Query Combinators

Clark C. Evans
cce@clarkevans.com

Kyrylo Simonov
xi@resolvent.net

Prometheus Research, LLC

Draft of November 20, 2016

Abstract

We introduce Rabbit, a combinator-based query language. Rabbit is designed to let data analysts and other accidental programmers query complex structured data.

We combine the functional data model and the categorical semantics of computations to develop denotational semantics of database queries. In Rabbit, a query is modeled as a Kleisli arrow for a monadic container determined by the query cardinality. In this model, monadic composition can be used to navigate the database, while other *query combinators* can aggregate, filter, sort and paginate data; construct compound data; connect self-referential data; and reorganize data with grouping and data cube operations. A context-aware query model, with the input context represented as a comonadic container, can express query parameters and window functions. Rabbit semantics enables pipeline notation, encouraging its users to construct database queries as a series of distinct steps, each individually crafted and tested. We believe that Rabbit can serve as a practical tool for data analytics.

1 Introduction

Combinators are a popular approach to the design of compositional domain-specific languages (DSLs). This approach views a DSL as an algebra of self-contained processing blocks, which either come from a set of predefined atomic *primitives* or are constructed from other blocks using block combinators.

The combinator approach gives us a roadmap to design a database query language:

- define the model of database queries;
- describe the set of primitive queries;
- describe the combinators for making composite queries.

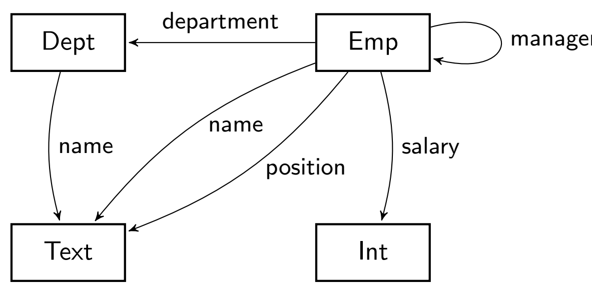

Figure 1: Sample database

To elaborate on this idea, we need some sample structured data. Throughout this paper, we use a simple database that contains just two classes of entities: *departments* and *employees*. Each department entity has one attribute: *name*. Each employee entity has three attributes: *name*, *position* and *salary*. Each employee belongs to a department. An employee may have a *manager*, who is also an employee.

In Figure 1, the structure of the sample database is visualized as a directed graph, with attributes and relationships (arcs) connecting entity classes and attribute types (graph nodes). This diagram may suggest that we view attributes and relationships as functions with the given types of input and output, for example

$$\begin{aligned} \mathsf{department} &: \mathsf{Emp} \to \mathsf{Dept}, \\ \mathsf{name} &: \mathsf{Dept} \to \mathsf{Text}. \end{aligned}$$

This is known as the functional database model [16, 22].

This model provides us with a starting point on our combinator roadmap. Indeed, a database query could be seen as a function; then, a set of primitive queries is formed by all the attributes and relationships, while function composition becomes a binary query combinator. With these considerations, we can write our first composite query.

Example 1.1 *Given an employee entity, show the name of their department.*

$$\mathsf{department.name} : \mathsf{Emp} \to \mathsf{Text}$$

In this example, $\mathsf{department.name}$ is a query written in Rabbit notation, and $\mathsf{Emp} \to \mathsf{Text}$ is its signature. The period ("$\boldsymbol{.}$") denotes the composition combinator, which is a polymorphic binary operator with a signature

$$- \boldsymbol{.} - : (A \to B,\ B \to C) \to (A \to C).$$

Even though this query model can express one database query, it does not seem to be powerful enough to become the foundation of a query language. What is this model missing?

First, it is awkward that a query always demands an input. It means that we cannot express an input-free query like *show a list of all employees.*[1]

Further, although the relationships are bidirectional, the model only covers one of their directions. Indeed, we chose to represent the relationship between departments and employees as a primitive with input Emp and output Dept. However, we may just as well be interested in finding, *for any given department, the corresponding list of employees.* It would be natural to add a primitive for the opposite direction, but it cannot be encoded as a function because its signature $\mathsf{Dept} \to \mathsf{Emp}$ would incorrectly imply that there is exactly one employee per department. Thus, the query model is unable to express multivalued or *plural* relationships.

The model also fails to capture the semantics of *optional* attributes and relationships. Such is the relationship between employees and their managers, which, according to Figure 1, should be encoded by a primitive with signature $\mathsf{Emp} \to \mathsf{Emp}$. But this signature implies that every employee must have a manager, which is untrue. Apparently, a pure functional model is too restrictive to express the variety of relationships between database entities.

This paper shows how to complete this query model and build a query language on top of it. It is organized as follows.

In Section 2, we show how to represent optional and plural relationships using the notion of query cardinality, which, following the approach of categorical semantics of computations [18], determines the monadic container for the query output. This lets us establish a compositional model of database queries.

In Section 3, we show how common data operations can be expressed as query combinators. Specifically, we describe combinators that extract, aggregate, filter, sort and paginate data; construct compound data; and connect self-referential data.

In Section 4, we show how grouping and data cube operations can be implemented as combinators that reorganize the intrinsic hierarchical structure of the database.

In Section 5, using the approach to the semantics of dataflow programming [25], we extend the query model to include a comonadic query context, which allows us to express query parameters and window functions.

In Section 6, we summarize the query model and briefly discuss some related work.

[1] We italicize *business questions* that specify database queries.

2 Query Cardinality

In Section 1, we suggested that a database query could be modeled as a function. However, this naïve model failed to represent optional and plural relationships as well as queries lacking apparent input. In this section, we resolve these issues by introducing the notion of *query cardinality.*

We found it difficult to model these two relationships:

(i) *An employee may have a manager.*

(ii) *A department is staffed by a number of employees.*

We were also puzzled on how to express input-free queries such as:

(iii) *Show a list of all employees.*

We could attempt to represent optional and plural output values as instances of the container types

$$\mathsf{Opt}\{A\} \quad \text{and} \quad \mathsf{Seq}\{A\},$$

where the *option* container $\mathsf{Opt}\{A\}$ holds zero or one value of type A, and the *sequence* container $\mathsf{Seq}\{A\}$ holds an ordered list of values of type A. Using these containers, relationships (i) and (ii) could be expressed as primitive queries with signatures

$$\begin{aligned} \mathsf{manager} &: \mathsf{Emp} \to \mathsf{Opt}\{\mathsf{Emp}\}, \\ \mathsf{employee} &: \mathsf{Dept} \to \mathsf{Seq}\{\mathsf{Emp}\}. \end{aligned}$$

Moreover, we could guess the output of query (iii). Indeed, *a list of all employees* can only mean $\mathsf{Seq}\{\mathsf{Emp}\}$.

To describe the input of query (iii), we introduce a *singleton* type

$$\mathsf{Void}.$$

The type Void has a unique inhabitant ($\top : \mathsf{Void}$), and because there is no freedom in choosing a value of this type, it can designate input that can never affect the result of a query. Thus, we can express (iii) as a *class* primitive

$$\mathsf{employee} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}.$$

Note that even though both (ii) and (iii) are denoted by the same name, we could always distinguish them by their input type.

Unfortunately, although containers let us represent optional and plural output, they do not compose well.

For example, it is tempting to express *for a given employee, find their manager's salary* as a composition

$$\mathsf{manager}.\mathsf{salary}, \tag{$\star$}$$

or *show the names of all employees* as

$$\mathsf{employee}.\mathsf{name}. \tag{$\star\star$}$$

However, if we look at the signatures of the components

$$\begin{aligned} &\mathsf{manager} : \mathsf{Emp} \to \mathsf{Opt}\{\mathsf{Emp}\}, && \mathsf{salary} : \mathsf{Emp} \to \mathsf{Int}, \\ &\mathsf{employee} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}, && \mathsf{name} : \mathsf{Emp} \to \mathsf{Text}, \end{aligned}$$

we see that their intermediate domains do not agree, which means their compositions are ill-formed.

To make the queries compose again, we should distinguish between the output type of a query and its cardinality.[2] For example, we should say that query (i) is an optional query from Emp to Emp, (ii) is a plural query from Dept to Emp, and (iii) is a plural query from Void to Emp. Then, any two queries should compose, regardless of their cardinalities, so long as they have compatible intermediate types; furthermore, the least upper bound of their cardinalities is the cardinality of their composition.

Specifically, given two queries

$$p : A \to M_1\{B\}, \qquad q : B \to M_2\{C\}$$

we first promote their output to a common cardinality

$$M = M_1 \sqcup M_2,$$

and then use the *monadic composition* combinator

$$- \,.\, - : (A \to M\{B\},\ B \to M\{C\}) \to (A \to M\{C\})$$

to construct

$$p\,.\,q : A \to M\{C\}.$$

Using this rule, we can justify the queries ($\star$) and ($\star\star$) and give them signatures

$$\begin{aligned} \mathsf{manager}.\mathsf{salary} &: \mathsf{Emp} \to \mathsf{Opt}\{\mathsf{Int}\}, \\ \mathsf{employee}.\mathsf{name} &: \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Text}\}. \end{aligned}$$

Let us work out the details. Query cardinalities are ordered with respect to inclusions

$$A \sqsubseteq \mathsf{Opt}\{A\} \sqsubseteq \mathsf{Seq}\{A\},$$

which, using the notation for container instances

$$\perp,\ \ulcorner a \urcorner : \mathsf{Opt}\{A\}, \qquad [a_1, \ldots, a_n] : \mathsf{Seq}\{A\},$$

[2]More precisely, we should represent cardinalities as monads and queries as their Kleisli arrows [18].

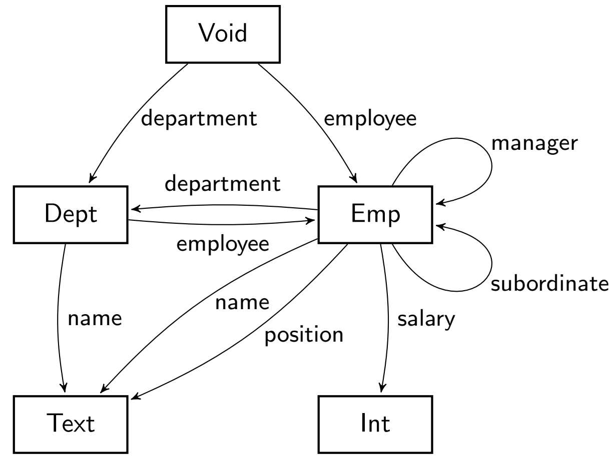

Figure 2: Database schema in folded form

are defined by

$$\begin{aligned} \perp\ &: \mathsf{Opt}\{A\} \longmapsto [\,] : \mathsf{Seq}\{A\}, \\ a : A \longmapsto \ulcorner a \urcorner &: \mathsf{Opt}\{A\} \longmapsto [a] : \mathsf{Seq}\{A\}. \end{aligned}$$

This order lets us, whenever necessary, promote any query $A \to M\{B\}$ to a query $A \to M'\{B\}$ with a greater cardinality $M' \sqsupseteq M$.

Monadic composition for the option and sequence containers is well known. For optional queries

$$p : A \to \mathsf{Opt}\{B\}, \qquad q : B \to \mathsf{Opt}\{C\},$$

it is defined by

$$\begin{aligned} &p\,.\,q : A \to \mathsf{Opt}\{C\}, \\ &p\,.\,q : a \mapsto \begin{cases} \ulcorner c \urcorner & (p(a) = \ulcorner b \urcorner,\ q(b) = \ulcorner c \urcorner), \\ \perp & (\text{otherwise}). \end{cases} \end{aligned}$$

For plural queries

$$p : A \to \mathsf{Seq}\{B\}, \qquad q : B \to \mathsf{Seq}\{C\},$$

the sequence $(p\,.\,q)(a)$ is calculated by applying p to a

$$a \xmapsto{p} [b_1, b_2, \ldots],$$

then applying q to every element of $p(a)$

$$[b_1, b_2, \ldots] \xmapsto{[q]} [[c_1^1, c_1^2, \ldots], [c_2^1, c_2^2, \ldots], \ldots],$$

and finally merging the nested sequences

$$[[c_1^1, c_1^2, \ldots], [c_2^1, c_2^2, \ldots], \ldots] \xmapsto{\cancel{[\,]}} [c_1^1, c_1^2, \ldots, c_2^1, c_2^2, \ldots].$$

We are now ready to present the design of a combinator-based query language.

Query model: A database query is characterized by its input type A, its output type B and its cardinality M, and can be represented as a function of the form

$$p : A \to M\{B\},$$

where $M\{B\}$ is one of B, $\mathsf{Opt}\{B\}$ or $\mathsf{Seq}\{B\}$; the respective queries are called singular, optional or plural.

Primitives: The set of primitives includes classes

$$\begin{aligned} &\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}, \\ &\mathsf{employee} \quad : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}, \end{aligned}$$

attributes

$$\begin{aligned} &\mathsf{name} \quad : \mathsf{Dept} \to \mathsf{Text}, && \mathsf{name} : \mathsf{Emp} \to \mathsf{Text}, \\ &\mathsf{position} : \mathsf{Emp} \to \mathsf{Text}, && \mathsf{salary} : \mathsf{Emp} \to \mathsf{Int}, \end{aligned}$$

and relationships

$$\begin{aligned} &\mathsf{department} : \mathsf{Emp} \to \mathsf{Dept}, \\ &\mathsf{employee} \quad : \mathsf{Dept} \to \mathsf{Seq}\{\mathsf{Emp}\}, \\ &\mathsf{manager} \quad : \mathsf{Emp} \to \mathsf{Opt}\{\mathsf{Emp}\}, \\ &\mathsf{subordinate} : \mathsf{Emp} \to \mathsf{Seq}\{\mathsf{Emp}\}. \end{aligned}$$

Combinators: The composition combinator sends two queries

$$p : A \to M_1\{B\}, \qquad q : B \to M_2\{C\}$$

to their composition

$$p \,.\, q : A \to M\{C\} \qquad (M = M_1 \sqcup M_2).$$

Other common combinators are listed in Table 1. They are described in the following sections.

Recall that we started with the schema graph in Figure 1, which gave us the original, incomplete set of primitives. To reflect the remaining primitives, we should add the Void node and the missing arcs (see Figure 2). Furthermore, we will transform the schema graph into an (infinite) tree by unfolding it starting from the Void node (see Figure 3). The unfolded tree represents the functional database in a universal hierarchical form.

3 Query Combinators

In this section, we show how the query model defined in Section 2 can support a wide range of operations on data.

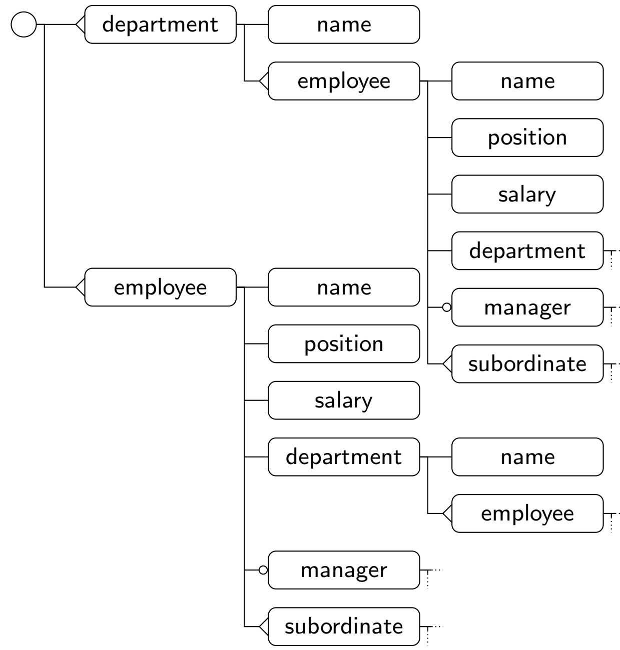

Figure 3: Database schema in unfolded form

Extracting Data

By traversing the tree of Figure 3, we can extract data from the database.

Example 3.1 *Show the name of each department.*

$$\mathsf{department.name}$$

This example is constructed by descending through nodes $\mathsf{department}$ and name, which represent primitives

$$\begin{aligned} &\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}, \\ &\mathsf{name} \qquad : \mathsf{Dept} \to \mathsf{Text}. \end{aligned}$$

The composition of the primitives inherits the input of the first component and the output of the second component. Since one of the components is plural, the composition is also plural, which gives it a signature

$$\mathsf{department.name} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Text}\}.$$

Example 3.2 *For each department, show the name of each employee.*

$$\mathsf{department.employee.name}$$

This example takes a path through

$$\begin{aligned} &\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}, \\ &\mathsf{employee} \quad : \mathsf{Dept} \to \mathsf{Seq}\{\mathsf{Emp}\}, \\ &\mathsf{name} \qquad : \mathsf{Emp} \to \mathsf{Text}, \end{aligned}$$

to construct a query

$$\mathsf{department.employee.name} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Text}\}.$$

This query produces a list of employee names. Since each employee belongs to exactly one department, the list should contain the name of every employee. The order in which the names appear in the output depends on the intrinsic order of the department and employee primitives, but, in any case, employees within the same department will be coupled together.

The same collection of names, although not necessarily in the same order, is produced by the following example.

Example 3.3 *Show the name of each employee.*

employee.name

On the other hand, the next example is very different from the apparently similar Example 3.1.

Example 3.4 *For each employee, show the name of their department.*

employee.department.name

Here, we should see a list of department names, but each name will appear as many times as there are employees in the corresponding department.

Example 3.5 *Show the position of each employee.*

employee.position

Similarly, employee.position will output duplicate position titles. We will see how to produce a list of *unique* positions in Section 4.

Example 3.6 *Show all employees.*

employee

This example emits a sequence of employee entities, which, in practice, could be represented as records with employee attributes.

Summarizing Data

Let us show how the extracted data can be summarized.

Example 3.7 *Show the number of departments.*

count(department)

Identity and constants

$$\begin{aligned}
&\mathsf{here} &&: A \to A &&= a \mapsto a\\
&150000 &&: A \to \mathsf{Int} &&= a \mapsto 150000\\
&\mathsf{home} &&: A \to \mathsf{Void} &&= a \mapsto \top\\
&\mathsf{null} &&: A \to \mathsf{Opt}\{B\} &&= a \mapsto \bot
\end{aligned}$$

Some scalar combinators

$$\begin{aligned}
&=, \neq &&: (A \to B, A \to B) &&\to (A \to \mathsf{Bool})\\
&<, \leq, >, \geq &&: (A \to B, A \to B) &&\to (A \to \mathsf{Bool})\\
&\&, | &&: (A \to \mathsf{Bool}, A \to \mathsf{Bool}) &&\to (A \to \mathsf{Bool})\\
&+, - &&: (A \to \mathsf{Int}, A \to \mathsf{Int}) &&\to (A \to \mathsf{Int})\\
&\mathsf{length} &&: (A \to \mathsf{Text}) &&\to (A \to \mathsf{Int})
\end{aligned}$$

Aggregate combinators

$$\begin{aligned}
&\mathsf{count} &&: (A \to \mathsf{Seq}\{B\}) &&\to (A \to \mathsf{Int})\\
&\mathsf{exists} &&: (A \to \mathsf{Seq}\{B\}) &&\to (A \to \mathsf{Bool})\\
&\mathsf{any}, \mathsf{all} &&: (A \to \mathsf{Seq}\{\mathsf{Bool}\}) &&\to (A \to \mathsf{Bool})\\
&\mathsf{sum} &&: (A \to \mathsf{Seq}\{\mathsf{Int}\}) &&\to (A \to \mathsf{Int})\\
&\mathsf{max}, \mathsf{min} &&: (A \to \mathsf{Seq}\{\mathsf{Int}\}) &&\to (A \to \mathsf{Opt}\{\mathsf{Int}\})
\end{aligned}$$

Sequence transformers

$$\begin{aligned}
&\mathsf{filter} : (A \to \mathsf{Seq}\{B\}, B \to \mathsf{Bool}) &&\to (A \to \mathsf{Seq}\{B\})\\
&\mathsf{sort} : (A \to \mathsf{Seq}\{B\},\\
&\qquad B \to C_1, \ldots, B \to C_n) &&\to (A \to \mathsf{Seq}\{B\})\\
&\mathsf{take} : (A \to \mathsf{Seq}\{B\}, A \to \mathsf{Int}) &&\to (A \to \mathsf{Seq}\{B\})\\
&\mathsf{unique} : (A \to \mathsf{Seq}\{B\}) &&\to (A \to \mathsf{Seq}\{B\})
\end{aligned}$$

Selector and modifiers

$$\begin{aligned}
&\mathsf{select} : (A \to M\{B\},\\
&\qquad B \to M_1\{C_1\}, \ldots, B \to M_n\{C_n\})\\
&\quad \to (A \to M\{\langle M_1\{C_1\}, \ldots, M_n\{C_n\}\rangle\})\\
&\mathsf{define} : (A \to M\{B\}, B \to T) \to (A \to M\{B\})\\
&\mathsf{asc}, \mathsf{desc} : (A \to B) \to (A \to B_{\lessgtr})
\end{aligned}$$

Hierarchical connector

$$\mathsf{connect} : (A \to \mathsf{Opt}\{A\}) \to (A \to \mathsf{Seq}\{A\})$$

Grouping

$$\begin{aligned}
&\mathsf{group} : (A \to \mathsf{Seq}\{B\}, B \to C_1, \ldots, B \to C_n)\\
&\quad \to (A \to \mathsf{Seq}\{\langle C_1, \ldots, C_n, \mathsf{Seq}\{B\}\rangle\})\\
&\mathsf{rollup} : (A \to \mathsf{Seq}\{B\}, B \to C_1, \ldots, B \to C_n)\\
&\quad \to (A \to \mathsf{Seq}\{\langle \mathsf{Opt}\{C_1\}, \ldots, \mathsf{Opt}\{C_n\}, \mathsf{Seq}\{B\}\rangle\})
\end{aligned}$$

Context primitives and combinators

$$\begin{aligned}
&\mathsf{frame} : (\mathsf{Rel}\{A\} \to M\{B\}) \to (A \to M\{B\})\\
&\mathsf{before}, \mathsf{around} : \mathsf{Rel}\{A\} \to \mathsf{Seq}\{A\}\\
&\mathsf{given} : (\mathsf{Env}_T\{A\} \to M\{B\}, A \to T) \to (A \to M\{B\})\\
&\mathit{PARAM} : \mathsf{Env}_T\{A\} \to T
\end{aligned}$$

Table 1: Some primitives and combinators

This query produces a single number, so that its signature is

$$\mathsf{count}(\mathsf{department}) : \mathsf{Void} \to \mathsf{Int}.$$

It is constructed by applying the count combinator to a query that generates *a list of all departments*

$$\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}.$$

Comparing the signatures of these two queries, we can derive the signature of the count combinator, in this specific case

$$(\mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}) \to (\mathsf{Void} \to \mathsf{Int}),$$

and, in general

$$\mathsf{count} : (A \to \mathsf{Seq}\{B\}) \to (A \to \mathsf{Int}).$$

In other words, the count combinator transforms any sequence-valued query to an integer-valued query. It is implemented by lifting the function that computes the length of a sequence

$$| - | : \mathsf{Seq}\{A\} \to \mathsf{Int}$$

to a query combinator

$$\mathsf{count}(q) = a \mapsto |q(a)|.$$

We call count and other unary combinators that transform a plural query to a singular (or optional) query *aggregate* combinators.

The next example follows the same pattern.

Example 3.8 *Show the highest salary of all employees.*

$$\mathsf{max}(\mathsf{employee.salary})$$

It extracts the relevant data with

$$\mathsf{employee.salary} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Int}\}$$

and summarizes it using the max aggregate

$$\mathsf{max}(\mathsf{employee.salary}) : \mathsf{Void} \to \mathsf{Opt}\{\mathsf{Int}\}.$$

This query is optional since it produces no output when the database contains no employees.

Example 3.9 *Show the number of employees in each department.*

$$\mathsf{department.count}(\mathsf{employee})$$

In this example, we transform a plural relationship, *all employees in the given department*

$$\mathsf{employee} : \mathsf{Dept} \to \mathsf{Seq}\{\mathsf{Emp}\}$$

to a calculated attribute, *the number of employees in the given department*

$$\mathsf{count}(\mathsf{employee}) : \mathsf{Dept} \to \mathsf{Int}.$$

Then we attach it to

$$\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}$$

to get *the number of employees in each department*

$$\mathsf{department.count}(\mathsf{employee}) : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Int}\}.$$

Using the combinator max, we can further collapse this list to a single number, as shown in the next example.

Example 3.10 *Show the size of the largest department.*

$$\mathsf{max}(\mathsf{department.count}(\mathsf{employee}))$$

Pipeline Notation

Queries are often constructed incrementally, by extracting relevant data and then shaping it into the desired form with a chain of combinators. This construction is made apparent with the *pipeline notation*.

In pipeline notation, the first argument of a combinator is placed in front of it, separated by colon ("**:**"):

$$p\mathbf{:}F \equiv F(p), \qquad p\mathbf{:}F(q_1, \ldots, q_n) \equiv F(p, q_1, \ldots, q_n).$$

For example, count(department) could also be written

$$\mathsf{department}\mathbf{:}\mathsf{count}.$$

A more sophisticated query written in pipeline notation is shown in the following example.

Example 3.11 *Show the top 10 highest paid employees in the Police department.*

```
employee
:filter(department.name = "POLICE")
:sort(salary:desc)
:select(name, position, salary)
:take(10)
```

Without pipeline notation, this query is much less intelligible:

```
take(select(sort(filter(
    employee, department.name = "POLICE"),
        desc(salary)), name, position, salary), 10).
```

Combinators filter, sort, select and take used in this query are described below.

Filtering Data

We can now demonstrate how to produce entities that satisfy a certain condition.

Example 3.12 *Which employees have a salary higher than \$150k?*

$$\mathsf{employee}\,\mathbf{:}\,\mathsf{filter}(\mathsf{salary} > 150000)$$

This query introduces several concepts.

First, the integer literal 150000 represents a primitive query that *for any given employee, produces the number* 150000

$$150000 : \mathsf{Emp} \to \mathsf{Int} = e \mapsto 150000.$$

Second, the relational symbol $>$ denotes a binary combinator, which builds a query *for a given employee, show whether their salary is higher than \$150000*

$$\mathsf{salary} > 150000 : \mathsf{Emp} \to \mathsf{Bool}.$$

The combinator

$$- > - : (A \to \mathsf{Int},\ A \to \mathsf{Int}) \to (A \to \mathsf{Bool})$$

is implemented by lifting the relational operator

$$- > - : (\mathsf{Int},\ \mathsf{Int}) \to \mathsf{Bool}$$

to an operation on queries

$$(p > q) = a \mapsto (p(a) > q(a)).$$

Third, a binary combinator filter emits those employee entities that satisfy the condition salary > 150000. In general, given

$$p : A \to \mathsf{Seq}\{B\}, \qquad q : B \to \mathsf{Bool},$$

a query

$$\mathsf{filter}(p,\ q) : A \to \mathsf{Seq}\{B\}$$

produces the values of p that satisfy condition q

$$\mathsf{filter}(p,\ q) = a \mapsto [\, b \mid b \leftarrow p(a),\ q(b) = \mathsf{true}\,].$$

The following example shows how filter could be used with other combinators.

Example 3.13 *How many departments have more than 1000 employees?*

$$\begin{aligned}&\mathsf{department}\\&\mathbf{:}\mathsf{filter}(\mathsf{count}(\mathsf{employee}) > 1000)\\&\mathbf{:}\mathsf{count}\end{aligned}$$

Sorting and Paginating Data

The combinator sort, applied to a plural query, sorts the query output in ascending order.

Example 3.14 *Show the names of all departments in alphabetical order.*

$$\mathsf{sort}(\mathsf{department.name})$$

The combinator sort is implemented by lifting a sequence function

$$\mathsf{sort} : \mathsf{Seq}\{A\} \to \mathsf{Seq}\{A\}$$

to a query combinator

$$\begin{aligned}&\mathsf{sort} : (A \to \mathsf{Seq}\{B\}) \to (A \to \mathsf{Seq}\{B\}),\\&\mathsf{sort}(p) = a \mapsto \mathsf{sort}(p(a)).\end{aligned}$$

Example 3.15 *Show all employees ordered by salary.*

$$\mathsf{employee}\,\mathbf{:}\,\mathsf{sort}(\mathsf{salary})$$

In this example, a list of employees is sorted by the value of the attribute salary, which is supplied as a second argument to the sort combinator. In this form, sort has a signature

$$\mathsf{sort} : (A \to \mathsf{Seq}\{B\},\ B \to C) \to (A \to \mathsf{Seq}\{B\}).$$

Example 3.16 *Show all employees ordered by salary, highest paid first.*

$$\mathsf{employee}\,\mathbf{:}\,\mathsf{sort}(\mathsf{salary}\,\mathbf{:}\,\mathsf{desc})$$

Here, the sort key is wrapped with the combinator desc to indicate the descending sort order.

It is not immediately obvious how to implement desc without violating the query model. Naïvely, desc acts like a negation operator, however, not every type supports negation. Instead, we make the sort order a part of the type definition, so that

$$\mathsf{Int}_{\le} \quad \text{and} \quad \mathsf{Int}_{\ge}$$

could indicate the integer type with ascending and descending sort order respectively. Then, desc could operate by switching the sort order of the output type and be given a signature

$$\mathsf{desc} : (A \to B) \to (A \to B_{\ge}).$$

Example 3.17 *Who are the top 1% of highest paid employees?*

$$\begin{aligned}&\mathsf{employee}\\&\mathbf{:}\mathsf{sort}(\mathsf{salary}\,\mathbf{:}\,\mathsf{desc})\\&\mathbf{:}\mathsf{take}(\mathsf{count}(\mathsf{employee}) \div 100)\end{aligned}$$

In this example, only the first 1% of employees are retained by the combinator take, which has two arguments: a query that produces a sequence of employees

$$\mathsf{employee{:}sort(salary{:}desc)} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}$$

and a query that returns how many employees to take

$$\mathsf{count(employee)} \div 100 : \mathsf{Void} \to \mathsf{Int}.$$

Notice that both arguments of take have the same input (Void in this case), which is reflected in the signature

$$\mathsf{take} : (A \to \mathsf{Seq}\{B\},\ A \to \mathsf{Int}) \to (A \to \mathsf{Seq}\{B\}).$$

Query Output

The combinator select customizes the query output.

Previously, we constructed a query to *show the number of employees in each department* (see Example 3.9):

$$\mathsf{department.count(employee)}.$$

However, this query only produces a list of bare numbers—it does not connect them to their respective departments. This is corrected in the following example.

Example 3.18 *For every department, show its name and the number of employees.*

$$\mathsf{department{:}select(name,\ size} \Rightarrow \mathsf{count(employee))}$$

In this example, the combinator select takes three arguments: the base query

$$\mathsf{department} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Dept}\}$$

and two field queries

$$\begin{aligned} &\mathsf{name} &&: \mathsf{Dept} \to \mathsf{Text},\\ &\mathsf{count(employee)} &&: \mathsf{Dept} \to \mathsf{Int}. \end{aligned}$$

The select combinator generates a sequence of records by applying each field query to every entity produced by the base query, giving this example a signature

$$\mathsf{Void} \to \mathsf{Seq}\{\langle \mathsf{name} : \mathsf{Text},\ \mathsf{size} : \mathsf{Int}\rangle\}.$$

The declaration

$$\langle \mathsf{name} : \mathsf{Text},\ \mathsf{size} : \mathsf{Int}\rangle$$

defines a *record* type with two fields: a text field name and an integer field size. The names of the record fields are derived from the tags of the field queries, which could be set using the *tagging notation*. For example,

$$\mathsf{size} \Rightarrow \mathsf{count(employee)}$$

binds a tag size to the query count(employee). Since the tag does not materially affect the query it annotates, we do not expose the tag in the query model.

A more complex output structure could be defined with nested select combinators.

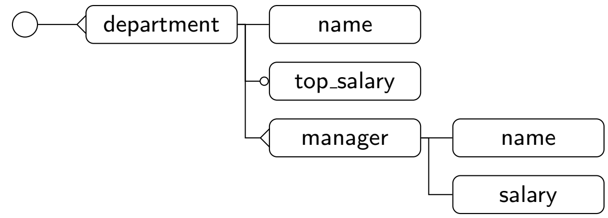

Figure 4: Output database for Example 3.19

Example 3.19 *For every department, show the top salary and a list of managers with their salaries.*

$$\begin{aligned} &\mathsf{department}\\ &\mathsf{{:}select(name,}\\ &\qquad \mathsf{top\_salary} \Rightarrow \mathsf{max(employee.salary)},\\ &\qquad \mathsf{manager} \Rightarrow \mathsf{employee}\\ &\qquad\qquad \mathsf{{:}filter(exists(subordinate))}\\ &\qquad\qquad \mathsf{{:}select(name,\ salary))} \end{aligned}$$

In this example, the query output has the type

$$\begin{aligned} &\mathsf{Seq}\{\langle \mathsf{name} : \mathsf{Text},\ \mathsf{top\_salary} : \mathsf{Opt}\{\mathsf{Int}\},\\ &\quad \mathsf{manager} : \mathsf{Seq}\{\langle \mathsf{name} : \mathsf{Text},\ \mathsf{salary} : \mathsf{Int}\rangle\}\rangle\}. \end{aligned}$$

Recall that we represented the data source in a universal hierarchical form (see Figure 3). Furthermore, the query output could also be represented as a hierarchical database, whose structure is determined by the query signature (see Figure 4). Thus, queries could be seen as transformations of hierarchical databases.

Query Aliases

A complex query could often be simplified by replacing duplicate expressions with aliases.

Example 3.20 *Show the top 3 largest departments and their sizes.*

$$\begin{aligned} &\mathsf{department}\\ &\mathsf{{:}define(size} \Rightarrow \mathsf{count(employee))}\\ &\mathsf{{:}sort(size{:}desc)}\\ &\mathsf{{:}select(name,\ size)}\\ &\mathsf{{:}take(3)} \end{aligned}$$

In this example, the alias size is created in two steps: first, the query

$$\mathsf{count(employee)} : \mathsf{Dept} \to \mathsf{Int}$$

is tagged with the name size, and then size is added to scope of Dept by the combinator define.

Although this query could have been written as

$$\begin{aligned}&\textsf{department}\\&\textsf{:sort(count(employee):desc)}\\&\textsf{:select(name, count(employee))}\\&\textsf{:take}(3),\end{aligned}$$

the use of an alias makes this example more legible, not only by reducing redundancy, but also by assigning a name to a key concept of the query.

Hierarchical Relationships

Hierarchical relationships are encoded by self-referential primitives.

For example, the relationship between an employee and their manager is expressed with

$$\textsf{manager} : \textsf{Emp} \to \textsf{Opt}\{\textsf{Emp}\}.$$

Example 3.21 *Find all employees whose salary is higher than the salary of their manager.*

$$\textsf{employee:filter(salary} > \textsf{manager.salary)}$$

This example uses familiar combinators filter and $>$ (see Example 3.12), but an alert reader will notice the disagreement between the signature of the combinator

$$- > - : (A \to \textsf{Int},\ A \to \textsf{Int}) \to (A \to \textsf{Bool})$$

and the signatures of its arguments

$$\begin{aligned}&\textsf{salary} &&: \textsf{Emp} \to \textsf{Int},\\&\textsf{manager.salary} &&: \textsf{Emp} \to \textsf{Opt}\{\textsf{Int}\}.\end{aligned}$$

Namely, $>$ expects its arguments to be singular, but the output of manager.salary is optional.

To legitimize this query, we adopt the following rule. When one argument of a scalar combinator has a non-trivial cardinality, this cardinality can be promoted to the output of the combinator. This rule gives $>$ a signature

$$- > - : (A \to \textsf{Int},\ A \to M\{\textsf{Int}\}) \to (A \to M\{\textsf{Bool}\})$$

or, in this specific case,

$$\textsf{salary} > \textsf{manager.salary} : \textsf{Emp} \to \textsf{Opt}\{\textsf{Bool}\}.$$

Finally, we need to let filter accept predicate queries with optional output, by treating $\perp$ as false.

Using expressions

$$\begin{aligned}&\textsf{manager},\\&\textsf{manager.manager},\\&\textsf{manager.manager.manager},\ \ldots\end{aligned}$$

we can build queries that involve the manager, the manager's manager, etc. We can also obtain *the complete chain of command for the given employee* with

$$\textsf{connect(manager)} : \textsf{Emp} \to \textsf{Seq}\{\textsf{Emp}\}.$$

Example 3.22 *Find all direct and indirect subordinates of the City Treasurer.*

$$\begin{aligned}&\textsf{employee}\\&\textsf{:filter(any(connect(manager).position} =\\&\qquad\qquad\qquad\qquad\text{“}\texttt{CITY TREASURER}\text{”}))\end{aligned}$$

Here, the query

$$\textsf{connect(manager).position} : \textsf{Emp} \to \textsf{Seq}\{\textsf{Text}\}$$

produces *the positions of all managers above the given employee.*

In general, the combinator connect maps an optional self-referential query to a plural self-referential query by taking its transitive closure:

$$\begin{aligned}&\textsf{connect} : (A \to \textsf{Opt}\{A\}) \to (A \to \textsf{Seq}\{A\}),\\&\textsf{connect}(p) = a \mapsto [\ p(a),\ p(p(a)),\ \ldots,\ p^{(n)}(a)\ ]\\&\qquad\qquad (p^{(n)}(a) \neq \perp,\ p^{(n+1)}(a) = \perp).\end{aligned}$$

4 Quotient Classes

Previously, we demonstrated how to group and aggregate data—so long as the structure of the data reflects the hierarchical form of the database. In this section, we show how to overcome this limitation.

In Figure 3, the schema graph is unfolded into an infinite tree, shaping the data into a hierarchical form. A section of this hierarchy could be extracted using the select combinator.

Example 4.1 *Show all departments, and, for each department, list the associated employees.*

$$\textsf{department:select(name, employee)}$$

But what if we ask for *positions* instead of *departments*?

Example 4.2 *Show all positions, and, for each position, list the associated employees.*

Unlike the previous example, this query does not match the structure of the database and, therefore, cannot be constructed as easily. Indeed, Example 4.1 is built from the primitives

$$\begin{aligned}&\textsf{department} &&: \textsf{Void} \to \textsf{Seq}\{\textsf{Dept}\},\\&\textsf{name} &&: \textsf{Dept} \to \textsf{Text},\\&\textsf{employee} &&: \textsf{Dept} \to \textsf{Seq}\{\textsf{Emp}\}.\end{aligned}$$

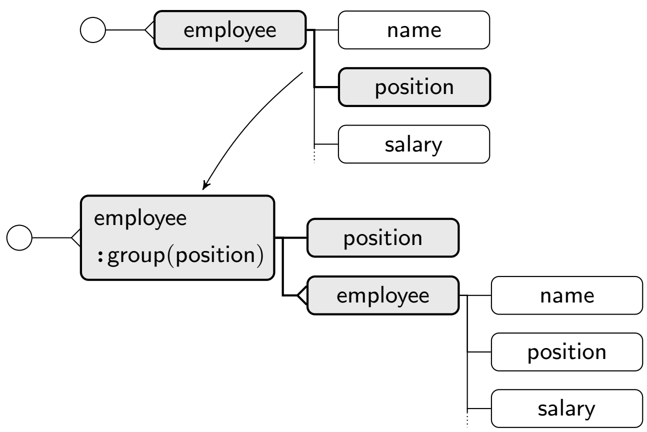

Figure 5: Action of the group combinator

To construct Example 4.2 in a similar fashion, we need a hypothetical class Pos of *position* entities and a set of queries with the corresponding signatures

$$\begin{aligned} &\mathsf{Void} \to \mathsf{Seq}\{\mathsf{Pos}\}, \\ &\mathsf{Pos} \to \mathsf{Text}, \\ &\mathsf{Pos} \to \mathsf{Seq}\{\mathsf{Emp}\}. \end{aligned} \qquad (\star\star\star)$$

However, there is no built-in class of position entities and we only have the following primitives available:

$$\begin{aligned} &\mathsf{employee} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}, \\ &\mathsf{position} \ : \mathsf{Emp} \to \mathsf{Text}. \end{aligned}$$

To make a "virtual" entity class from all distinct values of an attribute and inject this class into the database structure, we use the group combinator. For example (see Figure 5), a list of *all distinct employee positions* can be produced with the query

$$\mathsf{employee{:}group(position)} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Pos}\}.$$

The virtual Pos class comes with the primitives

$$\begin{aligned} &\mathsf{position} \ : \mathsf{Pos} \to \mathsf{Text}, \\ &\mathsf{employee} : \mathsf{Pos} \to \mathsf{Seq}\{\mathsf{Emp}\}, \end{aligned}$$

which, *given a position entity, produce respectively the position name and a list of associated employees.* This gives us all the query components (see $(\star\star\star)$ above) needed to complete the example.

Example 4.2 *Show all positions, and, for each position, list the associated employees.*

```
employee
:group(position)
:select(position, employee)
```

The query

$$\mathsf{employee{:}group(position)}$$

correlates all distinct values emitted by position with the respective employee entities and packs them together into the records of type

$$\mathsf{Pos} \equiv \langle \mathsf{position} : \mathsf{Text},\ \mathsf{employee} : \mathsf{Seq}\{\mathsf{Emp}\} \rangle.$$

We call Pos a *quotient class* and denote it by

$$\mathsf{Emp}\big/_{\mathsf{position}}.$$

Once the database hierarchy is rearranged to include the class Pos, we can answer any questions about position entities.

Example 4.3 *In the Police department, show all positions with the number of employees and the top salary.*

```
employee
:filter(department.name = "POLICE")
:group(position)
:select(position,
        count(employee),
        max(employee.salary))
```

Here, for each position in the Police department, we determine two calculated attributes, the number of employees and the top salary:

$$\begin{aligned} &\mathsf{count(employee)} &&: \mathsf{Emp}\big/_{\mathsf{position}} \to \mathsf{Int}, \\ &\mathsf{max(employee.salary)} &&: \mathsf{Emp}\big/_{\mathsf{position}} \to \mathsf{Opt}\{\mathsf{Int}\}. \end{aligned}$$

Example 4.4 *Arrange employees into a hierarchy: first by position, then by department.*

```
employee
:group(position)
:select(position,
        employee
        :group(department)
        :select(department.name, employee))
```

Nested group combinators can construct a hierarchical output of an arbitrary form. In this example, we rebuild the database hierarchy to place positions on top, then departments, and then employees. Notably, the nested group expression has a signature

$$\mathsf{employee{:}group(department)} : \\ \mathsf{Emp}\big/_{\mathsf{position}} \to \mathsf{Seq}\Big\{\mathsf{Emp}\big/_{\mathsf{department}}\Big\}.$$

Example 4.5 *Show all positions available in more than one department, and, for each position, list the respective departments.*

$$\begin{aligned}&\mathsf{employee}\\&\mathsf{:group(position)}\\&\mathsf{:define(department} \Rightarrow\\&\qquad\qquad \mathsf{unique(employee.department))}\\&\mathsf{:filter(count(department)} > 1)\\&\mathsf{:select(position,\ department.name)}\end{aligned}$$

This example uses the unique combinator to find all distinct values in a list of departments. The unique combinator can be expressed via group by forgetting the plural component of the quotient class. In this example, unique(employee.department) is equivalent to

$$\mathsf{employee{:}group(department).department}.$$

Example 4.6 *How many employees at each level of the organization chart?*

$$\begin{aligned}&\mathsf{employee}\\&\mathsf{:group(level} \Rightarrow \mathsf{count(connect(manager)))}\\&\mathsf{:select(level,\ count(employee))}\end{aligned}$$

This example demonstrates how group could be applied to a calculated attribute such as

$$\mathsf{count(connect(manager))} : \mathsf{Emp} \to \mathsf{Int}.$$

The group combinator could also be applied to more than one attribute, which can be used for summarizing data along several dimensions. When the summary data has to include subtotals and totals, we replace group with rollup.

Example 4.7 *Show the average salary by department and position, with subtotals for each department and the grand total.*

$$\begin{aligned}&\mathsf{employee}\\&\mathsf{:rollup(department,\ position)}\\&\mathsf{:select(department,}\\&\qquad\quad \mathsf{position,}\\&\qquad\quad \mathsf{mean(employee.salary))}\end{aligned}$$

In this example, the query

$$\mathsf{employee{:}rollup(department,\ position)}$$

produces a sequence of records of type

$$\begin{aligned}\mathsf{Emp}\big/_{\mathsf{department}_\perp,\,\mathsf{position}_\perp} \equiv&\\ &\langle \mathsf{department} : \mathsf{Opt}\{\mathsf{Dept}\},\\ &\ \ \mathsf{position} \quad\ \, : \mathsf{Opt}\{\mathsf{Text}\},\\ &\ \ \mathsf{employee} \ \ : \mathsf{Seq}\{\mathsf{Emp}\}\rangle.\end{aligned}$$

In addition to the records that would be generated by group, rollup emits one "subtotal" record per each department and one "grand total" record. The former has the position field set to $\perp$ and an employee list containing all employees in the given department. The latter has both department and position set to $\perp$ and employee containing the full list of employees.

5 Query Context

In this section, we extend the query model to support *context-aware* queries: parameterized queries and queries with window functions.

Example 5.1 *Show all employees in the given department* D *with the salary higher than* S*, where*

$$\mathsf{D} = \text{“}\mathtt{POLICE}\text{”}, \quad \mathsf{S} = 150000.$$

$$\begin{aligned}&\mathsf{employee}\\&\mathsf{:filter(department.name} = \mathsf{D}\ \&\ \mathsf{salary} > \mathsf{S})\\&\mathsf{:given(D} \Rightarrow \text{“}\mathtt{POLICE}\text{”},\ \mathsf{S} \Rightarrow 150000)\end{aligned}$$

Practical database queries often depend upon *query parameters*, which collectively form the query *environment*. The environment is represented by a container, such as

$$\mathsf{Env}_{\mathsf{D:Text},\mathsf{S:Int}}\{A\} \equiv \langle A,\ \langle \mathsf{D} : \mathsf{Text},\ \mathsf{S} : \mathsf{Int}\rangle\rangle,$$

that encapsulates both the regular input value and the values of the parameters. The parameters can be extracted from the environment with the primitives

$$\mathsf{D} : \mathsf{Env}_{\mathsf{D:Text}}\{A\} \to \mathsf{Text}, \qquad \mathsf{S} : \mathsf{Env}_{\mathsf{S:Int}}\{A\} \to \mathsf{Int}.$$

The query environment is populated using the combinator given. In this example, the first argument of given is a parameterized query

$$\begin{aligned}&\mathsf{employee}\\&\mathsf{:filter(department.name} = \mathsf{D}\ \&\ \mathsf{salary} > \mathsf{S}) :\\&\qquad\qquad\qquad \mathsf{Env}_{\mathsf{D:Text},\mathsf{S:Int}}\{\mathsf{Void}\} \to \mathsf{Seq}\{\mathsf{Emp}\}.\end{aligned}$$

The other two arguments are the constant queries

$$\text{“}\mathtt{POLICE}\text{”} : \mathsf{Void} \to \mathsf{Text}, \qquad 150000 : \mathsf{Void} \to \mathsf{Int}$$

that specify the values of the parameters. The combined query does *not* depend upon the parameters, and, hence, has a signature

$$\mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}.$$

In general, given takes a parameterized query

$$p : \mathsf{Env}_{x_1:T_1,\ldots,x_n:T_n}\{A\} \to M\{B\},$$

n queries that evaluate the parameters

$$q_1 : A \to T_1, \quad \ldots, \quad q_n : A \to T_n$$

and combines them into a context-free query

$$\begin{aligned}&\mathsf{given}(p,\ q_1,\ \ldots,\ q_n) : A \to M\{B\},\\ &\mathsf{given}(p,\ q_1,\ \ldots,\ q_n) = a \mapsto p(\langle a, \langle q_1(a), \ldots, q_n(a)\rangle\rangle).\end{aligned}$$

Example 5.2 *Which employees have higher than average salary?*

$$\begin{aligned}&\mathsf{employee}\\ &\mathbf{:}\mathsf{filter}(\mathsf{salary} > \mathsf{MS})\\ &\mathbf{:}\mathsf{given}(\mathsf{MS} \Rightarrow \mathsf{mean}(\mathsf{employee}\mathbf{.}\mathsf{salary}))\end{aligned}$$

This example uses the query environment to pass information between different scopes. The parameter MS is calculated in the scope of Void by the query

$$\mathsf{mean}(\mathsf{employee}\mathbf{.}\mathsf{salary}) : \mathsf{Void} \to \mathsf{Opt}\{\mathsf{Num}\}$$

and is extracted in the scope of Emp by the primitive

$$\mathsf{MS} : \mathsf{Env}_{\mathsf{MS}:\mathsf{Opt}\{\mathsf{Num}\}}\{\mathsf{Emp}\} \to \mathsf{Opt}\{\mathsf{Num}\}.$$

The query environment is one example of a *query context*, a comonadic container wrapping the query input. It could be shown that the environment is compatible with query composition (cf. Section 2), which permits us to incorporate it into the query model.

Another example of a query context is the *input flow*, a container of every input value seen by the query. We denote this context type by $\mathsf{Rel}\{A\}$ and its values by

$$[a_1,\ \ldots, (\!(a_j)\!),\ \ldots,\ a_n] : \mathsf{Rel}\{A\},$$

where a_j is the current input value, $a_1,\ \ldots,\ a_{j-1}$ are the values seen in the past, and $a_{j+1},\ \ldots,\ a_n$ are the values to be seen in the future. The input flow can be used for an alternative implementation of Example 5.2.

Example 5.2′ *Which employees have higher than average salary?*

$$\mathsf{employee}\mathbf{:}\mathsf{filter}(\mathsf{salary} > \mathsf{mean}(\mathsf{around}\mathbf{.}\mathsf{salary}))$$

When a query needs to relate each value in a dataset to the dataset as a whole, we can use the plural primitive around, which materializes its input flow as a sequence:

$$\begin{aligned}&\mathsf{around} : \mathsf{Rel}\{A\} \to \mathsf{Seq}\{A\}\\ &\mathsf{around} = [a_1,\ \ldots, (\!(a_j)\!),\ \ldots,\ a_n]\\ &\qquad\qquad \mapsto [a_1,\ \ldots,\ a_j,\ \ldots,\ a_n].\end{aligned}$$

In this example, around produces, *for a given employee, a list of all employees.* By composing it with salary, we get, *for a given employee, a list of all salaries*

$$\mathsf{around}\mathbf{.}\mathsf{salary} : \mathsf{Rel}\{\mathsf{Emp}\} \to \mathsf{Seq}\{\mathsf{Int}\},$$

which lets us determine the average salary

$$\mathsf{mean}(\mathsf{around}\mathbf{.}\mathsf{salary}) : \mathsf{Rel}\{\mathsf{Emp}\} \to \mathsf{Opt}\{\mathsf{Num}\}$$

without ever leaving the current scope.

Example 5.3 *In the Police department, show employees whose salary is higher than the average for their position.*

$$\begin{aligned}&\mathsf{employee}\\ &\mathbf{:}\mathsf{filter}(\mathsf{department}\mathbf{.}\mathsf{name} = \text{“}\texttt{POLICE}\text{”})\\ &\mathbf{:}\mathsf{filter}(\mathsf{salary} > \mathsf{mean}(\mathsf{around}(\mathsf{position})\mathbf{.}\mathsf{salary}))\end{aligned}$$

Here, each employee is matched with other employees having the same position using a variant of around:

$$\begin{aligned}&\mathsf{around} : (A \to B) \to (\mathsf{Rel}\{A\} \to \mathsf{Seq}\{A\})\\ &\mathsf{around}(q) = [a_1,\ \ldots, (\!(a_j)\!),\ \ldots,\ a_n]\\ &\qquad\qquad \mapsto [\, a_i \mid q(a_i) = q(a_j)\,].\end{aligned}$$

Note the use of two separate filter combinators. If we merge them into one, $\mathsf{around}(\mathsf{position})$ would list employees with the same position *across all departments.*

We can exploit the input flow to calculate running aggregates.

Example 5.4 *Show a numbered list of employees and their salaries along with the running total.*

$$\begin{aligned}&\mathsf{employee}\\ &\mathbf{:}\mathsf{select}(\mathsf{no} \Rightarrow \mathsf{count}(\mathsf{before}),\\ &\qquad\quad \mathsf{name},\\ &\qquad\quad \mathsf{salary},\\ &\qquad\quad \mathsf{total} \Rightarrow \mathsf{sum}(\mathsf{before}\mathbf{.}\mathsf{salary}))\end{aligned}$$

The primitive before exposes its input flow up to and including the current input value:

$$\begin{aligned}&\mathsf{before} : \mathsf{Rel}\{A\} \to \mathsf{Seq}\{A\}\\ &\mathsf{before} = [a_1,\ \ldots, (\!(a_j)\!),\ \ldots,\ a_n] \mapsto [a_1,\ \ldots,\ a_j].\end{aligned}$$

Using before, we can enumerate the rows in the output

$$\mathsf{count}(\mathsf{before}) : \mathsf{Rel}\{\mathsf{Emp}\} \to \mathsf{Int}$$

as well as calculate the running sum of salaries

$$\mathsf{sum}(\mathsf{before}\mathbf{.}\mathsf{salary}) : \mathsf{Rel}\{\mathsf{Emp}\} \to \mathsf{Int}.$$

Example 5.5 *For each department, show employee salaries along with the running total; the total should be reset at the department boundary.*

$$\begin{aligned}&\mathsf{department}\\ &\mathbf{:}\mathsf{select}(\mathsf{name},\\ &\qquad\quad \mathsf{employee}\\ &\qquad\quad \mathbf{:}\mathsf{select}(\mathsf{name},\ \mathsf{salary},\ \mathsf{sum}(\mathsf{before}\mathbf{.}\mathsf{salary}))\\ &\qquad\quad \mathbf{:}\mathsf{frame})\end{aligned}$$

The input flow propagates through composition, so that a query executed within

$$\mathsf{department.employee} : \mathsf{Void} \to \mathsf{Seq}\{\mathsf{Emp}\}$$

will see the input flow containing all the employees in all departments. To reset the input flow at a certain boundary, we use the combinator

$$\mathsf{frame} : (\mathsf{Rel}\{A\} \to M\{B\}) \to (A \to M\{B\}).$$

6 Conclusion and Related Work

In this paper, we introduce a combinator-based query language, *Rabbit*, and, using the framework of (co)monads and (bi-)Kleisli arrows [18, 25], describe the denotation of database queries.

The functional database model gives us the underlying category of serializable data. It consists of entity classes such as Dept and Emp; simple value types Void, Bool, Int, Text and others; and their composites $\langle \ldots \rangle$, Opt$\{\ldots\}$, and Seq$\{\ldots\}$. We bootstrap the query model by assuming that a query with input of type A and output of type B can be expressed as an arrow

$$A \to B.$$

To model optional and plural queries, we wrap their output in a monadic container and represent them as Kleisli arrows

$$A \to M\{B\}.$$

The containers form a family $\mathcal{M}$ of monads equipped with a join-semilattice structure: for any $M_1, M_2 \in \mathcal{M}$, there exists $M_1 \sqcup M_2 \in \mathcal{M}$ with natural injections

$$M_1\{A\} \to (M_1 \sqcup M_2)\{A\} \leftarrow M_2\{A\}.$$

To represent query parameters and the input flow, we wrap the query input in a comonadic container, expressing context-aware queries as bi-Kleisli arrows

$$W\{A\} \to M\{B\}.$$

Dually, the comonadic containers form a meet-semilattice $\mathcal{W}$ of comonads: for any $W_1, W_2 \in \mathcal{W}$, there exists $W_1 \sqcap W_2 \in \mathcal{W}$ with natural projections

$$W_1\{A\} \leftarrow (W_1 \sqcap W_2)\{A\} \to W_2\{A\}.$$

Moreover, for any monad $M \in \mathcal{M}$ and comonad $W \in \mathcal{W}$, there should exist a distributive law

$$W\{M\{A\}\} \to M\{W\{A\}\}.$$

Then, the composition of queries

$$p : W_1\{A\} \to M_1\{B\}, \quad q : W_2\{B\} \to M_2\{C\}$$

could be defined as a query of the form

$$p \,.\, q : W\{A\} \to M\{C\} \qquad (W = W_1 \sqcap W_2,\ M = M_1 \sqcup M_2)$$

constructed using the lattice structures of $\mathcal{M}$ and $\mathcal{W}$, compositional properties of monads and comonads, and the distributive law for M and W.

Rabbit has its roots in the authors' work on a URL-based query language [11], which provided a navigational interface to SQL databases. While looking for a way to formally specify this language, we arrived at the combinator-based query model.

Early on, we adopted the navigational approach of XPath [7], which led us to represent the schema as a rooted graph (e.g., Figure 2) and queries as paths in this graph. We recognized that each graph arc has some cardinality, and, consequently, so does each path. Next came the realization that, for any dataset, the dataset values are all related to each other, and this relationship can be denoted as a plural self-referential arc around. We discovered that the rule for composing around with other plural arcs is exactly the distributive law for the Rel comonad over the Seq monad, which pushed us to model database queries as Kleisli arrows.

Monads and their Kleisli arrows came to be a standard tool in denotational semantics after Moggi [18] used them to define a generic compositional model of computations. By varying the choice of monad, he expressed partiality, exceptions, input-output, and other computational effects. Uustalu and Vene [25] used a dual model of comonads and co-Kleisli arrows to describe semantics of dataflow programming. They also discussed distributive laws of a comonad over a monad. In the context of databases, Spivak [23] suggested using monads to encode data with complex structure. Monad comprehension syntax [24, 4] forms the core of query interfaces such as Kleisli [27] and LINQ [17].

The graph representation of the database schema is a variation of the functional database model [16, 22], which gave rise to a number of query languages: FQL [3], DAPLEX [21], GENESIS [1], Kleisli [27] and others; see [13] for a comprehensive survey. Among them, FQL and its derivatives are remarkably close to Rabbit—Example 1.1 is a valid query in both. The key difference is that we interpret the period ("**.**") as a composition of Kleisli arrows, which implies, for instance, that we cannot define count as Seq$\{A\} \to$ Int and write employee.count for *the number of employees*. Instead, we have to accept count as a query combinator.

Combinators are higher-order functions that serve to construct expressions without bound variables. They were introduced as the building blocks of mathematical logic [20, 8], from where they migrated to programming practice, becoming a popular tool for constructing DSLs; examples are found in diverse domains such as parsers [26, 14], reactive animation [9], financial contracts [15], and the view-update problem [12].

Although a few combinator-based query models have been proposed [3, 2, 1, 10, 6], it is generally accepted that "combinator-style languages are difficult for users to master and thus ill-suited as query languages" [6]. We have to disagree. The syntax of a combinator-based DSL directly mirrors its semantics, making it an *executable specification*. This is an attractive property for a language oriented towards domain experts—if the semantics does not contradict the experts' intuition.

In Rabbit, the intuition relies upon the hierarchical data model, which is simple, familiar and prolific. For querying purposes, we view a functional database as a universal hierarchical document obtained by unfolding the database schema into a potentially infinite schema tree (e.g., Figure 3). We learned this technique from concurrency theory, where static "system" models are unfolded into runtime "behavior" models [19]. This technique has also been used to specify an adjunction between the network and hierarchical data models [5].

Rabbit's query model lets us rigorously define the basic notions of data analysis. Indeed, it can naturally express optional and plural relationships, database navigation, transitive closure of hierarchical relationships, aggregate, grouping and data cube operations, query parameters, and window functions. All operations on data are represented as query combinators, which has many practical advantages. For data analysts, it allows them to build queries incrementally, validating the output at each step. For programmers, it dramatically simplifies GUI-driven query construction. Finally, Rabbit may be adapted to a variety of application domains by extending the sets of primitives, combinators, and (co)monadic containers.

7 Acknowledgements

We are indebted to Catherine Devlin for her early support of the project, and our colleagues at Prometheus Research for their continuous feedback.

Bibliography

[1] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Transactions on Database Systems*, 13(3):231–262, 1988.

[2] A. Bossi and C. Ghezzi. Using FP as a query language for relational data-bases. *Computer Languages*, 9(1):25–37, 1984.

[3] P. Buneman and R. E. Frankel. FQL — A functional query language. In *SIGMOD '79*, pages 52–58, 1979.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[5] J. Cartmell. Formalizing the network and hierarchical data models — an application of categorical logic. In *CTCS '85*, pages 466–492, 1985.

[6] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *SIGMOD '96*, pages 401–412, 1996.

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0. Technical Report REC-xpath-19991116, W3C, 1999.

[8] H. B. Curry. Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, 52(3):509–536, 1930.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP '97*, pages 263–273, 1997.

[10] M. Erwig and U. W. Lipeck. A functional DBPL revealing high level optimizations. In *DBPL '91*, pages 306–321, 1991.

[11] C. C. Evans. HTSQL — a native web query language. In *ICOMP '07*, pages 439–445, 2007.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL '05*, pages 233–246, 2005.

[13] P. M. D. Gray, P. J. H. King, and A. Poulovassilis. Introduction to the use of functions in the management of data. In P. M. D. Gray, L. Kerschberg, P. J. H. King, and A. Poulovassilis, editors, *The Functional Approach to Data Management*, pages 1–54. Springer, Berlin, Heidelberg, 2004.

[14] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, School of Computer Science and IT, University of Nottingham, 1996.

[15] S. L. P. Jones, J. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. In *ICFP '00*, pages 280–292, 2000.

[16] L. Kerschberg and J. E. S. Pacheco. A functional data base model. Technical Report 2/1976, Departamento de Informatica, Pontificia Universidade Catolica, Rio de Janeiro, Brazil, 1976.

[17] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06*, page 706, 2006.

[18] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[19] M. Nielsen, V. Sassone, and G. Winskel. Relationships between models of concurrency. In *REX '93*, pages 425–476, 1994.

[20] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[21] D. W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[22] E. H. Sibley and L. Kerschberg. Data architecture and data model considerations. In *AFIPS '77*, pages 85–96, 1977.

[23] D. I. Spivak. Kleisli database instances. *CoRR*, abs/1209.1011, 2012.

[24] P. Trinder and P. Wadler. Improving list comprehension database queries. In *TENCON '89*, pages 186–192, 1989.

[25] T. Uustalu and V. Vene. The essence of dataflow programming. In *CEFP '05*, pages 135–167, 2005.

[26] P. Wadler. How to replace failure by a list of successes. In *FPCA '85*, pages 113–128, 1985.

[27] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.