# Contents

1 Introduction 2

2 FQL Basics 3
   2.1 Schemas 3
   2.2 Instances 4
      2.2.1 Visual Editing 6
      2.2.2 Category of Elements 6
      2.2.3 Relationalization and Observation 7
   2.3 Schema Mappings 8
      2.3.1 Schema Flow 10
   2.4 Transformations (database homomorphisms) 10
   2.5 Data Migration 11
      2.5.1 Delta 11
      2.5.2 Pi 12
      2.5.3 Sigma 13
      2.5.4 Instance Flow 15
      2.5.5 Data Migration on Transformations 15
      2.5.6 Monads 15
   2.6 Queries and Query Composition 17
   2.7 Generating Queries from Schema Correspondences 19
   2.8 Union Schemas and Inclusion Mappings 19
   2.9 Cyclic Schemas 20
   2.10 Enumerations (User defined types) 20

3 Programming with FQL 21
   3.1 Categorical Combinators 21
   3.2 Programming with Schemas and Mappings 21
      3.2.1 Co-products of Schemas 21
      3.2.2 Products of Schemas 22
   3.3 Programming with Instances and Transformations 23
      3.3.1 Co-products of Instances 23
      3.3.2 Products of Instances 24

4 Connecting FQL with Other Systems 27
   4.1 Compiling to SQL 27
   4.2 JDBC Support 27
   4.3 Compiling to Embedded Dependencies (EDs) 28
   4.4 Translating SQL to FQL 28
   4.5 Translating Polynomials to FQL 29
   4.6 RDF Representation of Schemas and Instances 29
1 Introduction

FQL, a functorial query language, implements functorial data migration. Although functorial data migration is formally defined using the language of category theory, it is possible to understand functorial data migration at the level of relational database tables. Indeed, the FQL compiler emits SQL (technically, PSM) code that can be run on any RDBMS. The FQL compiler is hosted inside of an integrated development environment, the FQL IDE. The FQL IDE is an open-source Java program that provides a code editor for and visual representation of FQL programs. A screenshot of the initial screen of the FQL IDE is shown below.

The FQL IDE is a multi-tabbed text file editor that supports saving, opening, copy-paste, etc. Associated with each editor is a “compiler response” text area that displays the SQL output of the FQL compiler, or, if compilation fails, an error message. The built-in FQL examples can be loaded by selecting them from the “load example” combo box in the upper-right. In the rest of this tutorial we will refer to these examples. By default, only some examples are shown; more can show by enabling an option. Compilation can be aborted with the “abort” button in the Tools menu. Abort is “best effort” and can leave FQL in an inconsistent state; it is provided to allow users to terminate FQL gracefully. The FQL IDE contains both a naive SQL engine and the (faster) H2 SQL engine; to choose which to use, see the options menu. A code formatter is available in the edit menu.

To run FQL with more than the default 64mb heap, you must use command line options:

```
java -Xms512m -Xmx2048m -jar fql.jar
```
2 FQL Basics

An FQL program is an ordered list of uniquely named declarations. Each declaration defines either a schema, an instance, a mapping, or a query. Identifiers are case insensitive and must be unique. We now describe each of these concepts in turn. Comments in FQL are Java style, either “//” or “/* */”. Negative integers must be quoted with double quotes.

2.1 Schemas

Select the “typed employees” examples. This action will create a new tab containing the following FQL code:

```fql
schema S = {
  nodes
  Employee, Department;

  attributes
  name : Department -> string,
  first : Employee -> string,
  last : Employee -> string;

  arrows
  manager : Employee -> Employee,
  worksIn : Employee -> Department,
  secretary : Department -> Employee;

  equations
  Employee.manager.worksIn = Employee.worksIn, //1
  Department.secretary.worksIn = Department, //2
  Employee.manager.manager = Employee.manager; //3
}
```

This declaration defines a schema $S$ consisting of two nodes, three attributes, three arrows, and three equations. In relational terminology, this means that

- Each node corresponds to an entity type. In this example, the two types of entities are employees and departments.

- A node/entity type may have any number of attributes. Attributes correspond to observable atoms of type int or string. In this example, each department has one attribute, its name, and each employee has two attributes, his or her first and last name.

- Each arrow $f : X \to Y$ corresponds to a total function $f$ from entities of type $X$ to entities of type $Y$. In this example, manager maps employees to employees, worksIn maps employees to departments, and secretary maps departments to employees.
The equations specify the data integrity constraints that must hold of all instances that conform to this schema. FQL uses equalities of paths as constraints. A path $p$ is defined inductively as

$$ p ::= \text{node} \mid p.\text{arrow} $$

Intuitively, the meaning of “.” is composition. In this example, the constraints are: 1) every employee must work in the same department as his or her manager; 2) every departmental secretary must work for that department; and 3) there are employees and managers, but not managers of managers, managers of managers of managers of managers, etc.

The FQL IDE can render schemas into a graphical form similar to that of an Entity-Relationship (ER) diagram. Press “compile”, and select the schema $S$ from the viewer:

![Graphical Representation of Schema S](image)

Note that the four sections, “nodes”, “attributes”, “arrows”, and “equations” are ended with semi-colons, and must appear in that order, even when a section is empty. The “denotation” prints the category that the schema denotes.

### 2.2 Instances

Continuing with the built-in “typed employees” example, we see that it also contains FQL code that defines an instance of the schema $S$ defined in the previous section:

```plaintext
instance I = {
    nodes
    Employee -> { 101, 102, 103 },
    Department -> { q10, x02 },
}
```
attributes
  first -> { (101, Alan), (102, Camille), (103, Andrey) },
  last -> { (101, Turing), (102, Jordan), (103, Markov) },
  name -> { (q10, AppliedMath), (x02, PureMath) };  

arrows
  manager -> { (101, 103), (102, 102), (103, 103) },
  worksIn -> { (101, q10), (102, x02), (103, q10) },
  secretary -> { (q10, 101), (x02, 102) };  
} : S

This declaration defines an instance $I$ that conforms to schema $S$. This means that

- To each node/entity type corresponds a set of globally unique IDs. In this example, the employee IDs are 101, 102, and 103, and the departmental IDs are q10 and x02.

- Each attribute corresponds to a function that maps IDs to atoms. In this example, we see that employee 101 is Alan Turing, employee 102 is Camille Jordan, employee 103 is Andrey Markov, department q10 is AppliedMath, and department x02 is PureMath.

- Each arrow $f : X \to Y$ corresponds to a function that maps IDs of entity type $X$ to IDs of entity type $Y$. In this example, we see that Alan Turing and Andrey Markov work in the AppliedMath department, but Camille Jordan works in the PureMath department.

FQL assumes that every node, attribute, and arrow is stored as a binary table; node tables are stored as reflexive tables with types of the form $(x, x)$. In addition, FQL assumes that the exact value of IDs are irrelevant, and in fact FQL will replace our IDs “q10, x02, 101, 102” and “103” with generated IDs 1,2,3,4,5. Indeed, replacement of IDs will happen often with FQL data migration operations. To visualize this instance, press “compile”, select the instance from the viewer list, and click the “tabular” tab:
To enter a string that contains spaces, simply quote it. The ASWRITTEN keyword can automatically generate attributes based on node IDs (see “Written macro” example).

2.2.1 Visual Editing

Instances can be editing using a graphical, tabular interface similar to how instances are rendered. To visually edit an instance, right-click on an instance in the main text editor and select “visually edit”. A modal dialog will pop up; by clicking on a node, the joined view of that node appears. Double-clicking in a cell allows to change the contents of the cell; enter finalizes the change. Rows can be selected using the shift key. Note that instances need not be well-formed to be edited, although the FQL program must parse and the schema of the instance must be well-formed. Instances can be edited in such a way that they output non well-formed FQL. Empty values in cells are treated as empty strings.

2.2.2 Category of Elements

Provided the “Elements” option is enabled, the “Elements” tab in an instance displays the instance as a category, “the category of its elements”. In this view, nodes are entities and arrows are foreign-key correspondences. Th Elements view is well illustrated using the “People” example:
2.2.3 Relationalization and Observation

Associated with each type of entity in an instance is an “observation table”. For an entity type/node $N$, the observation table joins together all attributes reachable by all paths out of $N$. Consider the “relationalize” example:

schema $C=${
  nodes $A$;
  attributes $a:A\rightarrow$string;
  arrows $f:A\rightarrow A$;
  equations $A.f.f.f.f = A.f.f$;  
}

instance $I:C=${
  nodes $A$->{1,2,3,4,5,6,7};
  attributes $a$->{(1,1),(2,2),(3,3),(4,1),(5,5),(6,3),(7,5)};
  arrows $f$->{(1,2),(2,3),(3,5),(4,2),(5,3),(6,7),(7,6)};  
}

instance Rel$I:C$=relationalize $I$
transform trans = Rel$I$.relationalize
The operation “relationalize” will equate IDs that are not distinguished by attributes. In this example, 7 rows would collapse to 4. The transform `RelI.relationalize` gives the mapping of the 7 rows to the 4 rows. The relationalize operation is expensive, but necessary to faithfully implement relational projection on relations that have been encoded as functorial instances. The observation view displays the relationalization of every instance and can be enabled in the options menu. The Observables pane is never computed via JDBC.

- **If your program is taking too long to run, try disabling the observables and elements GUI panes.** FQL computes all GUI panes; they are not computed lazily.

### 2.3 Schema Mappings

Next, load the “typed delta” example. It defines two schemas, \( C \) and \( D \), and a mapping \( F \) from \( C \) to \( D \):

```plaintext
schema C = {
  nodes T1, T2;
  attributes
  t1_ssn : T1 -> string,
  t1_first : T1 -> string,
  t1_last : T1 -> string,
  t2_first : T2 -> string,
  t2_last : T2 -> string,
  t2_salary : T2 -> int;
  arrows; equations;
}

schema D = {
  nodes T;
  attributes
  ssn0 : T -> string,
  first0 : T -> string,
  last0 : T -> string,
  salary0 : T -> int;
  arrows; equations;
}

mapping F = {
  nodes T1 -> T, T2 -> T;
  attributes
  t1_ssn -> ssn0, t1_first -> first0, t1_last -> last0,
  t2_first -> first0, t2_last -> last0,
  t2_salary -> salary0,
  arrows; equations;
}
```
A mapping $F : C \rightarrow D$ consists of three parts:

- a mapping from the nodes in $C$ to the nodes in $D$
- a mapping from the attributes in $C$ to the attributes in $D$
- a mapping from the arrows in $C$ to paths in $D$

A mapping must respect the equations of $C$ and $D$: if $p_1$ and $p_2$ are equal paths in $C$, then $F(p_1)$ and $F(p_2)$ must be equal paths in $D$. If this condition is not met, FQL will throw an exception. Our example mapping is rendered in the viewer as follows:

An identity mapping can be formed using the keyword “id” as follows:

```
mapping F = id C
```

Mappings can be composed using the keyword “then” (parenthesis required) as follows:

```
mapping F = (G then H)
```
2.3.1 Schema Flow

In the viewer, select “Schema Flow” at the bottom left to see a graph where every vertex is a named schema, and every edge is a named mapping. Clicking on a vertex or an edge will cause the viewer display to change to that schema or mapping.

2.4 Transformations (database homomorphisms)

For each schema $T$ and $T$-instances $I, J$, a transformation $f : I \rightarrow J$ is a database homomorphism from $I$ to $J$: for each node $n \in T$, $f_n$ is a constraint and attribute respecting function from $I_n$ IDs to $J_n$ IDs. Transformations are illustrated in the “Transform2” example:

```plaintext
schema C = {
  nodes A;
  attributes att:A->string;
  arrows f:A->A;
  equations A.f.f.f=A.f.f;
}

instance I = {
  nodes A->{1,2,3,4,5};
  attributes att->{(1,common),(2,common),(3,common),(4,common),(5,common)};
  arrows f->{(1,2),(2,3),(3,3),(4,2),(5,3)};
} : C

instance J = {
  nodes A->{1,2,3};
  attributes att->{(1,common),(2,common),(3,common)};
  arrows f->{(1,2),(2,3),(3,3)};
} : C

//transform BadTransform = {
//  nodes A->{(1,1),(2,2),(3,4)};
//} : J -> I

transform GoodTransform1 = {
  nodes A->{(1,1),(2,2),(3,3)};
} : J -> I

transform GoodTransform2 = {
  nodes A->{(1,1),(2,2),(3,3),(4,1),(5,2)};
} : I -> J
```

The viewer for transforms is similar to the “Elements” viewer instances, except that IDs are colored according to their corresponding nodes.
2.5 Data Migration

Associated with a mapping \( F : C \rightarrow D \) are three data migration operators:

- \( \Delta_F \), taking \( D \) instances to \( C \) instances, roughly corresponding to projection
- \( \Pi_F \), taking \( C \) instances to \( D \) instances, roughly corresponding to join
- \( \Sigma_F \), taking \( C \) instances to \( D \) instances, roughly corresponding to union

In general, certain restrictions must be placed on \( F \) to guarantee the above operations exist. We now describe each in turn.

2.5.1 Delta

Continuing with the “typed delta” example, we see that the FQL program also defines a \( D \)-instance \( J \), and computes \( I := \Delta_F(J) \):

\[
\text{instance } I = \text{delta } F \; J
\]

Graphically, we have

In effect, we have projected the columns salary0 and last0 from \( J \).
2.5.2 Pi

Load the “typed pi” example:

\[
\text{schema } C = \{ \\
\quad \text{nodes } c1, c2; \\
\quad \text{attributes} \quad \text{att1} : c1 \rightarrow \text{string}, \quad \text{att2} : c1 \rightarrow \text{string}, \quad \text{att3} : c2 \rightarrow \text{string}; \\
\quad \text{arrows}; \\
\quad \text{equations}; \\
\}\]

\[
\rightarrow_F
\]

\[
\text{schema } D = \{ \\
\quad \text{nodes } d; \\
\quad \text{attributes} \quad \text{a1} : d \rightarrow \text{string}, \quad \text{a2} : d \rightarrow \text{string}, \quad \text{a3} : d \rightarrow \text{string}; \\
\quad \text{arrows}; \\
\quad \text{equations}; \\
\}\]

\[
\text{mapping } F = \{ \\
\quad \text{nodes } c1 \rightarrow d, \ c2 \rightarrow d; \\
\quad \text{attributes } \text{att1} \rightarrow \text{a1}, \ \text{att2} \rightarrow \text{a2}, \ \text{att3} \rightarrow \text{a3}; \\
\quad \text{arrows}; \\
\} : C \rightarrow D
\]

This example defines an instance \( I : C \) and computes \( J := \Pi_F(I) \):

\[
\text{instance } J = \pi F I
\]

Graphically, this is rendered as:
We see that we have computed the cartesian product of tables $c_1$ and $c_2$. Note that the attribute mapping part of $F$ must be a bijection for $\Pi_F$ to be defined; if this condition fails FQL will throw an exception. There is menu option to allow surjections to be used instead of bijections; we are confident that surjections behave correctly, but haven’t proven all requisite theorems.

2.5.3 Sigma

Load the “sigma” example:

```plaintext
schema C = {
    nodes
    a1, a2, a3, b1, b2, c1, c2, c3, c4;
    attributes;
    arrows
    g1 : a1 -> b1,
    g2 : a2 -> b2,
    g3 : a3 -> b2,
    h1 : a1 -> c1,
    h2 : a2 -> c2,
    h3 : a3 -> c4;
    equations;
}

mapping F = {
    nodes
    a1 -> A, a2 -> A, a3 -> A,
    b1 -> B, b2 -> B,
    c1 -> C, c2 -> C, c3 -> C, c4 -> C;
    attributes;
}

schema D = {
    nodes A, B, C;
    attributes;
    arrows
    G : A -> B,
    H : A -> C;
    equations;
}
```

13
g1 -> A.G, g2 -> A.G, g3 -> A.G,
h1 -> A.H, h2 -> A.H, h3 -> A.H;
} : C -> D

This example defines an instance $I : C$ and computes $J := \Sigma_F(I)$:

instance J : D = sigma F I

Graphically, this is rendered as:

We see that we have computed union of tables $a_1$, $a_2$, and $a_3$ as $A$ (6 rows), the union of tables $b_1$, $b_2$ as $B$ (5 rows), and the union of $c_1$, $c_2$, $c_3$ and $c_4$ as $C$ (7 rows).

To be implementable in SQL, the functor $F$ must satisfy the special condition of being a discrete op-fibration, which basically means “union compatible in the sense of Codd”. Technically, it is possible to define $\Sigma$ for any schema mapping, not just mappings that are union-compatible, and the FQL IDE can still compute unrestricted sigmas, see the “full sigma” example for details. Full sigmas can create null values if that option is checked, but this feature is experimental and dangerous. Full sigma can be implemented using EDs.
2.5.4 Instance Flow

In the viewer, select “Instance Flow” at the bottom left to see a graph where every vertex is a named instance, and every edge is a data migration operation. Clicking on a vertex or an edge will cause the viewer display to change to that instance or data migration. Note that some data migration operations may not be explicitly named; in this case, the operation will still display, but the operation cannot be selected by the list of named FQL declarations.

2.5.5 Data Migration on Transformations

If \( f : I \rightarrow J \) is a transformation, then so is \( \Delta_F(f) : \Delta_F(J) \rightarrow \Delta_F(I) \), \( \Sigma_F(f) : \Sigma_F(I) \rightarrow \Sigma_F(J) \), \( \Pi_F(f) : \Pi_F(I) \rightarrow \Pi_F(J) \), and \( \text{relationalize}(f) : \text{relationalize}(I) \rightarrow \text{relationalize}(J) \). See the All Syntax example for details.

- Note: For a full \( \Sigma \) the input transform must be explicitly named. While not required in theory, this restriction is necessary to allow full sigma on transforms to easily work with external SQL engines that cannot natively support full sigmas.

2.5.6 Monads

For every \( F \) and \( I \), there are canonical transforms \( \text{return} : I \rightarrow \Delta_F(\Sigma_F(I)) \) and \( \text{coreturn} : \Sigma_F(\Delta_F(I)) \rightarrow I \), as well as canonical transforms \( \text{return} : I \rightarrow \Pi_F(\Delta_F(I)) \) and \( \text{coreturn} : \Delta_F(\Pi_F(I)) \rightarrow I \). These transforms come from the mathematical theory of monads. They are illustrated in the Sigma and Pi examples, but here we will focus on the “Connected Components” example. To load it, you must enable “all examples” in the options menu. The basic idea is that these transforms provide a connection between \( I \) and \( \Sigma_F(I) \), and between \( \Pi_F(I) \) and \( I \).

```plaintext
schema Graph = {
    nodes arrow,vertex;
    attributes;
    arrows src:arrow->vertex,tgt:arrow->vertex;
    equations;
}

//has 4 connected components
instance G = {
    nodes arrow->\{a,b,c,d,e\}, vertex->\{t,u,v,w,x,y,z\};
    attributes;
    arrows src->\{(a,t),(b,t),(c,x),(d,y),(e,z)\},
              tgt->\{(a,u),(b,v),(c,x),(d,z),(e,y)\};
} : Graph

schema Terminal = {
    nodes X;
```
attributes;
arrows;
equations;
}

mapping F = {
nodes arrow->X, vertex->X;
attributes;
arrows src->X, tgt->X;
} : Graph -> Terminal

//has 4 rows
instance Components=SIGMA F G

//puts 4 rows into vertex, 4 rows into arrow, corresponding to the connected components
instance I = delta F Components

//gives the transform from the original graph to the connected components
transform t = I.return

In this example, instances on schema Graph are representations of (directed) (multi) graphs: instance G contains 7 vertices and 5 edges. Running the example, we see that Components has 4 elements:

//instance G
{
    nodes
    arrow -> { 1, 2, 3, 4, 5 },
    vertex -> { 6, 7, 8, 9, 11, 10, 12 }
    ;
    attributes
    ;
    arrows
tgt -> { (3,10), (2,11), (5,8), (1,7), (4,6) },
src -> { (3,12), (2,12), (4,8), (5,6), (1,7) };
}

//instance Components
{
    nodes X -> { 15, 14, 20, 12 };
    attributes;
arrows;
}

We might wish to know which connected component each arrow and vertex is a part of. Since the connected components were computed by Σ, we can use Δ to compute another
graph whose arrows are vertices are the connected components. Then, the return monad
operation provides a transform from the original graph to this new graph.

//instance I
{
    nodes
    arrow -> { 32, 31, 30, 29 },
    vertex -> { 33, 34, 35, 36 }
    ;
    attributes
    ;
    arrows
tgt -> { (31,35), (30,34), (32,36), (29,33) },
src -> { (31,35), (30,34), (32,36), (29,33) };
}

//transform t : G -> I
{nodes
    arrow -> {{(4,32),(5,32),(3,31),(1,30),(2,31)},
    vertex -> {{(11,35),(10,35),(12,35),(8,36),(7,34),(6,36),(9,33)};
}

We can see above that t associates certain arrows, such as 2 and 3 to the same connected
component (31).

2.6 Queries and Query Composition

In SQL, unions of select-from-where clauses are the common programming idiom. In FQL,
the common idiom is Σs of Πs of ∆s. Load the “composition” example:

schema S = { nodes s ; attributes; arrows; equations; }
schema T = { nodes t ; attributes; arrows; equations; }
schema B = { nodes b1,b2; attributes; arrows; equations; }
schema A = { nodes a1,a2,a3; attributes; arrows; equations; }

mapping s = { nodes b1 -> s, b2 -> s; attributes; arrows; }: B -> S
mapping f = { nodes b1 -> a1, b2 -> a2 ; attributes; arrows; } : B -> A
mapping t = { nodes a1 -> t, a2 -> t, a3 -> t ; attributes; arrows; }: A -> T

query q1 = delta s pi f sigma t

In general, a query is simply a convenient shorthand with special support for composition.
Queries may be evaluated using the keyword “eval”:

instance J = ...
instance I = eval q1 J

FQL includes special support for composing queries. Continuing with the “composition” example, we see that it defines another query:

schema D = { nodes d1,d2 ; attributes; arrows; equations; }
schema C = { nodes c ; attributes; arrows; equations; }
schema U = { nodes u ; attributes; arrows; equations; }

mapping u = { nodes d1 -> t, d2 -> t ; attributes; arrows;}: D -> T
mapping g = { nodes d1 -> c, d2 -> c ; attributes; arrows;}: D -> C
mapping v = { nodes c -> u ; attributes; arrows; }: C -> U

query q2 = delta u pi g sigma v

We compose our two queries as follows (parenthesis required):

query q = (q1 then q2)

FQL supports another top-level type, the \texttt{QUERY}, for free compositions of data migration functors (i.e., not triplets). Full sigma (\texttt{SIGMA}) can be used with \texttt{QUERY}. 
2.7 Generating Queries from Schema Correspondences

FQL has experimental support for generating queries from attribute correspondences (schema matching). Because such mappings are often not discrete op-fibrations, FQL defines these using QUERY. Many different queries can be generated from a correspondence; these are identified by strings. Queries can be evaluated on instances with EVAL, similarly to eval.

Load the “Schema Matching” example (requires the additional examples options):

```plaintext
schema ab = { nodes a, b; attributes atta : a -> string, attb : b -> string; arrows f : a -> b; equations; }

schema c = { nodes c; attributes attc : c -> string; arrows ; equations; }

QUERY q1 = match {(atta,attc),(attb,attc)} ab c "delta sigma forward"
QUERY q2 = match {(atta,attc),(attb,attc)} ab c "delta sigma backward"
QUERY q3 = match {(atta,attc),(attb,attc)} ab c "delta pi forward"
QUERY q4 = match {(atta,attc),(attb,attc)} ab c "delta pi backward"
```

The viewer displays the associated data migration queries:

```plaintext
(SIGMA {nodes c -> right_c; attributes attc -> right_attc; arrows;} : c -> )
{nodes left_a, left_b, right_c; attributes left_atta : left_a -> string, left_attb : left_b -> string, right_attc : right_c -> string; arrows left_f : left_a -> left_b; equations; }
then
delta {nodes b -> left_b, a -> left_a; attributes attb -> left_attb, atta -> left_atta; arrows f -> left_a.left_f; }
: ab -> { nodes left_a, left_b, right_c; attributes left_atta : left_a -> string, left_attb : left_b -> string, right_attc : right_c -> string; arrows left_f : left_a -> left_b; equations; }
```

2.8 Union Schemas and Inclusion Mappings

For convenience, FQL provides a “macro” to union two schemas together (which can fail, for example, when the two schemas have attributes with the same name but different types), and a “macro” for constructing inclusion mappings. Details are in the “Sub Schema” example:

```plaintext
schema ab = { nodes a, b; attributes atta : a -> string, attb : b -> string; arrows f : a -> b; equations; }
```
2.9 Cyclic Schemas

By default, FQL requires all schemas to be finite. Such schemas can be cyclic, provided they have “finitizing” equations – see the “Typed Employees” example. However, FQL can implement \( \Delta, \Sigma \) migrations over infinite schemas, provided the “do not validate mappings” option is checked. When checked, mappings will not be checked for functoriality, and inputting a non-functorial mapping may result in undefined behavior.

2.10 Enumerations (User defined types)

FQL supports named enumerations. In the “Enums” example, we see:

```plaintext
enum color = { red, green, blue }
```
```plaintext
schema S = { nodes S_node;
  attributes S_att : S_node -> color; arrows; equations; } 
```
3 Programming with FQL

3.1 Categorical Combinators

FQL contains programming capabilities to FQL in the guise of categorical combinators. Intuitively, these combinators are a standard functional programming language similar to e.g., point-free Haskell. Their semantics derives from the fact that the category of schemas and mappings is bi-cartesian closed, and for each $T$, the category of $T$-instances and their homomorphisms is a topos. The main GUI contains a type-checker menu option.

3.2 Programming with Schemas and Mappings

Schema and mapping expressions can be freely nested:

```
schema C = {nodes; attributes; arrows; equations;}
schema C1 = void
schema C2 = unit {string, int}
schema C3 = (C + C)
schema C4 = (C * C)
schema C5 = opposite C

mapping F = id C
mapping F1 = (F then F)
mapping F2 = {nodes; attributes; arrows;} : C -> C
mapping F3 = inl C C
mapping F4 = inr C C
mapping F5 = (F3 + F4)
mapping F6 = fst C C
mapping F7 = snd C C
mapping F8 = (F6 * F7)
mapping F9 = void C
mapping F10= unit {string, int} C
mapping F11 = opposite F
```

Given a signature $C$, opposite $C$ defines the signature with all the arrows reversed, and given a mapping $F : C \rightarrow D$, opposite $F : opposite C \rightarrow opposite D$.

3.2.1 Co-products of Schemas

The “Co-products Schema” example illustrates co-products of schemas:

```
schema C = { nodes a, b; attributes att : a -> string;
arrows f : a -> b, g : a -> a; equations a.g = a; }
schema D = { nodes a; attributes att : a -> string; arrows ; equations ; }
```
schema E = (C + D)
mapping f = inl C D
mapping g = inr C D
mapping h = (f + g) // this is actually the identity!

schema X = void
mapping q = void C

schema Y = ((C + (C + C)) + void)

Schemas C and D are disjointly unioned in E := C + D: each node in C appears in C + D (albeit with an automatically generated name), and each node in D appears in C + D. Similarly, the attributes and arrows of C and D are injected into C + D. The injection mappings have types inl C D : C → C + D and inr C D : D → C + D. Given mappings f : C → X and g : D → X, mapping (f + g) (parenthesis required) has type C + D → X. Co-products obey the equations

\[ \text{inl}; (f + g) = f \quad \text{inr}; (f + g) = g \quad (\text{inl} + \text{inr}) = \text{id} \]

All manipulations of co-product schemas must be done using inl, inr, and (+). The empty schema is written void and for every schema C, the mapping void C : void → C is the unique (empty) map to C. The empty schema is a unit for co-products in the sense that C and void + C are isomorphic.

3.2.2 Products of Schemas

The ‘Products’ Schema’ example illustrates products of schemas:

schema S = {
    nodes a, b, c;
    attributes att:a->string;
    arrows f:a->b, g:b->c, h:a->c;
    equations a.h = a.f.g;
}

schema T = {
    nodes x, y;
    attributes att:x->string;
    arrows u:x->y, z:x->y;
    equations x.u = x.z;
}

mapping F = {
    nodes x -> a, y -> c;
    attributes att->att;
}
arrows u -> a.f.g, z->a.f.g;
} : T -> S

schema A = (S * T)
mapping p1 = fst S T
mapping p2 = snd S T
mapping p = (p1*p2) //is identity

schema X = unit {string}
mapping H = unit {string} T

Schemas $C$ and $D$ are multiplied in $E := C \times D$: nodes in $C \times D$ are pairs $(c, d)$ where $c \in C$ and $d \in D$, albeit with automatically generated names. Hence, the attributes and arrows of $C \times D$ are pairs of attributes and arrows from $C$ and $D$. The projection mappings have types $\text{fst } C \times D : C \times D \to C$ and $\text{snd } C \times D : C \times D \to D$. Given mappings $f : X \to C$ and $g : X \to D$, mapping $(f \times g)$ (parenthesis required) has type $X \to C \times D$. Products obey the equations

$$ (f \times g); \text{fst} = f \quad (f \times g); \text{snd} = g \quad (\text{fst} \times \text{snd}) = \text{id} $$

All manipulations of product schemas must be done using $\text{fst}$, $\text{snd}$, and $(\times)$. The schema with one node and one attribute for each type \{t_1,\ldots\} is written unit \{t_1,\ldots\} and for every schema $C$, the mapping unit \{t_1,\ldots\} $C : C \to \text{unit}\{t_1,\ldots\}$ is the unique (collapsing) map to unit \{t_1,\ldots\}. The terminal schema is a unit for products in the sense that $C$ and unit \{t_1,\ldots\} $\times C$ are isomorphic.

### 3.3 Programming with Instances and Transformations

For each schema $T$ and $T$-instances $I,J$, a transformation $f : I \to J$ is a database homomorphism from $I$ to $J$: for each node $n \in T$, $f_n$ is a constraint and attribute respecting function from $I_n$ IDs to $J_n$ IDs. Transformations are The following illustrates all FQL syntax for programming with instances and transformations:

instance I  = { nodes; attributes; arrows; } : C
instance I1 = delta F I
instance I2 = pi F I
instance I3 = sigma F I
instance I4 = relationalize I
instance I5 = SIGMA F I
instance I8 = (I + I)
instance I9 = (I * I)
instance I11 = unit C
instance I12 = void C
instance I6 = external C name
instance I7 = eval q I
instance I7x = EVAL Q1 I
Note that equality for instances is nominal, meaning that instance names matter; if $I := E$ and $J := E$ are two instances, then $I \neq J$ even though $I$ and $J$ have the same definition. Indeed, $I$ and $J$ will have different GUIDs. As a result the FQL language of instances and transformations is somewhat different than the FQL language of schemas and mappings. Transforms may be freely nested, but Instances cannot be, as they rely on effectful SQL operations.

### 3.3.1 Co-products of Instances

The “Co-products Inst’ example illustrates co-products of instances (of the same schema):

```plaintext
schema S = {
  nodes a, b;
  attributes att : a -> string;
  arrows f : a -> b;
  equations;
}

instance I = {
  nodes a -> {1,2}, b -> {3};
  attributes att -> {(1,one),(2,two)};
  arrows f -> {(1,3),(2,3)};
} : S

instance J = {
  nodes a -> {a,b,c}, b -> {d,e};
```
attributes att -> {((a,foo),(b,bar),(c,baz))};
arrows f -> {((a,d),(b,e),(c,e))};
} : S

instance A = (I + J)
transform K = A.inl
transform L = A.inr
transform M = A.(K + L) //is id

instance N = void S
transform O = N.void J

Instances I and J are unioned in A := I + J: each ID in I appears in A (albeit with a fresh, automatically generated name), and each ID in J appears in A. Similarly, the attributes and arrows of I and J are injected into A. The injection transformations have types A.inl I → A and A.inr J → A. Given transformations f : I → X and g : J → X, transformation A.(f + g) (parenthesis required) has type A → X. The global unique ID assumption ensures all unions are disjoint. Co-products obey the equations

\[ A.inl; A.(f + g) = f \quad A.inr; A.(f + g) = g \quad A.(A.inl + A.inr) = id \]

All manipulations of co-product instances must be done using \textit{inl}, \textit{inr}, and (+). The empty instance \(U\) of schema \(T\) is written \(U := \text{void } T\) and for every instance \(V\) of type \(T\), the mapping \(U.\text{void } V : U \to V\) is the unique (empty) map from \(V\). The empty instance is a unit for co-products in the sense that \(V\) and \(\text{void } + V\) are isomorphic.

### 3.3.2 Products of Instances

The "Products Inst" example illustrates products of instances (of the same schema):

```latex
schema S = {
nodes a, b;
attributes att : a -> string;
arrows f : a -> b;
equations;
}

instance I = {
nodes a -> \{1,2\}, b -> \{3\};
attributes att -> \{(1,common),(2,common)\};
arrows f -> \{(1,3),(2,3)\};
} : S

instance J = {
nodes a -> \{a,b,c\}, b -> \{d,e\};
attributes att -> \{(a,common),(b,common),(c,baz)\};
```
arrows f → {(a,d),(b,e),(c,e)};
} : S

instance A = (I * J)

transform K = A.fst

transform L = A.snd

transform M = A.(K * L) //is id

schema X = {
  nodes a;
  attributes;
  arrows;
  equations;
}

instance N = unit X

transform O = N.unit N

Instances I and J are joined in A := I × J: each (fresh) ID in I × J corresponds to a pair of ids (i,j) with i ∈ I and j ∈ J such that the attributes of i and j match. The attributes and arrows of I and J are multiplied into I × J. The projection transformations have types A.fst A → I and A.snd A → J. Given transformations f : X → I and g : X → J, transformation A.(f × g) (parenthesis required) has type X → A. Products obey the equations

\[ A.(f \times g); A.fst = f \quad A.(f \times g); A.inv = g \quad A.(A.fst \times A.snd) = id \]

All manipulations of product instances must be done using fst, snd, and (×). The terminal instance U of schema T (where T has only enum (or no) attributes) is written U := unit T and for every instance V of type T, the mapping U.unit V : V → U is the unique (collapsing) map to U. The unit instance is a unit for products in the sense that V and unit × V are isomorphic. See the “Products” example for details.
4 Connecting FQL with Other Systems

4.1 Compiling to SQL

The FQL compiler emits (very naive) SQL code that implements the FQL program. In fact, the FQL IDE executes the generated SQL to populate the viewer. The generated SQL may simply be copied into a command-line RDBMS top-level. For example, it can be executed by “mysql_embedded”. However, to use the generated SQL correctly, note the following:

- In mySQL, certain column names are not allowed, such as “left” and “right”.
- A binary table $R$ of an instance $I$ is referred to as $I_R$. Hence, every node, attribute, and arrow in a schema must have a unique name. Names may appear in multiple schema, however.
- The “drop” command may be used to drop tables in the output SQL:

  \[
  \text{drop } I \ t1 //\text{any number of instance/transformations names}
  \]

If given a single argument on the command line, a string, the FQL IDE will compile it to SQL and emit the compiled SQL on stdout. If given no arguments, the FQL IDE launches its GUI.

To use pre-existing database tables (for instances or transforms) with the generated SQL output, CREATE TABLE commands in the generated SQL may need be suppressed using the “external” keyword. This mechanism is described in more detail in the “external” example. The original GUIDifying substitution for a manually entered instance/transform $I$ is stored as tables $I_{\text{subst}}$ and $I_{\text{subst inv}}$. When computing a data migration, FQL will maintain a number of additional that are used for computing various transforms. For example, if $I$ is the result of a $\Pi$ migration, then FQL will keep the “limit table” for $I$.

4.2 JDBC Support

The FQL IDE contains a SQL interpreter, but can be configured to use an external SQL engine using JDBC. To enable JDBC, select the appropriate check-box in the options menu, and input the Java class name of your JDBC driver, as well as the URL to your database instance. Henceforth, the FQL IDE should behave exactly as before, except that it will send its emitted SQL to the external database, and query the external database to populate the viewer. We have tested JDBC functionality on mySQL on mac. Key points:

- To load the JDBC driver from an external jar, you must not run java using “java -jar”. Instead, you must run “java” directly, for example:

  \[
  \text{java -cp "./mysql-connector-java-5.1.27-bin.jar:/fql3.jar" fql.FQL}
  \]

- On compile, the FQL IDE sends the prelude the the engine, executes the emitted SQL code, and then sends the postlude.
• To support FQL, the database engine must support variables in the form:

\[
\text{set @guid := 0}
\]

• The database to connect to is specified in the options menu. This database should be empty, save for any “external tables”. To delete and create databases in the external engine, try:

\[
\text{drop database databasename;}
\]

\[
\text{create database databasename;}
\]

To examine the database directly, connect to it with:

\[
\text{use databasename;}
\]

• Most FQL programs omit “drop” statements. Hence, running the same script twice is likely to result in an error, because instances computed during execution of the FQL program will already exist. This can be avoided by use of the “drop” statement in the FQL program, or by creating a blank database as described above.

### 4.3 Compiling to Embedded Dependencies (EDs)

The FQL IDE can emit embedded dependencies (ED) that implement $\Delta$ and untyped (full or restricted) $\Sigma$ migrations. The are found in the “ED” GUI panes. $\Pi$ migrations cannot be so implemented. The IDE also contains a dialog for performing the chase on the emitted EDs.

### 4.4 Translating SQL to FQL

SQL schemas and instances in categorical normal form (CNF) can be treated as FQL instances directly. To be in CNF, every table must have a primary key column called `id`. This column will be treated as a meaningless ID. The SQL to FQL item in the tool menu translates CREATE TABLE and INSERT INTO statements into FQL programs. Columns not marked as foreign keys are treated as attributes.

Bags of tuples can be represented in FQL using an explicit active domain construction. Unions of conjunctive queries are supported, using DISTINCT and ALL for set semantics. Primary and foreign keys are not supported by this encoding. The RA to FQL item in the tool menu translates CREATE TABLE, INSERT INTO, SELECT FROM WHERE, and UNION statements into FQL programs. WHERE clauses cannot use equality between constants, only columns. UNIONs in RA are translated to co-products in FQL, although theoretically UNIONS of SELECT-FROM-WHERE be translated to FQL queries - sigmas of pis of deltas.
4.5 Translating Polynomials to FQL

Consider polynomials $p, q, r$ in variables $x, y$:

\[
p := x^2 + 2y^3 \quad q := x + 3xy \quad r := 2 + y
\]

Let $vars$ be the FQL schema with nodes $x, y$ and let $output$ be the FQL schema with nodes $p, q, r$. Then FQL can compute a query $q$ that computes the polynomials. For example, let $I$ be a $vars$-instance such that $|I_x| = 2$ and $|I_y| = 1$. Then $|q(I)_p| = 6$, $|q(I)_q| = 8$, and $|q(I)_r| = 4$. To do this, select “Polynomials to FQL” from the “Tools” menu.

4.6 RDF Representation of Schemas and Instances

FQL schemas and instances support a natural RDF representation. In the schema viewer, see the “RDF” tab for an RDF schema; note that path equations are not supported by RDF schema, but the domains and ranges of each FQL schema edge are represented. Similarly see the “RDF” tab in the FQL instance viewer for an RDF representation of instances.