Towards an Effective Calculus for Object Query Languages

Leonidas Fegaras David Maier

Oregon Graduate Institute

The Gap between Theory & Practice

Most commercial relational query languages go beyond the formal model in some respects:

- aggregate operators,
- sort orders,
- grouping,
- update capabilities.

New Requirements

New database languages must be able to handle:

- type extensibility;
- multiple collection types (e.g., sets, lists, trees, arrays);
- arbitrary nesting of type constructors;
- large objects (e.g., text, sound, image);
- temporal data;
- methods.

Why do we Need a Formal Calculus?

- facilitates equational reasoning;
- provides a theory for proving query transformations correct;
- imposes language uniformity;
- avoids language inconsistencies.

functional languages	$\longleftrightarrow$	lambda calculus
relational databases	$\longleftrightarrow$	relational algebra/calculus
object-oriented databases	$\longleftrightarrow$	?

What is an Effective Calculus?

Several aspects:

- coverage,
- ease of manipulation,
- ease of evaluation,
- uniformity.

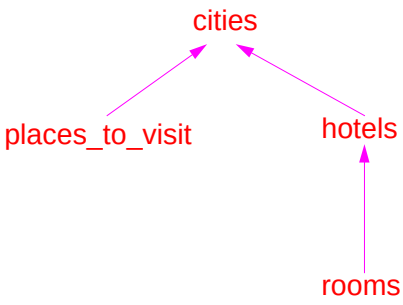
Rest of the Talk

- Monoids;
- *Algebra:* monoid homomorphisms;
- *Calculus:* monoid comprehensions;
- unnesting comprehensions.

Case Study: ODMG-93 OQL

class city = < name: *string*,
hotels: *set*(hotel),
places_to_visit: *list*(< name: *string*, address: *string* >) >
extent cities;

class hotel = < name: *string*,
rooms: *bag*(< bed#: *int*, price: *int* >) >;




```
select distinct h.name
  from   c in cities,
         h in c.hotels,
         p in c.places_to_visit
 where   c.name="Portland"
   and   h.name=p.name
```


OQL:

```
select distinct h.name
  from   c in cities,
         h in c.hotels,
         p in c.places_to_visit
 where   c.name="Portland"
   and   h.name=p.name
```

Monoid comprehension:

$$set\{\ h.name \mid c \leftarrow cities,\ h \leftarrow c.hotels,\ p \leftarrow c.places\_to\_visit,\ c.name = \text{“Portland”},\ h.name = p.name\ \}$$

Monoids

A *monoid* is an algebraic structure that captures most collection and aggregate types:

operator	*functionality*	e.g., sets
zero	the identity value	{ }
merge(x,y)	associative with identity **zero**	x ∪ y
unit(a)	singleton construction	{ a }

{ 1, 2, 3 } = { 1 } ∪ { 2 } ∪ { 3 }

Some Monoids

Collection Monoids

monoid	type	zero	unit(a)	merge
list	list(α)	[ ]	[a]	append
set	set(α)	{ }	{a}	∪
bag	bag(α)	{{ }}	{{a}}	⊎

Primitive Monoids

monoid	type	zero	unit(a)	merge
sum	integer	0	a	+
some	boolean	false	a	∨
all	boolean	true	a	∧

Monoid Homomorphisms

hom[T,S](f) is a *homomorphism* from a collection monoid T to any monoid S:

hom[T,S](f) (**zero**[T])	=	**zero**[S]
hom[T,S](f) (**unit**[T](a))	=	f(a)
hom[T,S](f) (**merge**[T](x,y))	=	**merge**[S](**hom**[T,S](f) (x), **hom**[T,S](f) (y))

Example

Notation: $\lambda a \,.\, \{ a^2 \}$ is the function f such that: $f(a) = \{ a^2 \}$

hom[*bag,set*]($\lambda a \,.\, \{ a^2 \}$) $\{\!\{1, 2, 3\}\!\}$

= **hom**[*bag,set*](f) ($\{\!\{1\}\!\} \uplus \{\!\{2\}\!\} \uplus \{\!\{3\}\!\}$)

= $f(1) \cup f(2) \cup f(3)$

= $\{1\} \cup \{4\} \cup \{9\}$

= $\{1, 4, 9\}$

Monoid Comprehensions

A *monoid comprehension* takes the form:

$$T\{ e \mid r_1, \ldots, r_n \}$$

head *qualifiers*

where T is a monoid and each qualifier r_i is either:

- a *generator* $v \leftarrow u$;
- a *filter* pred.

Examples

$set\{ (a,b) \mid a \leftarrow x, b \leftarrow y \}$ =

```
res = { };
for each a in x do
    for each b in y do
        res = res ∪ { (a,b) };
return res;
```

$set\{ (a,b) \mid a \leftarrow [1,2,3], b \leftarrow \{\!\{4,5\}\!\} \}$ = { (1,4), (1,5), (2,4), (2,5), (3,4), (3,5) }

$sum\{ a \mid a \leftarrow [1,2,3], a \geq 2 \}$ = 2+3 = 5

Formal Definition of a Monoid Comprehension

T{ e | }	=	**unit**[T](e)
T{ e | v←u, r_1, ..., r_n }	=	**hom**[S,T](λv. T{ e | r_1, ..., r_n })(u) *where* S *is the type of* u
T{ e | pred, r_1, ..., r_n }	=	**if** pred **then** T{ e | r_1, ..., r_n } **else zero**[T]

set{ (a,b) \| a←[1,2,3], b←{{4,5}} }

= **hom**[*list,set*](λa. **hom**[*bag,set*](λb. { (a,b) })
({{4,5}}))
([1,2,3])

Other Examples

filter(pred) e	=	*set*{ x | x ← e, pred(x) }
$e_1 \cap e_2$	=	*set*{ x | x ← e_1, x ∈ e_2 }
length(e)	=	*sum*{ 1 | x ← e }
∃a ∈ e: pred	=	*some*{ pred | a ← e }
∀a ∈ e: pred	=	*all*{ pred | a ← e }
nest(k) e	=	*set*{ < KEY= k(x), DATA= *set*{ y | y ← e, k(x)=k(y) } > | x ← e }
unnest(e)	=	*set*{ x | s ← e, x ← s.DATA }

Translating OQL

```
select h.name
 from hl in ( select c.hotels
                from c in cities
               where c.name="Portland" ),
      h in hl
 where exists r in h.rooms: ( r.bed#=3 )
```


bag{ h.name | hl ← *bag*{ c.hotels | c ← cities, c.name="Portland" },
h ← hl,
some{ r.bed#=3 | r ← h.rooms } }

Program Normalization

Canonical form: (a path is a cascade of projections: $X.A_1.A_2 \ldots A_m$)

- T{ e | $x_1 \leftarrow path_1, \ldots, x_n \leftarrow path_n$, pred }

Examples of normalization rules:

- T{ e |  , x ← S{ u |  },  }
  → T{ e |  ,  , x ≡ u,  }

- T{ e |  , some{ pred |  },  }
  → T{ e |  ,  , pred,  }

Example

bag{ h.name | hl ← *bag*{ c.hotels | c ← cities, c.name="Portland" },
h ← hl,
some{ r.bed#=3 | r ← h.rooms } }

= *bag*{ h.name | c ← cities, c.name="Portland",
hl ≡ c.hotels,
h ← hl,
some{ r.bed#=3 | r ← h.rooms } }

Substitute c.hotels for hl

= *bag*{ h.name | c ← cities, c.name="Portland",
h ← c.hotels,
some{ r.bed#=3 | r ← h.rooms } }

= *bag*{ h.name | c ← cities,
h ← c.hotels,
r ← h.rooms,
(c.name="Portland") ∧ (r.bed#=3) }

Unnesting OQL Queries

```
select h.name
 from hl in ( select c.hotels
              from c in cities
              where c.name="Portland" ),
      h in hl
 where exists r in h.rooms: ( r.bed#=3 )
```

normalization

```
select h.name
 from c in cities,
      h in c.hotels,
      r in h.rooms
 where c.name="Portland" and r.bed#=3
```

Related Work

- *monoid homomorphisms* [V. Tannen et al];
  - SRU: **sr_comb**$_T$(u,f,e)
  - EXT: **ext**$_T$(f) = **hom**[T,T](f)
- *boom hierarchy of types* [R. Bird, L. Meertens, R. Backhouse];
- *monad comprehensions* [P. Wadler, P. Trinder, P. Buneman et al];
- *normalization* [L. Wong].

Effectiveness of the Calculus

- *coverage:* it captures many new database language features:
  - supports expression nesting;
  - allows arbitrary nesting of type constructors;
  - supports multiple collection types;
  - handles aggregates & predicates directly.
- *ease of manipulation:* nested comprehensions can be unnested by a simple normalization algorithm.

Future Work

Model extensions:

- vectors & arrays;
- object identity & mutable objects;
- updates;
- methods.

An algebraic model for query translation:

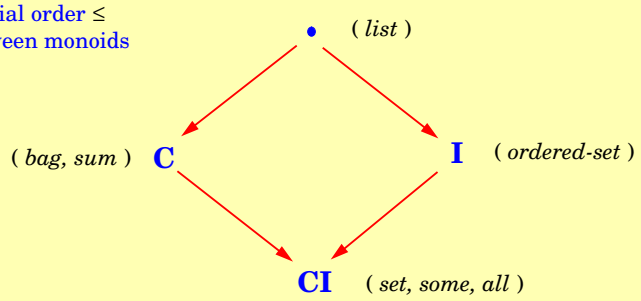
- facilitates data independence;
- captures many OO physical designs.

Implementation:

- a translator from OQL to the monoid calculus;
- a query optimizer.

Restriction

Partial order ≤ between monoids



C: *Commutative*
I: *Idempotent*

hom[T,S] is *well-formed* if and only if T≤S

bag-cardinality(x) = **hom**[*bag*,*sum*](λa.1)(x)
is well-formed, since *bag* ≤ *sum*

set-cardinality(x) = **hom**[*set*,*sum*](λa.1)(x)
is **not** well-formed, since *set* > *sum*