzcd}$$

Tensor Products (Programming)

- Any $x : \mathcal{U} \otimes \mathcal{V}$ can be thought of as $y_1 \otimes z_1 + \ldots + y_n \otimes z_n$ where $y_i : \mathcal{U}$ and $z_i : \mathcal{V}$.
- Mapping out can be done by pattern matching:

$$f(y \otimes z) = E \quad \rightsquigarrow \quad f = \mathbf{uncurry}\langle \lambda y.\lambda z.E \rangle$$

- No non-zero natural map $\mathcal{U} \otimes \mathcal{V} \to \mathcal{U}$, but $\mathcal{U} \otimes \mathbf{P}[B] \to \mathcal{U}$ is possible.
- Functorial action is $(f \otimes g)(y \otimes z) = f(y) \otimes g(z)$.

Query processing via multilinear functions

- Union, difference, selection and projection are *linear*.
- Cartesian product is *bilinear*.
- Equi-joins are *bilinear*.
- Aggregation is *linear* if the aggregation function is linear.

Idea:

- Interpret query functions as (multi)linear maps over *polysets* (= fast).
- Add nonlinear (= expensive) conversions to multisets (raise multiplicity to ≥ 0) and sets (lower multiplicity to ≤ 1) *only where needed*.

Joins (Efficient Implementation)

$$\mathbf{index}\langle f\rangle : \mathbf{P}[B] \to \coprod_A \mathbf{P}[B]$$
$$\mathbf{index}\langle f\rangle([b]) = \mathbf{inj}_{f(b)}([b])$$

$$\mathbf{flatten} : \coprod_A \mathcal{V} \to \mathcal{V}$$
$$\mathbf{flatten}(\mathbf{inj}_i(x)) = x$$

$$\mathbf{merge}\langle I\rangle : (\coprod_A \mathcal{U}) \otimes (\coprod_A \mathcal{V}) \to \coprod_A (\mathcal{U} \otimes \mathcal{V})$$

$$(f \bowtie g) = \mathbf{flatten} \circ \mathbf{merge}\langle I\rangle \circ (\mathbf{index}\langle f\rangle \otimes \mathbf{index}\langle g\rangle)$$

Joins (Merging)

$$\alpha : \coprod_{A_1+A_2} \mathcal{V} \cong (\coprod_{A_1} \mathcal{V}) \oplus (\coprod_{A_2} \mathcal{V})$$
$$\beta : \coprod_{A_1\times A_2} \mathcal{V} \cong (\coprod_{A_1} \coprod_{A_2} \mathcal{V})$$

$$\mathbf{merge}\langle\mathbb{Z}\rangle = \mathbf{intmerge}$$
$$\mathbf{merge}\langle A_1 + A_2\rangle = \alpha^{-1} \circ (\mathbf{merge}\langle A_1\rangle \oplus \mathbf{merge}\langle A_2\rangle) \circ (\alpha \otimes \alpha)$$
$$\mathbf{merge}\langle A_1 \times A_2\rangle = \beta^{-1} \circ \coprod_{A_1}(\mathbf{merge}\langle A_2\rangle) \circ \mathbf{merge}\langle A_1\rangle \circ (\beta \otimes \beta)$$

Joins (Efficiency)

- **merge** runs in linear time if **intmerge** does.
- Size of output representation is linear due to symbolic tensor products.

Three Way Joins (Merging)

For convenience define:

$$\triangleright : (\coprod_A \mathcal{U}) \otimes (\mathcal{U} \to \mathcal{V}) \to \coprod_A \mathcal{V}$$
$$x \triangleright f = (\coprod_A f)(x)$$

$$\begin{aligned}
&\mathbf{merge}'\langle A_1, A_2, A_3\rangle(x \otimes y \otimes z) \\
&\quad = \mathbf{merge}\langle A_1\rangle(x \otimes y) \\
&\qquad \triangleright \lambda(x' \otimes y').\mathbf{merge}\langle A_2\rangle(x' \otimes z) \\
&\qquad\quad \triangleright \lambda(x'' \otimes z').\mathbf{merge}\langle A_3\rangle(y' \otimes z') \\
&\qquad\qquad \triangleright \lambda(y'' \otimes z'').x'' \otimes y'' \otimes z''
\end{aligned}$$

Three Way Joins (Efficiency)

- For inputs all of size n, $\mathbf{merge}'$ runs in time $O(n\sqrt{n})$.
- In general, it is *worst-case optimal*.
- Practical advantage, especially for cyclic joins: 4 seconds versus 1 hour 49 minutes for MySQL.

Summary

- Categorical development of linear algebra.
- Connection with databases and queries.
- Efficient data representations.
- An efficient join algorithm.

Linear algebra as a query processing language:

- Quite expressive.
- Functorial and natural constructions.
- Symbolic representations, especially tensor products.
- Efficient joins.