Query Combinators for Clinical Research

Clark C. Evans
cce@clarkevans.com

Kyrylo Simonov
xi@resolvent.net

Prometheus Research, LLC

Draft of June 6, 2018

Abstract

A clinical research workflow involves many participants including a primary investigator, a statistician, and an informatician. In current practice, these participants typically use different tools that are not integrated. As a result, data sourcing, clinical knowledge, and analysis methods used to produce clinically relevant findings may be opaque, hard to discuss, and challenging to reproduce.

This paper shows how a new computational method, Query Combinators, might be used to create a data processing environment shared among the entire clinical research team. For a given research context, a specialized technical language can be created that represents unique data sources, analysis methods, and domain knowledge. Research questions can then have an intuitive, high-level form that can be reasoned about and discussed. A hypertension medication effectiveness question is explored to demonstrate how a specialized database query language could be incrementally constructed, applied, and then reused.

1 Introduction

To facilitate the access of institutional data resources for clinical research projects, medical researchers engage informaticians who have specialized database skills and electronic health record expertise [4]. In the clinical research workflow illustrated in Figure 1, an informatician participates as a member of the research team, generating data sets from institutional data resources [3]. These data sets are the output of technical processes including database queries as well as data integration and data cleaning programs.

To generate relevant data sets, an informatician enters an iterative dialogue, or *query mediation*[2] with research team members. To locate and repurpose relevant data, the informatician must learn about the principal investigator's research context and specialized vocabulary. At the same time, the investigator may need to revise the research plan to reflect availability of information or to consider limitations upon how repurposed data may be interpreted. Therefore, query mediation is often a bidirectional communication.

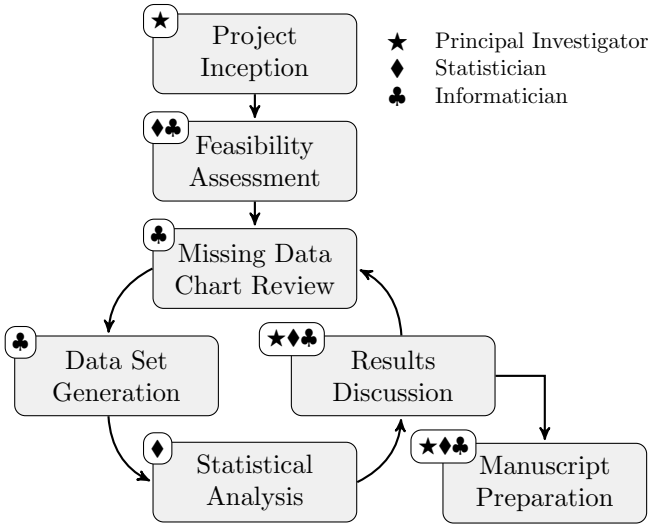

Figure 1: Clinical Research Workflow at Columbia University's urology department as observed by Hruby [3]

In this clinical research workflow, generated data sets and statistical outputs are the working artifacts that drive research discussion. The technical *processes* which generate these artifacts are often a secondary concern. Typically, the principal investigator has little access to this technical realm. Inability to name, document, and encapsulate the complexities encountered in data set generation and statistical analysis can frustrate discussion and hinder reproducibility.

Complementary to the review of generated data sets and statistical outputs, what the research team could have a meaningful discussion of the technical processes which produce these artifacts? To entice a principal investigator into a more technical dialogue, informatics tooling must become much more conceptual, capable of encapsulating complexity with a reflection of the researcher's context and vocabulary.

Query Combinators offer a formal yet implementable computational method to address this unmet need. Specifically, Query Combinators can be used to compose high-level processes with a nomenclature suitable to medical research discussion.

In a prior paper[1], Query Combinators were shown to be rigorously defined, eminently usable, and at least as powerful as contemporary alternatives such as the Structured Query Language (SQL) or the "R" `dplyrs` package. This paper demonstrates how Query Combinators permit the construction of query languages that are tailored to specific clinical analysis domains.

2 Hypertensive Drug Efficacy

To demonstrate the application of Query Combinators to clinical research, this paper describes the process of converting a researcher's question into an executable specification. Consider the following question.

> Within 6 months of a hypertension diagnosis, when an anti-hypertensive medication was added or intensified, was there a blood pressure decrease of 5 mmHg or more within 5 days after the medication adjustment?

There are several concepts densely packed within this apparently simple question. To capture the nuances of each concept, a dialogue with the researcher is needed. The informatician then implements and documents a set of *query combinators* that encapsulate these concepts.

Table 2 lists a few combinators relevant to this question and abbreviated query mediation notes.

Combinator	Query Mediation Notes
hypertension_diagnosis	exclude pregnancy & kidney failure
antihypertensive_medication	a product list is provided
added_or_intensified	new therapy or larger dose
blood_pressure_decrease	of both systolic & diastolic
medication_adjustment	change of daily medication
normalized_active_ingredient	normalize dosage records across compound products

Table 1: Hypertensive Query Combinators

Although combinators typically reflect the intent and vocabulary of the investigator, sometimes new concepts related to data sources or analysis methods are required. These are also expressed as combinators and given a name so they can be discussed among research team members. Let's presume this researcher is interested in active ingredients, not compounded pharmaceutical products as stored in the information system. Hence, normalized_active_ingredient is defined to mean medication records that are normalized across compounded products by active ingredient.

With these combinators defined, the inquiry could be translated into a high-level query that is written with formal terminology relevant to the research project.

Query 2.1 (Hypertensive Drug Efficancy).

```
patient
.medication_adjustment(
  normalized_active_ingredient(
    antihypertensive_medication))
:filter(
  added_or_intensified &
  during(previous(6months),
    patient.hypertension_diagnosis)
:define(
  is_effective :=
    during(subsequent(5days),
      patient.blood_pressure_decrease(5mmHg)))
:group(normalized_active_ingredient)
:select(normalized_active_ingredient,
  count(medication_adjustment:filter(is_effective)),
  count(medication_adjustment:filter(not(is_effective))))
```

This query is quite compact and unambiguous. Its high-level vocabulary and encapsulation of lower-level concerns permits it to closely follow the research inquiry.

Step 1: For each patient: (a) find the correlated anti-hypertensive medication records, (b) normalize those records to return dosage by active ingredient, and then, (c) compute the adjustments to that patient's daily medication regime.

Step 2: Consider medication adjustments that reflect additions or intensifications; include only those that have a qualifying hypertension diagnosis within the previous 6 months.

Step 3: Define what it means to be an effective medication adjustment: did the patient have a blood pressure decrease of 5mmHg within the subsequent 5 days?

Step 4: Group these medication adjustments by the normalized active ingredient and then count the number of effective and ineffective adjustments.

As this query is elaborated, combinators will be named, defined, and used to encapsulate data sources, analysis methods and clinical knowledge. Collectively these combinators form a shared vocabulary that increases understanding among all team members and faciltates reuse across a variety of research projects.

3 Thinking in Query Combinators

Before proceeding to the query presented in Section 2, it is helpful to discuss the reasoning and mathematics underlying Query Combinators.

For this section, consider a simplified Clinical Research Data Repository (CRDR) with patients and coded conditions. This database might be described with a series of statements enumerated in Table 2.

1. This Database has a set of Patient records.
2. Each Patient record has an identifier which uniquely locates it within the Database.
3. Each Patient record has a mandatory birthdate.
4. Each Patient is associated with a set of Condition records which represent problems, diagnoses or any other issues of concern.
5. Each Condition has a mandatory category which is a *SNOMED-CT* concept such as `59621000|Essential hypertension`.
6. Each Condition has a mandatory onset to record approximately when the problem started.
7. Each Condition has an optional abatement to record if/when the problem went into remission.

Table 2: CRDR Database Description

Information Modeling Approaches

Following are three equivalent information models for the CRDR database described in Table 2. These models differ in how they represent entity classes, relationships between entities, datatypes, and attributes.

Figure 2 shows a standard tabular database model where information is structured as *tables* and *columns*.

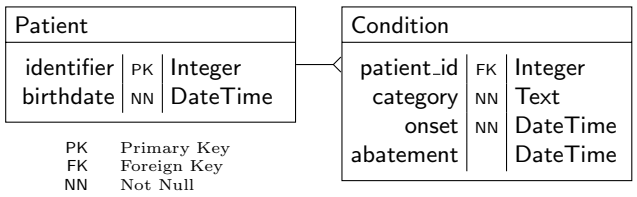

Figure 2: Tabular Model for CRDR

In the tabular model, entity classes, such as Patient are represented as tables, and attributes, such as birthdate, as columns. Column datatypes are provided. Relationships between entity classes are encoded as *foreign key* constraints. In this example, patient_id on the Condition table refers to identifier on the Patient table.

Query construction with this tabular model is well supported by a number of languages and tools, including the Structured Query Language (SQL) and Language INtegrated Query (LINQ). However, neither SQL nor LINQ have proven to be a suitable query language for casual use by domain experts.

Figure 3 shows a functional model where information is conceptualized as a graph of *nodes* and *arcs*.

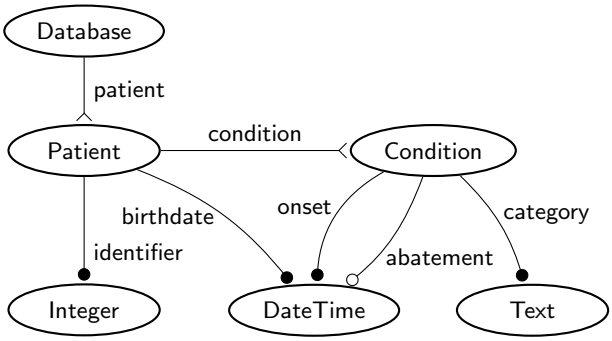

Figure 3: Functional Model for CRDR

In this diagram, entity classes and datatypes are represented as nodes, while attributes and relationships are represented by arcs between them. The database as a whole is also represented as a node in this model and it is connected to entity classes such as Patient with a relationship arc of the same name, e.g. patient.

This functional diagram makes clear that each attribute and relationship has a certain input and output, at the head and tail of each arc. Moreover, this unified treatment of relationships and attributes suggests that one might traverse data by connecting arcs.

While promising, query construction with this functional model is not well supported. Languages such as the Functional Query Language (FQL) that are based on this model have not proven to be practical.

Figure 4 shows an equivalent hierarchical model where information is visualized as *elements* in a tree.

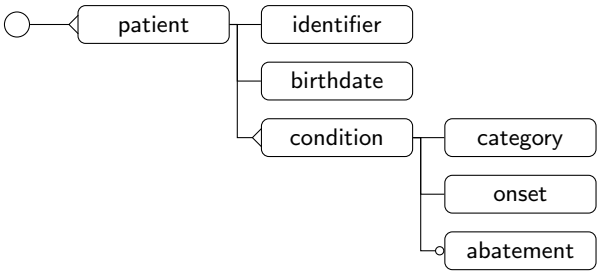

Figure 4: Hierarchical Model for CRDR

In this diagram, relationships between entities and attributes are represented as tree elements; entity classes and data types are implicit. Similar to the functional model, the database itself is expressly represented, this time as the root of the tree.

Query construction with this hierarchical model is popularized by the eXtensible Markup Language (XML) data format and the excellent, but limited, XML

Path Language (XPath). XPath has proven that path-based queries, such as count(/patient/condition), are quite accessible to domain experts.

This hierarchical perspective has been found to match domain experts' intuitive understanding of their data, and thus motivates a new way of thinking about information processing. Elements in the hierarchical model of Figure 4, imply path-based navigation. Further, these tree elements directly correspond to the arcs of the functional model as shown in Figure 3.

Query Combinators present a user experience motivated this hierarchical model backed by the power and flexibility provided by the functional model. This user experience is further compatible with data that is stored in a contemporary tabular model.

An Algebra of Queries

Historically, algebras have provided powerful reasoning frameworks and enabled usable systems. Algebraic expressions can be incrementally constructed, rearranged, and reduced to help build meaningful models of reality.

For example, elementary algebra (or Arithmetic) is an algebra of numbers. This algebra consists of numeric primitives (0, 1, 2, etc.) and a set of numeric operations, including operators ($\times$, $\div$, $+$, etc.) and functions (*sqrt*, *cos*, etc.), that can be used to construct numeric expressions such as $sqrt(49) \times (5 + 1)$.

This familiar algebra of numbers is not the only algebra. For example, in the Relational Algebra upon which SQL is loosely based, elements are sets and the operations include restriction, projection, product, union and difference. Each operation takes one or more sets as an input and produces a set for its output.

Query Combinators are an algebra of query functions.

This algebra's elements, or *queries*, represent relationships among class entities and datatypes. For example, condition in Figure 3 names a query primitive that, for each Patient record, yields a sequence of correlated Condition records.

This algebra's operations, or *combinators*, are applied to construct query expressions. For example, count(condition) is a query expression that is constructed by applying the count combinator to the condition query. Observe that count(condition) is itself a query; for each Patient record, count(condition) yields the number of correlated Condition records.

Notice that this algebra is significantly different from the Relational Algebra. The query condition does not name a table of Condition records; instead, the condition query is a function from each Patient to a sequence of associated Condition records. Further, the combinator count does not directly count records. Instead, the count combinator is used to build queries, such as count(condition), which do the actual counting.

In particular, Query Combinators are a many-sorted, or typed algebra. This is different from Arithmetic in which all elements are numbers and are treated uniformly. Every query has a type, or *query signature*, that specifies the expected input and output of the query. For example, the signature of the query condition is (Patient $\rightarrow$ Condition*). The output cardinality in a query signature is *singular* unless it is either marked as *plural* with * or marked as *optional* with $^?$.

This typed algebra of query functions permits a query processor to automatically track user context, providing an intuitive yet robust query language.

Query Primitives

This algebra has two kinds of queries: primitives and expressions. Query primitives, such as condition, are elementary building blocks; they reflect functional relationships among data within a given data source.

Consider again the CRDR database described in Table 2, perhaps stored in a tabular database as shown in Figure 2. To interrogate this database with combinators, it must first be converted into a functional form, as shown in Figure 3. The Database as a whole, entity classes (such as the Patient table), and scalar types (e.g. DateTime) become nodes in this graph.

Query primitives are represented as the arcs in this functional graph. They include the relationship patient between the Database each Patient entity, the relationship condition between a Patient and correlated Condition records, and relationships such as birthdate between a Patient and a scalar DateTime value.

Table 3 lists the query primitives for the CRDR presented previously. In this table, each query, such as condition, is associated with its signature, e.g. Patient $\rightarrow$ Condition*.

Primitive	Signature
patient	Database $\rightarrow$ Patient*
identifier	Patient $\rightarrow$ Integer
birthdate	Patient $\rightarrow$ DateTime
condition	Patient $\rightarrow$ Condition*
category	Condition $\rightarrow$ Text
onset	Condition $\rightarrow$ DateTime
abatement	Condition $\rightarrow$ DateTime$^?$

Table 3: Query Primitives for CRDR

Constants are also considered primitive queries. For example, `'Hello World'` is a query that yields the same scalar value, *Hello World*, regardless of its input. Hence, this query has a signature of Any $\rightarrow$ Text.

When query primitives are arranged hierarchically, as shown in Figure 4, query signatures, the input and output types of each query, fade into the background. They become details to be managed by the query processor,

freeing the user to think in terms of relationships and combinations of relationships that reflect higher-level domain-specific meaning.

Query Expressions

Query expressions, such as count(condition) are constructed by applying combinators, such as count to queries, such as condition. In particular, count takes any query and makes a combined query that, for each input, yields the count of associated outputs.

Combinators, such as count, don't have signatures, instead, each has a rule that describes the signature of the query it constructs based upon the signature of its inputs. Table 4 shows the signature rule for the count combinator. When count is applied to any query f with input A and output B^*, the constructed query, count(f), has an input of A and an output of Integer.

f	$A \to B^*$
count(f)	$A \to$ Integer

Table 4: Count Combinator

By substituting patient for f as shown in Table 5, the query signature for count(patient) is computed be Database → Integer. Hence, for a given Database, count(patient) yields a singular integer value, the number of Patient records in that database.

patient	Database → Patient*
count(patient)	Database → Integer

Table 5: count() applied to patient query

Path-based navigation among entities is expressed as a binary combinator indicated with the period (.). Consider the patient and condition arcs in Figure 3. These arcs can be connected to compose a new arc, patient.condition. This navigation can also be visualized as traversing down the tree from patient to condition in the hierarchical model of Figure 4.

More formally, query composition builds a new query by chaining the output of one as the input of the other. This composition is permitted when two queries, f and g, have a shared intermediate type B, as shown in Table 6. Thus, $f.g$ is interpreted as $g(f(x))$ for any x.

f	$A \to B^*$
g	$B \to C^*$
$f.g$	$A \to C^*$

Table 6: Composition Combinator

For example, when applied to a given Database, the query patient.condition would feed the output of patient (a sequence of Patient records) as the input of condition to produce a sequence of Condition records. This is shown in Table 7.

patient	Database → Patient*
condition	Patient → Condition*
patient.condition	Database → Condition*

Table 7: Composition of patient and condition

Query composition is *monadic*, that is, outputs are treated as streams of values. Nested sequences are automatically flattened. Further, any mandatory value can be treated as an optional value, and any optional value can be treated as a sequence of zero or more values. Monadic composition allows the user to compose queries without having to be concerned about containers or cardinality.

Query expressions are algebraic. So long as each combinator's rule can be followed, arbitrarily sophisticated expressions can be generated. Since count and patient.condition are defined, count(patient.condition) is also a valid query expression in the algebra: it counts the number of condition records across all patients in the database. This query's signature can be automatically computed as Database → Integer.

In this algebra, the order of operations matters. For example, count(patient.condition) is different from patient.count(condition). The latter yields a list of integers, one for each patient. Each integer in this list would reflect the number of conditions for each enumerated patient. These subscores can then be used to compute the average number of conditions across all patient records.

Query 3.1 (Average # of Conditions by Patient).

mean(patient.count(condition))

Additional combinators, such as filter, group, sort, define, select are used without definition here. They are explored in the paper by the same authors entitled *Query Combinators*[1].

Combinators are Extensible

Query Combinators are completely extensible. There is nothing special in how count or filter operate within the query language. Software developers can extend this query system with any sort of data access adapters, processing algorithms, or statistical functions.

In particular, any scalar function or operation can be *lifted* to a query combinator. For example, addition, which is a binary operator over numbers, can be converted to a combinator on queries with numeric output.

Another example is a function now() that returns the current DateTime value. This scalar function can be lifted to a combinator now() which takes no arguments and constructs a query; this query would then yield the current time. The operation of the now() combinator is once indirect, adapted to cooperating within a higher-order information processing context.

Domain Specific Query Languages

The logic of Query Combinators permits not only the rigorous definition of composed queries, but also the creation of custom, domain specific vocabularies.

Suppose that an informatician would like to conduct a feasibility assessment to see if the CRDR database has at least some candidate patients relevant to this hypertension effectiveness inquiry. A simple test might be framed as follows:

> How many patients, ages 18 or older, have an active diagnosis of *Essential Hypertension*?

The first step is to create the necessary higher-level combinators: one that defines essential hypertension, one that computes a patient's age, and one that tests for an active diagnosis.

Query 3.2 (Inquiry Concepts).

```
essential_hypertension := '59621000'
age := years(now() − birthdate)
has_active_diagnosis(x) :=
    exists(condition.filter(
            category = x
             & is_null(abatement)))
```

First, essential_hypertension is defined to be the SNOMED-CT concept 59621000. While this is not a comprehensive test for hypertension, it might be good enough as a first-pass feasibility check.

Second, age is defined to be the timespan from the Patient's birthdate to the system time, truncated to a yearly resolution. It's a rather obvious definition, but there's no use in letting this logic bleed into the more critical aspects of the inquiry.

Third, has_active_diagnosis(x) encapsulates the logic that a patient has an active condition of a particular code category, represented by the variable x, for which an abatement has not been entered. This combinator wraps data model nuances into words that reflect the inquiry's use of that model.

With these reusable combinators defined, the translated query for the inquiry above is succinctly expressed:

Query 3.3 (Adults /w Hypertension).

```
patient
:filter (age >= 18
        & has_active_diagnosis(
            essential_hypertension))
:count
```

One minor syntax note: with Query Combinators' pipeline notation, $x : f$ is equivalent to $f(x)$. This allows queries to be written in a sequential rather than nested manner. For example, count($query$) could be written $query$: count. This syntax becomes valuable as queries grow more complex. If a combinator has more than one argument, then $x : f(y)$ is equivalent to $f(x, y)$. Hence, filter(patient, age >= 18) could be written patient:filter(age > 18).

What is most important about this query, and indeed the general approach, is that it could be shared, reviewed, and discussed with the entire clinical research team. At the top-level, the query is seen to generally follow the question. Yet, the terms used (age, essential_hypertension, and has_active_diagnosis) encapsulate complexity, can be independently reviewed, and quite possibly reused in multiple contexts.

With modest training and plenty of examples, casual domain experts, including medical research assistants and principal investigators, can be taught to read and understand these queries. Moreover, informaticians can customize the query vocabulary to fit research needs.

4 Hypertensive Question

(remainder of paper is being written)

...

Loading FHIR/SNOMED Data

...

Hypertension Diagnosis

...

Anti-Hypertensive Medication

...

Adjustment

...

Blood Pressure Decrease

...

Bibliography

[1] C. C. Evans and K. Simonov. Query combinators. *CoRR arXiv*, abs/1702.08409, 2017.

[2] G. Hruby, J. J Cimino, V. Patel, and C. Weng. Toward a cognitive task analysis for biomedical query mediation. 2014:218–22, 04 2014.

[3] G. Hruby, J. McKiernan, S. Bakken, and C. Weng. A centralized research data repository enhances retrospective outcomes research capacity: A case report. 20, 01 2013.

[4] G. W. Hruby, M. R. Boland, J. J. Cimino, J. Gao, A. B. Wilcox, J. Hirschberg, and C. Weng. Characterization of the biomedical query mediation process. *AMIA Jt Summits Transl Sci Proc*, 2013:89–93, 2013.