Comprehending Monadic Queries

Jeremy Gibbons
(joint work with Fritz Henglein, Ralf Hinze, Nicolas Wu)
WG2.11#15, November 2015

1. Comprehensions

- ZF axiom schema of specification:

$$\{x^2 \mid x \in Nat \wedge x < 10 \wedge x \text{ is even}\}$$

- SETL set-formers:

$$\{x * x : x \textbf{ in } \{0\,..\,9\} \mid x \textbf{ mod } 2 = 1\}$$

- Eindhoven Quantifier Notation:

$$(x : 0 \leqslant x < 10 \wedge x \text{ is even} : x^2)$$

- Haskell (NPL, Python, ...) list comprehensions:

$$[x \hat{} 2 \mid x \leftarrow [0\,..\,9], even\ x]$$

2. Relational algebra vs calculus

Consider two database tables:

$$\begin{aligned} &\mathit{customers} : \underline{\mathit{cid}}, \mathit{name}, \mathit{address} \\ &\mathit{invoices} \quad : \underline{\mathit{iid}}, \mathit{customer}, \mathit{amount}, \mathit{due} \end{aligned}$$

A query in relational *algebra* ('point-free', on relations):

$$\pi_{\mathit{name},\mathit{amount},\mathit{address}} \left(\sigma_{\mathit{due}<\mathit{today}} \left(\mathit{customers} \bowtie_{\mathit{cid}=\mathit{customer}} \mathit{invoices}\right)\right)$$

The same query in relational *calculus* ('point-wise', on tuples):

```
SELECT  name, amount, address
FROM    customers, invoices
WHERE   cid = customer AND due < today
```

The algebraic style may be convenient for formal manipulation,
but the calculus style is much more accessible for readers.
DBMSs typically translate from calculus-style input to algebra-style
intermediate representation.

3. Comprehending queries

Trinder (1991) argued for comprehensions as a query notation:

$$
\begin{array}{l}
[\ (c.name, c.address, i.amount) \\
\mid c \leftarrow customers, \\
\quad i \leftarrow invoices, \\
\quad c.cid == i.customer, \\
\quad i.due < today\ ]
\end{array}
$$

Very influential observation in the DBPL community.
Formed the basis of languages such as Buneman's *Kleisli*, Microsoft *LINQ*, Wadler's *Links*, as well as querying for objects (*OQL*) and XML (*XQuery*).

4. Comprehending monads (Wadler 1992)

The necessary structure is that of a *monad* $(\mathsf{T}, \gg\!=, \mathit{return})$:

$$
\begin{array}{ll}
(\gg\!=) :: \mathsf{T}\ a \to (a \to \mathsf{T}\ b) \to \mathsf{T}\ b & (x \gg\!= f) \gg\!= k = x \gg\!= (\lambda a \to f\ a \gg\!= k) \\
\mathit{return} :: a \to \mathsf{T}\ a & \mathit{return}\ a \gg\!= k = k\ a \\
 & x \gg\!= \mathit{return} \quad = x
\end{array}
$$

with additionally $\mathit{mzero} :: \mathsf{T}\ a$.

Comprehensions can then be generalized to other monads:

$$
\begin{array}{ll}
\mathcal{D}\ [\, e \mid \,] & = \mathit{return}\ e \\
\mathcal{D}\ [\, e \mid p \leftarrow e', Q \,] & = e' \gg\!= \lambda p \to \mathcal{D}\ [\, e \mid Q \,] \\
\mathcal{D}\ [\, e \mid e', Q \,] & = \mathit{guard}\ e' \gg \mathcal{D}\ [\, e \mid Q \,] \\
\mathcal{D}\ [\, e \mid \mathbf{let}\ d, Q \,] & = \mathbf{let}\ d\ \mathbf{in}\ \mathcal{D}\ [\, e \mid Q \,]
\end{array}
$$

(where $\mathit{guard}\ b = \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ ()\ \mathbf{else}\ \mathit{mzero}$).

Hence monad comprehensions for sets, bags, maps-to-monad-zeroes, etc.

5. The problem with joins

The comprehension yields a terrible query plan!
Constructs entire cartesian product, then discards most of it:

$$
\begin{aligned}
&\mathit{cp}\ \mathit{customers}\ \mathit{invoices} && \triangleright \\
&\mathit{filter}\ (\lambda(c, i) \to c.\mathit{cid} == i.\mathit{customer}) && \triangleright \\
&\mathit{filter}\ (\lambda(c, i) \to i.\mathit{due} < \mathit{today}) && \triangleright \\
&\mathit{fmap}\ (\lambda(c, i) \to (c.\mathit{name}, c.\mathit{address}, i.\mathit{amount})
\end{aligned}
$$

(where $\triangleright$ is reverse function application).
Better to group by customer identifier, then handle groups separately:

$$
\begin{aligned}
&(\mathit{indexBy}\ \mathit{cid}\ \mathit{customers})\ \text{`}\mathit{merge}\text{`}\ (\mathit{indexBy}\ \mathit{customer}\ \mathit{invoices}) && \triangleright \\
&\mathit{fmap}\ (\mathit{id} \times \mathit{filter}\ (\lambda i \to i.\mathit{due} < \mathit{today})) && \triangleright \\
&\mathit{fmap}\ (\mathit{fmap}\ (\lambda c \to (c.\mathit{name}, c.\mathit{address})) \times \mathit{fmap}\ (\lambda i \to i.\mathit{amount}))
\end{aligned}
$$

(where *indexBy* partitions, and *merge* pairs on common index).
But this doesn't correspond to anything expressible in comprehensions.

6. Comprehensive comprehensions

Various extensions to the comprehension syntax:

- parallel ('zip') comprehensions (since GHC 5.0, 2001):

$$[\,(x, y) \mid x \leftarrow [\,1, 2, 3\,] \mid y \leftarrow [\,4, 5, 6\,]\,]$$

- 'order by' and 'group by' (Wadler & Peyton Jones, 2007):

$$\begin{aligned}
&[\ (\mathit{the}\ \mathit{dept}, \mathit{sum}\ \mathit{salary}) \\
&\mid (\mathit{name}, \mathit{dept}, \mathit{salary}) \leftarrow \mathit{employees} \\
&,\ \textbf{then group by}\ \mathit{dept}\ \textbf{using}\ \mathit{groupWith} \\
&,\ \textbf{then}\ \mathit{sortWith}\ \textbf{by}\ \mathit{sum}\ \mathit{salary} \\
&,\ \textbf{then}\ \mathit{take}\ 5\ ]
\end{aligned}$$

(NB **group by** rebinds the variables bound earlier!)

Initially just for lists, but…

Generalized comprehensive comprehensions

. . . generalizes nicely to other monads (Giorgidze et al, 2011):

$$\mathcal{D}\,[\,e \mid (Q \mid R), S\,]$$
$$= mzip\ (\mathcal{D}\,[\,\nu Q \mid Q\,])\ (\mathcal{D}\,[\,\nu R \mid R\,]) \gg\!= \lambda(\nu Q, \nu R) \to \mathcal{D}\,[\,e \mid S\,]$$

$$\mathcal{D}\,[\,e \mid Q, \textbf{then}\ f\ \textbf{by}\ b, R\,]$$
$$= f\ (\lambda \nu Q \to b)\ (\mathcal{D}\,[\,\nu Q \mid Q\,]) \gg\!= \lambda \nu Q \to \mathcal{D}\,[\,e \mid R\,]$$

$$\mathcal{D}\,[\,e \mid Q, \textbf{then group by}\ b\ \textbf{using}\ f, R\,]$$
$$= f\ (\lambda \nu Q \to b)\ (\mathcal{D}\,[\,\nu Q \mid Q\,]) \gg\!= \lambda ys \to$$
$$\textbf{case}\ (fmap\ \nu Q_1\ ys, \ldots, fmap\ \nu Q_n\ ys)\ \textbf{of}\ \nu Q \to \mathcal{D}\,[\,e \mid R\,]$$

where νQ is the tuple of variables bound by Q (and used subsequently), and νQ_i is a selector mapping νQ to its ith component.

7. Solving the problem with (equi-)joins

Maps-to-bags form a monad-with-zero—roughly:

```
type Map k v   = k → v
type Table k v = Map k (Bag v)
```

Now define

```
indexBy :: Eq k ⇒ (v → k) → Bag v → Table k v
indexBy f xs k = filter (λv → f v == k) xs

merge   :: Table k v → Table k w → Table k (v, w)
merge f g = λk → cp (f k) (g k)
```

Can use *merge* for parallel comprehensions:

```
instance MonadZip (Table k) where mzip = merge
```

and *indexBy* for grouping.

Given input tables

$$customers :: Bag\ (CID, Name, Address)$$
$$invoices :: Bag\ (IID, CID, Amount, Date)$$

evaluate our example query as:

$$\begin{aligned}
&query :: Map\ Int\ (Name, Address, Bag\ Amount)\\
&query = [\ (the\ name, the\ addr, amount)\\
&\qquad\quad | (cid, name, addr) \leftarrow customers\\
&\qquad\quad , \textbf{then group by}\ cid\ \textbf{using}\ indexBy\\
&\qquad\quad | (iid, customer, amount, due) \leftarrow invoices\\
&\qquad\quad , due < today\\
&\qquad\quad , \textbf{then group by}\ customer\ \textbf{using}\ indexBy\ ]
\end{aligned}$$

Avoids expanding the whole cartesian product.

8. Aggregation

For database queries, want to aggregate collections: *count*, *sum*, *some*, ...

Problem: maps may be infinite.

Solution: restrict to *finite* maps.

Problem: not a monad—$return\ a = \lambda k \to a$ yields a non-finite map.

Solution? *semi-monads* (with bind but no return).

Problem: semi-monad comprehensions—base case uses *return*:

$$\mathcal{D}\ [\ e\ |\ ]\ =\ return\ e$$

This is surmountable... but we prefer:

Solution: *graded* (indexed, parametric) monads

9. Graded monads

Monad $(\mathsf{T}, \gg\!=, \mathit{return})$ has endofunctor $\mathsf{T} : \mathbf{C} \to \mathbf{C}$, polymorphic functions

$$
\begin{aligned}
(\gg\!=) \quad &:: \mathsf{T}\ a \to (a \to \mathsf{T}\ b) \to \mathsf{T}\ b \\
\mathit{return} &:: a \to \mathsf{T}\ a
\end{aligned}
$$

such that

$$
\begin{aligned}
(x \gg\!= f) \gg\!= k &= x \gg\!= (\lambda a \to f\ a \gg\!= k) \\
\mathit{return}\ a \gg\!= k &= k\ a \\
x \gg\!= \mathit{return} \quad &= x
\end{aligned}
$$

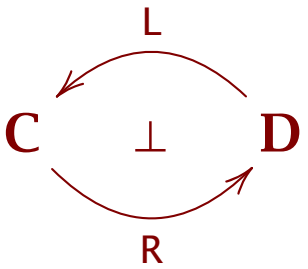
Katsumata's *M-graded monad* $(\mathsf{T}, \gg\!=, \mathit{return})$ for monoid $(M, \cdot, \varepsilon)$ has (non-endo-)functor $\mathsf{T} : M \to [\mathbf{C}, \mathbf{C}]$ and

$$
\begin{aligned}
(\gg\!=) \quad &:: \mathsf{T}\ m\ a \to (a \to \mathsf{T}\ n\ b) \to \mathsf{T}\ (m \cdot n)\ b \\
\mathit{return} &:: a \to \mathsf{T}\ \varepsilon\ a
\end{aligned}
$$

with same laws. We use $\mathsf{T} = \mathit{Table}$ over monoid $(K, \times, 1)$ of *finite* key types.

10. Adjunctions, and query optimization

Optimizations depend on a body of *meaning-preserving transformations*, all arising from algebraic properties of the datatypes—*adjunctions*:



with $\lfloor \cdot \rfloor : \mathbf{C}(\mathsf{L}\ X, Y) \simeq \mathbf{D}(X, \mathsf{R}\ Y) : \lceil \cdot \rceil$

Currying yields indexing; *products* yield projection and merge; *coproducts* yield filters; *free* commutative monoids yield selection and aggregation.

Monads famously arise from adjunctions; graded monads do too, albeit in a slightly more complicated way.

Work in progress: justifying standard query optimizations via these correspondences.

11. Comprehending semi-monads

Prohibit comprehensions with no qualifiers; multiple base cases instead.

$$
\begin{aligned}
\mathcal{D}\,[\,\varepsilon \mid p \leftarrow e'\,] &= \mathit{fmap}\ (\lambda p \to e')\ \varepsilon \\
\mathcal{D}\,[\,\varepsilon \mid e'\,] &= \ldots \quad \text{-- not allowed} \\
\mathcal{D}\,[\,\varepsilon \mid \mathbf{let}\ d\,] &= \ldots \quad \text{-- not allowed} \\
\mathcal{D}\,[\,\varepsilon \mid (Q \mid R)\,] &= \mathit{fmap}\ (\lambda(\nu Q, \nu R) \to \varepsilon) \\
&\quad\ (\mathit{mzip}\ (\mathcal{D}\,[\,\nu Q \mid Q\,])\ (\mathcal{D}\,[\,\nu R \mid R\,])) \\
\mathcal{D}\,[\,\varepsilon \mid Q, \mathbf{then}\ f\ \mathbf{by}\ b\,] &= \mathit{fmap}\ (\lambda \nu Q \to \varepsilon)\ (f\ (\lambda \nu Q \to b)\ (\mathcal{D}\,[\,\nu Q \mid Q\,])) \\
\mathcal{D}\,[\,\varepsilon \mid Q, \mathbf{then\ group\ by}\ b\ \mathbf{using}\ f\,] & \\
\quad = \mathit{fmap}\ (\lambda \mathit{ys} \to \mathbf{case}\ (\mathit{fmap}\ \nu Q_1\ \mathit{ys}, \ldots, \mathit{fmap}\ \nu Q_n\ \mathit{ys})\ \mathbf{of}\ \nu Q \to \varepsilon) & \\
\quad\ (f\ (\lambda \nu Q \to b)\ (\mathcal{D}\,[\,\nu Q \mid Q\,])) &
\end{aligned}
$$

Also, we can't define *guard* if we don't have *return*, so desugaring of guards needs to change:

$$
\mathcal{D}\,[\,\varepsilon \mid e', Q\,] = \mathbf{if}\ e'\ \mathbf{then}\ \mathcal{D}\,[\,\varepsilon \mid Q\,]\ \mathbf{else}\ \mathit{mzero}
$$