

Language-integrated query: state of the art and open problems

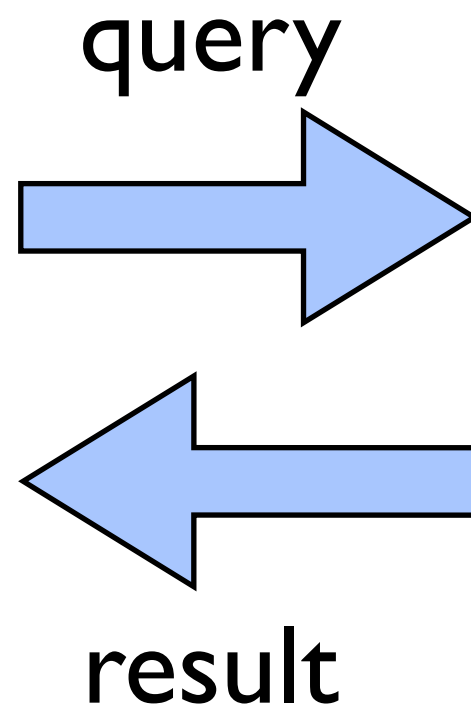
James Cheney
University of Edinburgh

Shonan Seminar on Language Integrated Queries
May 29, 2017

What problem are we trying to solve?



PL



DB

Why integrate language and query?

- Avoid "impedance mismatch" [Copeland & Maier 1985], queries that fail at run time?
 - or just because you forgot a space / closing paren?
- Avoid security vulnerabilities (SQL injection + friends)?
- Be able to optimize program and queries together / optimize across queries?
- Others?
 - My favorite: To be able to use a general purpose language to build complex, dynamic queries automatically (and safely)

Three strategies

(not mutually exclusive)

- using query operator APIs / ASTs
 - (e.g. .NET Linq operator, HaskellDB)
- directly embedding other query languages
 - (e.g. ...)
- Overcoming compression
 - (e.g. ...)
- I will talk about them, not because they are perfect but because they exemplify the three patterns above

(Very) far from an exhaustive list!

(No offense intended if I didn't list your system :)

LINQ

- Core: API and libraries handling a rich set of query operators (`Select`, `Where`, `GroupBy`, `Join`, `OrderBy`, etc.)

- `Expression<T>`, `IQueryable<T>` interfaces

```
employees.Where(x => x.Salary >= 50000)
           .Select(x => new { x.Name })
```

- "`x => e`" is a lambda-abstraction (implicitly quoted when appropriate)
- Expressions can be evaluated immediately (in-memory)
 - Or represented symbolically and analyzed/evaluated via remote queries
- Type-safety inherited from source language
 - Type providers (run-time type information in IDE) make this especially handy

LINQ in C#

- Query expressions can be written using special syntax (not SQL but similar):

```
from x in employees
where x.salary > 50000
select x.name
```
- Internally, this is treated as a *quoted* expression (using the `Expression<T>` API)
- The LINQ libraries reflect on and translate the expression to SQL (or potentially other targets)

LINQ in F#

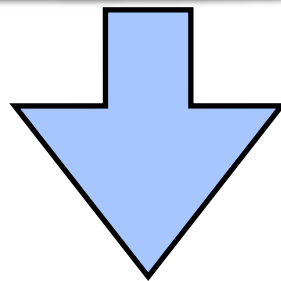
- Based on comprehension syntax (a.k.a. "do" notation, computation expressions, etc.)

```
query { for x in employees
        where (x.salary > 50000)
        yield {name=x.name} }
```

- As in C#, "query" is implicitly quoted and subject to rewriting / reflection
 - before being passed on to C# LINQ library...

LINQ (F#) example

```
query { for e in employees
        where (e.salary > 50000)
        yield {name=e.name} }
```



employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

```
select name
from employees e
where e.salary > 50000
```

name
Bert
Drew
Erik
Fred
Gina

Nested Relational Calculus

- SQL, C# queries, and F# queries are all based on a common foundation
- *Nested Relational Calculus*
[Buneman et al. 1995] - monadic core language for queries

Types	$A, B ::= O \mid \langle \vec{\ell} : \vec{A} \rangle \mid \text{Bag } A \mid A \rightarrow B$
Base types	$O ::= \text{Int} \mid \text{Bool} \mid \text{String}$
Terms	$M, N ::= x \mid c(\vec{M}) \mid \text{table } t \mid \text{if } M \text{ then } N \text{ else } N'$ $\mid \lambda x. M \mid M \ N \mid \langle \vec{\ell} = \vec{M} \rangle \mid M.\ell$ $\mid \text{return } M \mid \emptyset \mid M \uplus N \mid \text{for } (x \leftarrow M) \ N$ $\mid \text{empty } M$

Key question

- NRC allows *nesting*
 - queries can build sets of ... sets of sets
- Normal relational query languages (SQL + friends) only have flat relations
- Is NRC more expressive than SQL?
 - duh, yes, we can consume or return nested sets.
- More interesting question: is NRC more expressive for transforming flat inputs to flat outputs?

Surprising answer

- The Flat-Flat Theorem
 - Paredaens, van Gucht 1992
 - Showed that nested relational queries over flat inputs/outputs are no more expressive
- Conservativity Theorem
 - Wong [PODS 1994, JCSS 1996]
 - Showed a more general result:
 - queries with input/output of nesting depth n do not need to build intermediate structures of greater nesting depth

Conservativity and normalization

1. $(\lambda x. e) e' \rightarrow e[e'/x]$
2. $\pi_i(e_1, e_2) \rightarrow e_i$
3. *if true then* e_1 *else* $e_2 \rightarrow e_1$
4. *if false then* e_1 *else* $e_2 \rightarrow e_2$
5. *if (if* e_1 *then* e_2 *else* e_3) *then* e_4 *else* $e_5 \rightarrow$ *if* e_1 *then* *if* e_2 *then* e_4 *else* e_5 *else (if* e_3 *then* e_4 *else* e_5)
6. $\pi_i(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow \text{if } e_1 \text{ then } \pi_i e_2 \text{ else } \pi_i e_3$
7. $\{e \mid \Delta_1, x \in \{\}, \Delta_2\} \rightarrow \{\}$
8. $\{e \mid \Delta_1, x \in \{e'\}, \Delta_2\} \rightarrow \{e[e'/x] \mid \Delta_1, \Delta_2[e'/x]\}$
9. $\{e \mid \Delta_1, x \in e_1 \cup e_2, \Delta_2\} \rightarrow \{e \mid \Delta_1, x \in e_1, \Delta_2\} \cup \{e \mid \Delta_1, x \in e_2, \Delta_2\}$
10. $\{e \mid \Delta_1, x \in \{e' \mid \Delta'\}, \Delta_2\} \rightarrow \{e[e'/x] \mid \Delta_1, \Delta', \Delta_2[e'/x]\}$
11. $\{e \mid \Delta_1, x \in \text{if } e_1 \text{ then } e_2 \text{ else } e_3, \Delta_2\} \rightarrow \{e \mid \Delta_1, u \in \text{if } e_1 \text{ then } \{(\)\} \text{ else } \{\}, x \in e_2, \Delta_2\} \cup \{e \mid \Delta_1, u \in \text{if } e_1 \text{ then } \{\} \text{ else } \{(\)\}, e_3, \Delta_2\}$, provided (1) u is fresh, (2) e_2 is not $\{(\)\}$ and e_3 is not $\{\}$, and (3) e_2 is not $\{\}$ and e_3 is not $\{(\)\}$.

- Wong gave a straightforward *normalization* algorithm
- and an extension to handle (nonrecursive) sum types

Normal forms


- look roughly like this:

Query terms	$L ::= \biguplus \vec{C}$
Comprehensions	$C ::= \text{for } (\vec{G} \text{ where } X) \text{ return } M$
Generators	$G ::= x \leftarrow t$
Normalised terms	$M, N ::= X \mid R$
Record terms	$R ::= \langle \overrightarrow{\ell = \vec{M}} \rangle$
Base terms	$X ::= x.\ell \mid c(\vec{X}) \mid \text{empty } L$

- and can be translated to SQL:

Queries	$L, M, N ::= (\text{union all}) \overline{C}$
Comprehensions	$C ::= \text{select } R \text{ from } \overline{G} \text{ where } X$
Generators	$G ::= t \text{ as } x$
Record terms	$R ::= \overline{X \text{ as } \ell}$
Base terms	$X, Y, Z ::= x.\ell \mid c(\overline{X})$ $\mid \text{case when } X \text{ then } Y \text{ else } Z \text{ end}$

Dynamic/composable queries in F#?

 stackoverflow Questions Tags Users Badges Unanswered

How do you compose query expressions in F#?

I've been looking at query expressions here <http://msdn.microsoft.com/en-us/library/vstudio/hh225374.aspx>

And I've been wondering why the following is legitimate

```
let testQuery = query {  
    for number in netflix.Titles do  
    where (number.Name.Contains("Test"))  
}
```


But you can't really do something like this

```
let christmasPredicate = fun (x:Catalog.ServiceTypes.Title) -> x.Name.Contains("Christmas")  
let testQuery = query {  
    for number in netflix.Titles do  
    where christmasPredicate  
}
```

Surely F# allows composability like this so you can reuse a predicate?? What if I wanted Christmas titles combined with another predicate like before a specific date? I have to copy and paste my entire query? C# is completely unlike this and has several ways to build and combine predicates

f# computation-expression query-expressions

share | edit | flag

edited Dec 11 '12 at 19:38
 Ramon Snir
4,841 ● 2 ● 16 ● 39

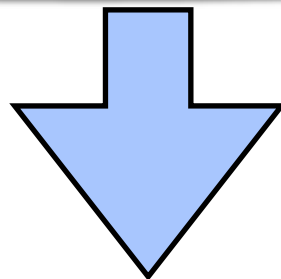
asked Dec 11 '12 at 19:02
 brian
452 ● 2 ● 17

Queries with function "parameters"?

- A way to (de)compose queries into reusable chunks?
 - (avoid repeating yourself)
- This could be very useful
 - a form of staged computation/meta-programming
- Queries could be constructed dynamically
 - including constructing queries of different "shape"
 - goes beyond simple int/string parameters
 - yet still strongly typed

LINQ (F#) example (revisited)

```
let p = <@ fun e -> e.salary > 50000 @>  
  
query { for e in employees  
        where (%p e)  
        yield {name=e.name} }
```



employees

dpt	name	salary
"Product"	"Alex"	40,000
"Product"	"Bert"	60,000
"Research"	"Cora"	50,000
"Research"	"Drew"	70,000
"Sales"	"Erik"	200,000
"Sales"	"Fred"	95,000
"Sales"	"Gina"	155,000

```
select name  
from employees e  
where e.salary > 50000
```

name
Bert
Drew
Erik
Fred
Gina

λ

- Wong's system included λ -abstraction
 - anonymous functions/application
- These are potentially useful for writing programs that generate queries
- Proof did not handle general case though
 - measure-based
 - Only handles first-order case
- Later work [Cooper, DBPL 2009] showed how to handle arbitrary (nonrecursive) λ 's in queries

Dynamic queries

- Queries whose structure isn't determined until run time
- Simple example: predicates

```
type Predicate =  
  | Above of int  
  | Below of int  
  | And of Predicate × Predicate  
  | Or of Predicate × Predicate  
  | Not of Predicate
```

```
let rec P(t : Predicate) : Expr<int → bool> =  
  match t with  
  | Above(a) → <@ fun(x) → (%lift(a)) ≤ x @>  
  | Below(a) → <@ fun(x) → x < (%lift(a)) @>  
  | And(t, u) → <@ fun(x) → (%P(t))(x) && (%P(u))(x) @>  
  | Or(t, u) → <@ fun(x) → (%P(t))(x) || (%P(u))(x) @>  
  | Not(t) → <@ fun(x) → not((%P(t))(x)) @>
```

```
query { for e in employees  
       where (%(P (Above(50000))))  
       yield {name=e.name} }
```

Richer query results

- We might ask: what if we want queries to return nested results?
 - or function values?
 - or even sums/datatypes?
- Grust et al. explore an alternative approach based on translating queries to SQL:1999 "OLAP" operations
 - including nesting, functions/defunctionalization, and sums
 - depends on sophisticated SQL:1999 optimization/rewrite engine called Pathfinder [Grust et al. 2008]
 - Recent work on DSH based on flattening avoids this
- Our work [SIGMOD '14] extends normalization-based approach to handle nested results ("query shredding") but not clear how to handle other features.

Open questions

- Expressiveness/performance vs. integration
 - Tradeoff between simplicity of implementation and power of underlying query language
 - Updates: have not been studied in much depth
- Most work in statically typed languages; what about dynamic typing?
 - conversely, type providers very useful --> gradual types?
- Measuring usability/value of LINQ and related techniques
 - Common problem: query performance unpredictable/sensitive to small changes
- Adapting to other data-centric heterogeneous programming models (GPU, data-parallel, MapReduce, etc.)
 - see e.g. Delite framework and others

Summary

- Language-integrated query has been investigated for >30 years
- This talk: attempt to cluster (recent) work and bring out common themes
 - Low-level API/typed ASTs: more programmable but less convenient
 - Query DSLs: more convenient but less programmable;
 - Reinterpretation/query
- All three approaches require care if host language features (e.g. higher-order functions) are allowed in queries
- Some signs of convergence toward a common facility based on quotation/reflection (or comparable DSL embedding techniques)

Systems: LINQ query expressions

- Microsoft LINQ to SQL
- Query API includes Select, Where, GroupBy, many other (higher-order) operators
- Query API calls can be implemented directly or recorded as ASTs for lazy optimization / query generation
- More recently, *type providers* offer dynamically typed access to databases and other data resources
- LINQ also includes other things such as object-relational mapping, XML queries, which we do not consider here.

Systems: SML#

[Ohori & Ueno 2011]

- queries syntactically like SQL
- uses record typing
- no higher-order parameters

```
# val Q =  
  _sql db =>  
    select #person.name as name,  
           #person.age as age  
    from #db.people as person  
    where SQL.>= (#person.age, 25);
```

Systems: Ur/Web

[Chlipala 2011]

- queries embedded as typed DSL
- uses records/row typing
- operations are directly mappable to SQL
- implemented internally by translation to a typed AST for SQL-like queries (I believe, Adam correct me if I'm wrong)
 - query generation from AST straightforward; types ensure schema validity
 - query construction/higher-order parameters possible using AST

Systems: C# LINQ

[Meijer et al SIGMOD 2006]

- uses query-like syntactic sugar for quotation (see also [Bierman et al. OOPSLA 2006])
- queries (or other expressions of type $\text{Expr}<T>$) are implicitly quoted and can be manipulated at run time

Systems: Kleisli

- Kleisli [Wong JFP 2000]
 - implicit separation, best effort to find queries, then execute in-memory
 - solved DoE's "twelve impossible queries"
 - led to a successful (and proprietary) commercial product

Systems: LINQ

- Microsoft LINQ to SQL (C#, F#)
- for C# [Meijer et al SIGMOD 2006]
 - uses query-like syntactic sugar for quotation (see also [Bierman et al. OOPSLA 2006])
- for F# [Syme, ML 2006]
 - translates F# expressions to C# expressions, then uses C# LINQ library
 - currently based on *computation expressions* [Petricek & Syme PADL 2014]
 - did not provide systematic support for HO functions in queries, but our ICFP 2013 paper showed how to add this (P-LINQ, T-LINQ)
- LINQ also includes other things such as object-relational mapping, XML queries, which we do not consider here.

Systems: Links

- Links [Cooper, Lindley, Wadler, Yallop 2006]
 - initially, Kleisli-like
 - developed effects and higher-order normalization [Cooper 2009, ...] to address performance/reliability
 - several other non-DB-related features (web programming, typed actor-based concurrency)

Systems: Ferry

- Ferry
 - A functional query language [Grust et al. 2009]
 - Data model is *ordered* (builds on XML query techniques)
 - Allows nesting, higher-order, sums; supports aggregation & grouping
 - Implemented for C# LINQ, on top of Pathfinder
 - Database-Supported Haskell: provides Haskell front-end, translates to SQL:1999 via Pathfinder