Theory for the Working Database Programmer

A Historical Approach

Ryan Wisnesky
ryan@conexus.com

Conexus AI

Data Day 2020

$$\Sigma \dashv \Delta \dashv \Pi$$

Introduction

- In this talk I will take a historical approach to describing seminal database systems,
- drawing connections to category theory, functional programming, and database theory,
- with the goal of improving the audience's computer science fundamentals.
- Case studies:
  - Relational Algebra
  - Structural Recursion
  - Comprehension Notation
  - Lambda Calculus
  - Ongoing research on graph query languages (joint with Marko Rodriguez on mm-adt, and Joshua Shinavier on APG)
- `wisnesky.net/dataday.pdf`

Outline

- For each system, we will go over:
  - its history
  - its definition
  - an example
  - suggested conclusions about its utility (but draw your own!)
  - how it is applied today
- Please ask questions during the talk!
- About me: Stanford undergrad, trained in database theory at IBM Almaden, programming language theory at Harvard (PhD), and category theory at MIT (postdoc). My site: `wisnesky.net`. Currently I work on the categorical query language CQL at Conexus. Open-source CQL site: `categoricaldata.net`.
- If you like this material, consider applying for grad school!

Relational Algebra: History

- 1959: Tarski invents the relational model in a footnote in a graduate textbook on algebraic logic.
- 1969: Codd re-invents the relational model in an IBM journal.
- 1970: The relational model is published in a public journal.
- Initial reaction: "Codd's concept of data arrangement was seen within IBM as an 'intellectual curiosity' at best and, at worst, as undermining IBM's existing products."
- 1977: Oracle Founded.
- 1981: Codd wins Turing Award.
- 1983: IBM DB2 launches, and goes on to become one of IBM's most successful products.
- Today: civilization runs on SQL.

Relational Algebra: Definition and Example

- A *relational schema* consists of a set of relation names and their arities (number of columns).
  - Example: $Friend$ of arity 2, $Parents$ of arity 3.
- A *relational database instance* (on a particular schema) consists of a set D, called the *domain*, and for each relation name of arity n, an n-ary relation on D.
  - Example: $D = String,\ Friend = \{(Alice, Bob), (Bob, Charlie)\}$, $Parents = \{(Bob, Daniel, Ellen)\}$
- The relational algebra (on domain D) is the set of operations:
  - Select, Project, Cartesian Product, Union, Difference.
- Codd's theorem: the relational algebra is equivalent to *domain independent* first-order logic queries.
  - Domain Independent: $\{(x, z) \mid \exists y\ \text{Friend}(x, y) \wedge \text{Friend}(y, z)\}$. Result depends only on the input data.
  - Not Domain Independent: $\{(x, z) \mid \neg\text{Friend}(x, z)\}$. Result depends on the domain D.

Relational Algebra: Conclusions

- Claim: the relational algebra succeeded as a query language **because** it **is** first-order logic.
- Key Theoretical Results:
  - query evaluation in P (data complexity)
  - equivalence of conjunctive queries (select/from/where) in NP
  - query minimization and composition (view unfolding) algorithms
  - efficient algorithms using hash-tables, b-trees, etc.
- Caveat: many "relational" systems deviate from the relational model.
  - example: `NULL = NULL` is not true in SQL (SQL uses 3-valued logic)
- Caveat: checking domain independence is undecidable.

Interregnum

- Killer app in 1978: replacing graph (network) data models.
- 1980s-1990s: Cambrian explosion of data models: object-oriented, object-relational, datalog, many others.
- 1990s: relational algebra proves insufficient for querying the new data models, and researchers turn to structural recursion (next).
- Killer app in 2020: provide a baseline of expressive power, and serve as an Lingua Franca.

Structural Recursion: History

- 1888: First use of primitive recursion to define a function, by Dedekind
- 1923: Skolem invents primitive recursive arithmetic (PRA)
- 1941: Haskell Curry shows that PRA does not require logical quantifiers or connectives, only equations
- 1974: Goguen generalizes primitive recursion to *structural recursion* over inductively defined data types
- 1992: Tannen and others propose structural recursion as an implementation technique for object-oriented databases, relational databases, and others, and it remains widely used today.
- 1994: The "iterator model" of SQL evaluation is popularized by Graefe's Volcano system and remains widely used today.

Structural Recursion: Example

- In data processing, "lists of t", where t is a known type such String or Int, is the most common inductive data type, and structural recursion is called "fold", or sometimes, "reduce". In Haskell:

```
data IntList = Nil | Cons Int IntList

fold :: a -> (Int -> a -> a) -> IntList -> a
fold v f Nil = v
fold v f (Cons h l) = f h (fold v f l)

sum :: IntList -> Int
sum = fold (+) 0

sum (Cons 5 (Cons 7 (Cons 9 Nil))) = 5 + (7 + (9 + 0))
```

Structural Recursion: Definition

- An *inductive* set uniquely builds its elements in terms of other elements in a *well-founded* way. For example, one definition of the set $\mathbb{N}$ of natural numbers is:
  - 1 is in $\mathbb{N}$.
  - If n is in $\mathbb{N}$ then $n + 1$ is in $\mathbb{N}$.
  - $\mathbb{N}$ is the smallest set satisfying the above.
- Inductive sets can be *recursively* processed according to each clause above, for example, one definition of multiplication by 9 is:
  - $1 \times 9 = 9$
  - $(n + 1) \times 9 = n + (n \times 9)$
  - multiplication by 9 is unique function $\mathbb{N} \to \mathbb{N}$ satisfying the above
- Associated proof principle: to prove P holds for all natural numbers, prove $P(1)$ and that $P(n)$ implies $P(n + 1)$.

Structural Recursion

- Key Theoretical Results:
  - Equivalence of structural and primitive recursion.
  - "fold fusion": $\texttt{fold}\ f \circ \texttt{fold}\ g = \texttt{fold}\ (f \circ g)$, and other optimizations.
  - Most algorithms are primitive recursive.
- **Caveat**: Most collection types are not inductive! For example, bags are not inductively defined, but can be processed as though they are lists by those functions **that ignore order and repetition in the list**.
- Caveat: optimization in practice requires many variants of structural recursion that vary in their runtime properties
  - `foldl` vs `foldr`
  - `wiki.haskell.org/Zygohistomorphic_prepromorphisms`
- To get the best of structural recursion but without having to check well-formedness of user-defined recursions, researchers proposed using *comprehensions* as a user-facing query language on top of structural recursion (next).

Comprehension Notation: History

- 1901: Russell discovers that unrestricted set comprehension is inconsistent: let $R = \{x \mid x \notin x\}$. Then $R \in R$ if and only if $R \notin R$.
- 1922: Bounded comprehension notation placed on firm footing with invention of ZFC set theory.
- 1958: Unbounded comprehension notation placed on firm footing with invention of categorical topos theory.
- 1982: Moggi invents monadic do-notation, a generalization of comprehension notation suitable for many collections types.
- 1992: Peyton-Jones and Wadler connect do-notation to I/O, and the notation remains in wide use today.
- 1994: Wong defines the nested relational algebra as comprehension notation in the set monad, and the notation remains in wide use today.

Comprehension Notation: Example and Definition

- Example: let S be a set of integers. Then

$$\{a + b \mid a \in S, b \in S, a \neq b\}$$

  traverses S twice and returns a set containing the sum of all non-equal elements of S.
- When $L = \{2, 3\}$, we have $C = \{5\}$.
- Comprehension syntax can be implemented with the primitives:

$$map : (a \to b) \to Set\ a \to Set\ b$$

$$filter : (a \to Bool) \to Set\ a \to Set\ b$$

$$empty : Set\ a \qquad singleton : a \to Set\ a$$

$$union : Set\ (Set\ a) \to Set\ a$$

$$tensor : a \to Set\ b \to Set\ (a, b)$$

Comprehension Notation: Conclusions

- Claim: comprehension notation succeeded as a query language because it literally is mathematical notation.
- Key Theoretical Results:
  - Normal form for comprehensions.
  - Equivalent to nested relational algebra.
  - Translation from comprehensions to structural recursion.
  - Works uniformly over most collection types (list, bag, set, etc)
- "Caveats": join is not a primitive; cannot aggregate.
- Implemented by most high-level programming languages (Python, Java, Haskell, etc) and under the hood in many data migration and integration systems (eg IBM's HIL)
- Next up: how to embed comprehensions in general purpose languages (leads to λ-calculus).

λ-Calculus: History

- 1932: Church introduces a predecessor to λ-calculus and proposes it as a logic.
- 1935: Kleene and Rosser show this system is logically inconsistent.
- 1936: Church isolates the portion relevant to computation, and calls it the untyped λ-calculus.
- 1940: Church introduces a weaker, but logically consistent system, called the simply typed λ-calculus.
- 1970: Scott invents the first non-trivial model of untyped λ-calculus.
- 1979: Martin-Lof invents dependent type theory.
- 1990: Haskell 1.0 released.
- 1980s-1990s: Intense work on implementation of many λ-calculi, which remain the cornerstone of programming languages to this day.

λ-Calculus: Definition and Examples

- In this talk we will describe untyped λ-calculus, but typed λ-calculus is much easier to work with formally.
- A *term* is inductively defined as either:
  - a *variable*, such as x or y, or
  - an *application* of a term f to a term g, written fg, or
  - an *abstraction* of a term f over a variable x, written $\lambda x.f$.
- Caveat: we must not distinguish terms that differ only by names of bound variables, e.g. $\lambda x.x = \lambda y.y$.
- A single equation called β-reduction provides computation:

  $$(\lambda x.f)g = f[x \mapsto g]$$

  where $f[x \mapsto g]$ indicates the substitution of g for x in f.
- Examples:
  - identity function: $\lambda x.x$
  - identity function applied to itself: $(\lambda x.x)(\lambda x.x) = \lambda x.x$
  - Y-combinator: $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. We have $Y(g) = g$

λ-calculus: Discussion

- Key Theoretical Results:
  - Untyped λ-calculus is Turing complete; simply typed λ-calculus is not.
  - Proofs in various logics can be represented in various typed λ-calculi (Curry-Howard isomorphism).
  - λ-calculi are the internal languages of cartesian closed categories.
  - λ-calculi admit eager and lazy evaluation strategies and have equivalent variable-free forms (combinatory logics).
- Application: adding comprehension notation to a λ-calculus results in a *language-integrated query system*, such as Microsoft LINQ, Data Parallel Haskell, Monad Comprehension Calculus, and more.
- "Caveat": Programming languages that contain side effects, such as I/O, can't easily be modeled in λ-calculus because functions in these languages need not be functions in the sense of math. Example: a $fire - missiles$ function can run out of missiles.
- Conclusion: almost all programming languages extend a λ-calculus or variable-free equivalent.

Current Research on Graphs

- As a data model, graphs haven't received as much attention from academics as the data models discussed today.
- Speculation: this is because most graphs are not inductive, and a (the?) natural query language for them is based on stateful edge walks a la Apache Tinkerpop.
- Current work:
  - Marko Rodriguez and I are formalizing gremlin as a λ-calculus and are providing a sound mathematical basis for `mm-adt`. Key results: use of abstract interpretation and equational re-writing
  - Joshua Shinavier and I formalized Uber's algebraic property graph data model (APG) using category theory and it may be included in Tinkerpop 4. Paper URL: `arxiv.org/abs/1909.04881`
  - APG and `mm-adt` are fragments of a more general approach to data based on *category theory*, which we at Conexus have been exploring via our categorical query language CQL. `categoricaldata.net`

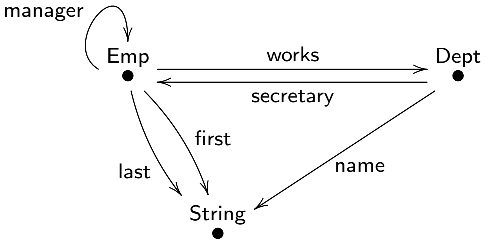
Category Theory

- Category theory was invented in 1946 to migrate **theorems** from one **area of mathematics** to another, so it is a very natural language with which to describe migrating **data** from one **schema** to another.
- A category $\mathcal{C}$ consists of
  - a set of *objects*, $\mathsf{Ob}(\mathcal{C})$
  - forall $X, Y \in \mathsf{Ob}(\mathcal{C})$, a set $\mathcal{C}(X, Y)$ of *morphisms* a.k.a *arrows*
  - forall $X \in \mathsf{Ob}(\mathcal{C})$, a morphism $id \in \mathcal{C}(X, X)$
  - forall $X, Y, Z \in \mathsf{Ob}(\mathcal{C})$, a function $\circ : \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) \to \mathcal{C}(X, Z)$ s.t.

$$f \circ id = f \qquad id \circ f = f \qquad\qquad (f \circ g) \circ h = f \circ (g \circ h)$$

- The category $\mathbf{Set}$ has sets as objects and functions as arrows, and the category $\mathbf{Haskell}$ has types as objects and programs as arrows.

Categorical Schemas and Instances



$$[\text{manager.works}] = [\text{works}] \qquad [\text{secretary.works}] = []$$

Emp				
ID	**mgr**	**works**	**first**	**last**
101	103	q10	Al	Akin
102	102	x02	Bob	Bo
103	103	q10	Carl	Cork

Dept		
ID	**sec**	**name**
q10	101	CS
x02	102	Math

String
ID
Al
Bob
...

A CQL Schema: Code

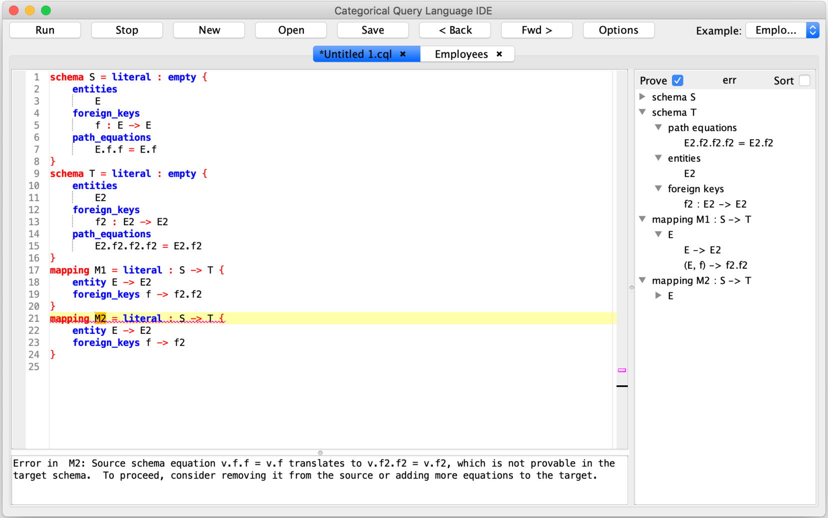
```
entities
    Emp
    Dept

foreign keys
    manager : Emp -> Emp
    works : Emp -> Dept
    secretary : Dept -> Emp

attributes
    first last : Emp -> string
    name : Dept -> string

path equations
    manager.works = works
    secretary.works = Department
```

The CQL IDE


Categorical Query Language IDE
Run
Stop
New
Open
Save
< Back
Fwd >
Options
Example:
Emplo...
*Untitled 1.cql
Employees
schema S = literal : empty {
entities
E
foreign_keys
f : E -> E
path_equations
E.f.f = E.f
}
schema T = literal : empty {
entities
E2
foreign_keys
f2 : E2 -> E2
path_equations
E2.f2.f2.f2 = E2.f2
}
mapping M1 = literal : S -> T {
entity E -> E2
foreign_keys f -> f2.f2
}
mapping M2 = literal : S -> T {
entity E -> E2
foreign_keys f -> f2
}
Error in M2: Source schema equation v.f.f = v.f translates to v.f2.f2 = v.f2, which is not provable in the target schema. To proceed, consider removing it from the source or adding more equations to the target.
Prove
err
Sort
schema S
schema T
path equations
E2.f2.f2.f2 = E2.f2
entities
E2
foreign keys
f2 : E2 -> E2
mapping M1 : S -> T
E
E -> E2
(E, f) -> f2.f2
mapping M2 : S -> T
E

Summary

- We discussed four seminal systems:
  - Relational Algebra
  - Structural Recursion
  - Comprehension Notation
  - Lambda Calculus
- and three next-generation systems based on category theory:
  - The categorical query language CQL (Conexus)
  - Algebraic Property Graphs (Uber)
  - mm-adt: A multi-model abstract data type (RRedux)
- Get involved! All inquiries welcome at ryan@conexus.com.