Algebraic Languages and Optimization Methods
for Functional Programs and Database Languages

Leonidas Fegaras
Assistant Professor, UTA

The Gap Between Theory & Practice

Most commercial relational query languages are based on the relational calculus. However in some respects they go beyond the formal model. They support:

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

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),
- methods.

A New Formal Model is Needed

A formal algebra:

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

What is an effective algebra?
Several aspects:

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

Functional Languages to the Rescue

Functional languages are *value-based* (no side-effects!).
Values are immutable; new values are constructed from old values.
Programs are organized into functions.
Easier to write, understand, and reason about programs.
Functional vs. imperative programs:

Pure functions can always be tested separately.
Many optimizations are not always valid in imperative languages:

```
x+y -> y+x
x*0 -> 0
```

e.g., if `f() = { a:=a+1; 5 }`, then `f()*0` is not equivalent to `0`

Modern Functional Languages

Most popular: SML and Haskell.
They are based on the *lambda calculus*.

They support:

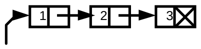
- strong static typing with type inference,
- automatic garbage collection,
- resilience to store corruption (no core dumps!),
- parametric polymorphism,
- higher-order functions,
- algebraic data types (no pointers!),
- pattern matching.

Some features are showing up in new imperative languages (e.g., Java).

Example from Haskell

`data list a = Nil | Cons a (list a)`

`Cons 1 (Cons 2 (Cons 3 Nil))`



Mapping a function `f` over the elements of a list:

```
map :: (a->b) -> list a -> list b
map f Nil        = Nil
map f (Cons a r) = Cons (f a) (map f r)
```

```
map(\a -> a+1) (Cons 1 (Cons 2 (Cons 3 Nil)))
   = Cons 2 (Cons 3 (Cons 4 Nil))
```

(where `\a->a+1` is the function `f` such that `f(a)=a+1`)

Loop Fusion and Deforestation

```
map f Nil        = Nil
map f (Cons a r) = Cons (f a) (map f r)
```

```
sum Nil        = 0
sum (Cons a r) = a+(sum r)
```

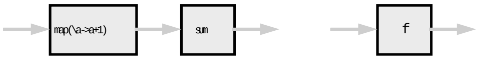
Suppose that we compose these operations:

```
f x  =  sum(map(\a -> a+1) x)
```

A better definition of `f`:

```
f Nil        = 0
f (Cons a r) = (a+1)+(f r)
```

How Can we Fuse Programs?



We can use standard program transformation techniques
... or we can use **folds.**

Folds
- can be defined for a large number of algebraic data types;
- support calculation-based program optimizations;
- support loop fusion and deforestation;
- facilitate equational reasoning and theorem proving.

So **What is a fold?**

The Fold Operator

A **fold** is the *natural* control structure for an algebraic data type.
It uses functional parameters to *abstract* over common inductive patterns.
It replaces data constructors with functions.

```
data list a = Nil | Cons a (list a)
```

```
fold c n Nil        = n
fold c n (Cons a r) = c a (fold c n r)
```

For example, if `x = Cons 1 (Cons 2 (Cons 3 Nil))`
then:

```
   fold c n x  =  c 1 (c 2 (c 3 n))
```

Examples

```
fold c n Nil        = n
fold c n (Cons a r) = c a (fold c n r)
```

```
sum Nil        = 0
sum (Cons a r) = a+(sum r)
```

```
➡     sum x  =  fold (\a r -> a+r) 0 x
       sum (Cons 1 (Cons 2 (Cons 3 Nil))) =  1+2+3+0 = 6
```

```
map f Nil        = Nil
map f (Cons a r) = Cons (f a) (map f r)
```

```
➡     map f x  =  fold (\a s -> Cons (f a) s) Nil x
```

Other Fold Operators

```
data tree a  =  Leaf a | Node (tree a) a (tree a)

fold m n (Leaf  a)     = m
fold m n (Node x a y) = n (fold m n x) a (fold m n y)

flatten x = fold (\l a r -> append l (Cons a r)) Nil
```

Fusion Laws

For lists:

$$\frac{\begin{array}{r} n' \;=\; g(n) \\ c'\;a\;(g\;r) \;=\; g(c\;a\;r) \end{array}}{g(\;\text{fold}\;c\;n\;x\;) \;=\; \text{fold}\;c'\;n'\;x}$$

For trees:

$$\frac{\begin{array}{r} m'\;a \;=\; g(m\;a) \\ n'\;(g\;l)\;a\;(g\;r) \;=\; g(n\;l\;a\;r) \end{array}}{g(\;\text{fold}\;m\;n\;x\;) \;=\; \text{fold}\;m'\;n'\;x}$$

Fusion Example

```
map f x  =  fold (\a s -> Cons (f a) s) Nil x
         =  fold c n x
sum x  =  fold (\a s -> a+s) 0 x
```

$$\boxed{\frac{n' = \text{sum } n \qquad c'\ a\ (\text{sum } r) = \text{sum}(c\ a\ r)}{\text{sum}(\ \text{fold } c\ n\ x\ ) = \text{fold } c'\ n'\ x}}$$

```
n' = sum n = sum Nil = 0
c' a (sum r) = sum(c a r)
             = sum(Cons (f a) r)
             = (f a)+(sum r)
⇒ c' a s = (f a)+s
⇒ sum(map f x) = fold (\a s -> (f a)+s) 0 x
```

Folds as a Basis for a Query Algebra

Folds have been used as a query algebra for an object-oriented database:

- relational database operations can be expressed as folds:

```
join f p x y =
  fold (\a r -> fold(\b s -> if (p a b) then
             Cons (f a b) r else r) y r) x Nil
```

- query plans can be expressed as folds;
- the fusion algorithm generalizes many algebraic query optimization techniques (such as unnesting queries);
- fusion can be used for eliminating the object translation overhead when translating queries into plans.

Problems: set commutativity and idempotence:

```
x ∪ y = y ∪ x
x ∪ x = x
```

Conclusion

Value-based, higher-order, operations:

- provide a uniform way of expressing database queries;
- have sufficient expressive power to capture modern database languages;
- satisfy simple laws that facilitate the proof of program correctness;
- support algebraic optimization methods.

Current Work

My current research work includes:

- building a query optimizer for ODMG'93 OQL using the monoid comprehension calculus. Using a physical design language to map conceptual queries into physical plans. (Work with Dave Maier at OGI; currently supported by NSF).
- extending OODB languages with temporal features (work with Ramez Elmasri at UTA).
- making functional languages more efficient by using program transformation techniques (work with Tim Sheard at OGI).