An Effective Framework for Processing Object-Oriented Database Languages

Leonidas Fegaras
Oregon Graduate Institute
(Ph.D. University of Massachusetts)

Maintaining Persistent Data

Data Base Management System:

Key issue: data independence.
The Gap Between Theory & Practice

Most commercial relational query languages are based on

- tuple calculus (SQL, Quel)
- domain calculus (QBE)

However, in some respects they go beyond the formal model:

- aggregate operators,
- sort orders,
- grouping,
- update capabilities.
**Insufficient Modeling Power**

Relational databases cannot effectively model many new applications:

- multimedia & World-Wide-Web,
- scientific databases,
- CAD,
- CASE,
- GIS,
- office automation.

---

**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 & spatial data;
- unstructured data;
- active rules;
- methods.
New Proposals for Database Languages

Relational extensions:

- UniSQL,
- POSTGRES/Illustra,
- SQL3.

Object-oriented databases:

- Gemstone,
- O2,
- OQL of ODMG-93.

Why Do We Need a Formal Calculus?

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

<table>
<thead>
<tr>
<th>functional languages</th>
<th>λ-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>relational databases</td>
<td>relational algebra/calculus</td>
</tr>
<tr>
<td>object-oriented databases</td>
<td>?</td>
</tr>
</tbody>
</table>
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;
- overview of the query translation framework;
- physical design specification;
- one example of query translation.
Case Study: ODMG-93 OQL

**Class**

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

- **hotel** = < name: string, 
  rooms: set (< bed#: int, price: int >) >
  
**OQL:**

```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:**

```oql
set { h.name | c ← cities, 
  h ← c.hotels, 
  p ← c.places_to_visit, 
  c.name="Portland", 
  h.name=p.name }
```
Monoids

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

<table>
<thead>
<tr>
<th>operator</th>
<th>functionality</th>
<th>e.g., sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>zero</strong></td>
<td>the identity value</td>
<td>{ }</td>
</tr>
<tr>
<td><strong>merge</strong>(x,y)</td>
<td>associative with identity <strong>zero</strong></td>
<td>x ∪ y</td>
</tr>
<tr>
<td><strong>unit</strong>(a)</td>
<td>singleton construction</td>
<td>{ a }</td>
</tr>
</tbody>
</table>

\[
\{ 1, 2, 3 \} = \{ 1 \} \cup \{ 2 \} \cup \{ 3 \}
\]

Optional properties:
- **commutativity**: \(\text{merge}(x,y) = \text{merge}(y,x)\)
- **idempotence**: \(\text{merge}(x,x) = x\)

Some Monoids

<table>
<thead>
<tr>
<th>Collection Monoids</th>
<th>Primitive Monoids</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>list</strong></td>
<td><strong>sum</strong></td>
</tr>
<tr>
<td><strong>set</strong></td>
<td><strong>prod</strong></td>
</tr>
<tr>
<td><strong>bag</strong></td>
<td><strong>some</strong></td>
</tr>
<tr>
<td><strong>sorted</strong>[f]</td>
<td><strong>all</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>monoid</th>
<th>type</th>
<th>zero</th>
<th>unit(a)</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>list</strong>(α)</td>
<td>list(α)</td>
<td>[ ]</td>
<td>[a]</td>
<td>append</td>
</tr>
<tr>
<td><strong>set</strong>(α)</td>
<td>set(α)</td>
<td>{ }</td>
<td>{a}</td>
<td>∪</td>
</tr>
<tr>
<td><strong>bag</strong>(α)</td>
<td>bag(α)</td>
<td>{ }</td>
<td>{a}</td>
<td>⊔</td>
</tr>
<tr>
<td><strong>sorted</strong><a href="%CE%B1">f</a></td>
<td>list(α)</td>
<td>[ ]</td>
<td>[a]</td>
<td>list_merge[f]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>monoid</th>
<th>type</th>
<th>zero</th>
<th>unit(a)</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sum</strong></td>
<td>integer</td>
<td>0</td>
<td>a</td>
<td>+</td>
</tr>
<tr>
<td><strong>prod</strong></td>
<td>integer</td>
<td>1</td>
<td>a</td>
<td>*</td>
</tr>
<tr>
<td><strong>some</strong></td>
<td>boolean</td>
<td>false</td>
<td>a</td>
<td>∨</td>
</tr>
<tr>
<td><strong>all</strong></td>
<td>boolean</td>
<td>true</td>
<td>a</td>
<td>∧</td>
</tr>
</tbody>
</table>
**Monoid Homomorphisms**

$\text{hom}_{[T,S]}(f)$ is a homomorphism from a collection monoid $T$ to any monoid $S$:

$$
\begin{align*}
\alpha & \quad f \quad \rightarrow \quad S \\
T(\alpha) & \quad \quad \quad \quad \quad \rightarrow \quad S \\
\text{hom}_{[T,S]}(f)(\text{zero}[T]) & = \text{zero}[S] \\
\text{hom}_{[T,S]}(f)(\text{unit}[T](a)) & = f(a) \\
\text{hom}_{[T,S]}(f)(\text{merge}[T](x,y)) & = \text{merge}[S](\text{hom}_{[T,S]}(f)(x), \text{hom}_{[T,S]}(f)(y))
\end{align*}
$$

**Example**

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

$$
\begin{align*}
\text{hom}_{[\text{bag},\text{set}]}(\lambda a.\{a^2\}) & \{1, 2, 3\} \\
& = \text{hom}_{[\text{bag},\text{set}]}(f) (\{1\} \cup \{2\} \cup \{3\}) \\
& = f(1) \cup f(2) \cup f(3) \\
& = \{1\} \cup \{4\} \cup \{9\} \\
& = \{1, 4, 9\}
\end{align*}
$$
Other Examples

\[ 5 \in x = \text{hom}[set,some](\lambda a. (a=5))(x) \]
\[ \{8\} \cup \{5\} = \{8,5\} \]
\[ 8=5 \quad 5=5 \]
\[ \text{false} \lor \text{true} = \text{true} \]

\[ \text{length}(x) = \text{hom}[list,sum](\lambda a. 1)(x) \]
\[ \text{append}([2],[3]) = [2,3] \]
\[ 1 \quad 1 \]
\[ 1 + 1 = 2 \]

Monoid Comprehensions

A monoid comprehension takes the form:

\[ T\{e | r_1, \ldots, r_n\} \]

where \(T\) is a monoid and each qualifier \(r_i\) is either:

- a generator \(v \leftarrow u\);
- a filter \(\text{pred}\).
Examples

\[
\text{set}\{ (a,b) \mid a \leftarrow x, b \leftarrow y \} = \begin{cases} \text{res} = \{ \}; \\
\text{for each } a \text{ in } x \text{ do} \\
\quad \text{for each } b \text{ in } y \text{ do} \\
\quad \quad \text{res} = \text{res} \cup \{ (a,b) \}; \\
\text{return res;}
\end{cases}
\]

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

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

Formal Definition of a Monoid Comprehension

\[
\begin{align*}
T\{ e \mid \} & = \text{unit}[T](e) \\
T\{ e \mid v \leftarrow u, r_1, ..., r_n \} & = \text{hom}[S,T](\lambda v. T\{ e \mid r_1, ..., r_n \})(u) \\
& \quad \text{where } S \text{ is the type of } u \\
T\{ e \mid \text{pred}, r_1, ..., r_n \} & = \text{if pred then } T\{ e \mid r_1, ..., r_n \} \text{ else zero}[T]
\end{align*}
\]

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

\[
= \text{hom}[\text{list,set}](\lambda a. \text{hom}[\text{bag,set}](\lambda b. \{ (a,b) \})(\{4,5\}))(\{1,2,3\})
\]
Other Examples

\[
\begin{align*}
\text{filter}(\text{pred})\ e &= \{ x | x \leftarrow e, \text{pred}(x) \} \\
\text{flatten}(e) &= \{ x | s \leftarrow e, x \leftarrow s \} \\
\text{e}_1 \cap \text{e}_2 &= \{ x | x \leftarrow \text{e}_1, x \in \text{e}_2 \} \\
\text{length}(e) &= \sum \{ 1 | x \leftarrow e \} \\
\exists a \in e: \text{pred} &= \{ \text{pred} | a \leftarrow e \} \\
\forall a \in e: \text{pred} &= \{ \text{pred} | a \leftarrow e \} \\
\text{nest}(k)\ e &= \{ \text{< KEY= k(x), DATA= \{ y | y \leftarrow e, k(x)=k(y) \} > | x \leftarrow e } \} \\
\text{unnest}(e) &= \{ x | s \leftarrow e, x \leftarrow s.DAT A \}
\end{align*}
\]

Translating OQL

\[
\begin{align*}
\text{select distinct h.name} \\
\text{from hl in} ( \text{select c.hotels} \\
\text{from c in} \text{cities} \\
\text{where c.name='Portland'}, \\
\text{h in hl} \\
\text{where exists r in} \text{h.rooms: ( r.bed#=3 )}
\end{align*}
\]

\[
\begin{align*}
\text{set}\{ h.name | hl \leftarrow \text{bag}\{ c.hotels | c \leftarrow \text{cities, c.name='Portland'} \}, \\
\text{h \leftarrow hl,} \\
\text{some}\{ r.bed#=3 | r \leftarrow \text{h.rooms } \} \}
\end{align*}
\]
Program Normalization

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

- $T\{ e | x_1 \leftarrow \text{path}_1, \ldots, x_n \leftarrow \text{path}_n, \text{pred} \}$

Examples of normalization rules:

- $T\{ e | S\{ u | \}, x \leftarrow S\{ u | \}, \} \rightarrow T\{ e | u, x \equiv u, \}$

- $T\{ e | \text{some} \{ \text{pred} | \}, \} \rightarrow T\{ e | \text{pred}, \}$

Example

$\text{set}\{ \text{h.name} | \text{hl} \leftarrow \text{bag}\{ \text{c.hotels} | \text{c} \leftarrow \text{cities}, \text{c.name=“Portland”} \}, \text{h} \leftarrow \text{hl}, \text{some}\{ \text{r.bed#=3} | \text{r} \leftarrow \text{h.rooms} \} \}$

$= \text{set}\{ \text{h.name} | \text{c} \leftarrow \text{cities}, \text{c.name=“Portland”}, \text{hl} \leftarrow \text{c.hotels}, \text{h} \leftarrow \text{hl}, \text{some}\{ \text{r.bed#=3} | \text{r} \leftarrow \text{h.rooms} \} \}$

$= \text{set}\{ \text{h.name} | \text{c} \leftarrow \text{cities}, \text{c.name=“Portland”}, \text{h} \leftarrow \text{c.hotels}, \text{some}\{ \text{r.bed#=3} | \text{r} \leftarrow \text{h.rooms} \} \}$

$= \text{set}\{ \text{h.name} | \text{c} \leftarrow \text{cities}, \text{h} \leftarrow \text{c.hotels}, \text{r} \leftarrow \text{h.rooms}, \text{c.name=“Portland”} \wedge ( \text{r.bed#=3} ) \}$
Unnesting OQL Queries

```
select distinct 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 )
```

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

Query Translation

Need to support data independence.

Tasks:

- the database administrator specifies the conceptual database schema;
- the database implementor specifies the physical design;
- an application programmer submits a query against the database;
- the query translator translates the query into a physical plan that reflects the physical design.
Conceptual-to-Internal Mapping

R(x) = set{ a | a ← x }

Plan transformers:

DB = C_i(DB)
QUERY_i(DB) = QUERY(C_i(DB))
**Example**

Conceptual Schema:

```
set(< A: int, B: set(< C: int, D: int >), E: int >)
```

Physical design:

- normalize the inner set;
- implement the outer set as a B-tree with key A;
- attach a secondary index to the outer set with a key E.
Abstraction function:

\[ R(DB) = \text{set}\{ \begin{array}{l} A = a.A, \\ B = \text{set}\{ C = b.C, D = b.D \mid b \leftarrow DB.T2, b.\#=@a \}, \\ E = a.E \} \\ | a \leftarrow DB.T1 \} \]

Plan Transformer: \( DB = C_1(DB) \) (reconstructs \( T1 \) from \( T3 \))

Conceptual query:

\[ \text{query(db)} = \text{sum}\{ y.C \mid x \leftarrow \text{db}, y \leftarrow x.B, x.A=10, y.D>5 \} \]

Physical plan:

\[
\begin{align*}
\text{QUERY(DB)} &= \text{query(R(DB))} \\
&= \text{sum}\{ y.C \mid x \leftarrow \text{R(DB)}, y \leftarrow x.B, x.A=10, y.D>5 \} \\
&= \ldots \\
&= \text{sum}\{ b.C \mid a \leftarrow DB.T1, b \leftarrow DB.T2, \\
&\quad b.\#=@a, a.A=10, b.D>5 \}
\end{align*}
\]

(A sort-merge join!)

Alternative plan:

\[
\begin{align*}
\text{QUERY}_1(DB) &= \text{QUERY}(C_1(DB)) \\
&= \ldots \\
&= \ldots \quad (a \text{ plan that uses the secondary index } T3)
\end{align*}
\]
Our Query Translation Framework is Purely Algebraic

It formalizes and generalizes many ad-hoc techniques used in relational query translation and optimization:

- the mapping from logical queries to physical plans is *meaning preserving*;
- the data translation overhead is completely eliminated by normalization;
- alternative physical plans are derived by a very simple cost-directed term-rewriting system that uses 3 types of rules:
  - associativity of monoids;
  - commutativity of some monoids;
  - plan transformers.

Physical Design Specification

A *physical design language* is provided that is

- declarative,
- extensible.

Captures many physical designs:

- object clustering;
- horizontal/vertical partitioning;
- schema normalization;
- join indices;
- multiple access paths via secondary indices.
Conclusion

I have presented:

- a uniform calculus that captures many new database language features:
  - supports expression nesting;
  - allows arbitrary nesting of type constructors;
  - supports multiple collection types;
  - handles aggregates & predicates directly.
- a normalization algorithm that unnests nested comprehensions;
- an effective algebraic model for query translation that facilitates data independence.

Current and Future Work

Model extensions:

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

Implementation:

- translators from OQL and a subset of SQL3 to the monoid calculus;
- a query optimizer.
**Restriction**

Partial order $\leq$ between monoids

$\mathbf{C}$: Commutative
$\mathbf{I}$: Idempotent

$\mathbf{CI}$: (set, some, all)

(ordered-set)

(ordered-set)

(list)

$\mathbf{hom}[T,S]$ is well-formed if and only if $T \leq S$

$\text{bag-cardinality}(x) = \mathbf{hom}([\text{bag,sum}](\lambda a.1))(x)$ is well-formed, since $\text{bag} \leq \text{sum}$

$\text{set-cardinality}(x) = \mathbf{hom}([\text{set,sum}](\lambda a.1))(x)$ is not well-formed, since $\text{set} > \text{sum}$

**Related Work**

- monoid homomorphisms [V. Tannen et al];

- SRU

  - $\text{ext}(f) A = \mathbf{hom}[T,T](f) A$

- boom hierarchy of types [R. Bird, L. Meertens, R. Backhouse];

- monad comprehensions [P. Wadler, P. Trinder, P. Buneman];

- normalization [L. Wong].
**In Terms of Category Theory**

The collection monoid \( T(\alpha) \) is a **free monoid** generated by \( \alpha \):

\[
\begin{array}{ccc}
\text{unit}[T] & \alpha & T(\alpha) \\
\downarrow & \downarrow & \downarrow \\
\text{Adjunction:} & f & \text{hom}[T,S](f) \\
\downarrow & \downarrow & \downarrow \\
\text{S} & \text{unit}[T] \text{ is a natural transformation: } \text{id} \rightarrow T. \\
\end{array}
\]

\( \text{hom}[T,S](f) \) is the **homomorphic extension** of \( f \).

---

**Monads vs. Monoids**

The Kleisli triple \( (T, [,], \text{ext}(f)) \), which represents a monad, is a monoid: the Kleisli composition \( f \circ g = \text{ext}(f) \circ g \) is associative with zero \([]\).