Categorical Databases and Urbit

`datnut-pollen`

March 2, 2022

These are some informal notes on thoughts I've had related to categorical databases and Urbit over the past couple years. Writing this up was prompted by a discussion at the Mexico City offsite meeting and the suggestion that any substantive discussions we have ought to be documented.

I do not have any hard suggestions on how these concepts ought be utilized, so for now this is just something that might be interesting and valuable to pursue in the future.

Some potential use cases:

1. Composable state for composable Gall agents
2. A principled method for data migration between different versions of state that guarantees no degradation
3. A better `graph-store`
4. A model for how to conceptualize the entire state of the network, especially state shared among two or more nodes.

Use of category theory in actual code is not necessarily required for this - it can serve as a spec. As Urbit was not designed with category theory in mind, it is likely that not all of these ideas presented fit cleanly with existing concepts. But I believe that there are enough similarities, such as the fact that all entries in a categorical database have provenance just as all state and events in Urbit have provenance, that it is fruitful to understand this language even if we do not make any direct use of it.

I'd like to emphasize that compared to most other uses of category theory in programming, categorical databases are quite simple and grounded. So if you have found category theory too abstract before, this may be a good entry point.

Remark 0.1. This note was written in one sitting and had very little editing, so some parts of it may be sloppy. Some parts of it are straight-up plagiarized (such as the department store example). The only original contribution I claim here are the comparisons to Urbit.

1 Category theory

First let's give a quick refresher on what categories and functors are. We will subsequently present a model for databases that instantiates them as categories and functors.

Definition 1. *A* category $\mathcal{A}$ *consists of:*

1. *A set of objects* $\mathrm{Ob}(\mathcal{C})$
2. *For all* $A, B \in \mathrm{Ob}(\mathcal{C})$*, a set* $\mathrm{Hom}_{\mathcal{C}}(A, B)$ *of* morphisms *aka* arrows *or just* $\mathrm{Hom}(A, B)$ *when* $\mathcal{C}$ *is understood*
3. *Identity: For all* $A \in \mathrm{Ob}(\mathcal{A})$*, an identity morphism* $\mathrm{id} \in \mathrm{Hom}_{\mathcal{C}}(A, B)$
4. *Composition: For all* $A, B, C \in \mathrm{Ob}(\mathcal{A})$*, a function* $\circ : \mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) \to \mathrm{Hom}(A, C)$ *such that*

$$f \circ \mathrm{id} = f, \quad \mathrm{id} \circ f = f, \quad (f \circ g) \circ h = f \circ (g \circ h).$$

The category **Set** has sets as objects and functions as arrows. The "category" **Haskell** has types as objects and programs as arrows. The "category" **Nock** has nouns as objects and given any two nouns a, b, there exists an arrow $f_c : a \to b$ whenever there exists a noun c such that the product of `.*(a c)` is b.

For us, we will think of categories as roughly being a directed graph with objects as nodes and morphisms as arrows between nodes, with the ability to declare two paths through the graph equivalent (the composition rule).

Definition 2. *A* functor $F : \mathcal{C} \to \mathcal{D}$ *between two categories* $\mathcal{C}$ *and* $\mathcal{D}$ *consists of*

1. *A function* $\mathrm{Ob}(\mathcal{C}) \to \mathrm{Ob}(\mathcal{D})$*,*
2. *For all* $A, B \in \mathrm{Ob}(\mathcal{C})$*, a function* $\mathrm{Hom}_{\mathcal{C}}(A, B) \to \mathrm{Hom}_{\mathcal{D}}(F(A), F(B))$ *such that*

$$F(\mathrm{id}) = \mathrm{id}, \quad F(f \circ g) = F(f) \circ F(g).$$

The functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ sends each set to its power set. The functor $\mathrm{List} : \mathbf{Haskell} \to \mathbf{Haskell}$ sends each type t to the type $\mathrm{List}\, t$ (an example of a *monad*).

2 Databases

A database is a collection of tables, each table T which consists of a set of columns and a set of rows. The existence of a table T suggests the existence of a fixed methodology for observing objects or events of a certain type. Each column c in T prescribes a single method of observation, so that the datum inhabiting any cell in column c refers to an observation of that kind. Each row r in T has a fixed sourcing event or object, which can be observed using the methods prescribed by the columns. The cell (r, c) refers to the observation of kind c made on event r.

There are three important types of columns in tables: the *primary ID column* written on the left, some *data columns*, and some *foreign key columns*.

The primary key column uniquely identifies rows. This may be as simple as just enumerating them.

Data columns house elementary data of a certain sort.

Foreign key columns link one table to another, creating a pattern of connections between tables. Each foreign key column houses data that needs to be further unpacked. It thus refers to another *foreign table*, namely the primary ID column of some other table.

Example 2.1. We may think of an Arvo event log as a table with two columns: one column which enumerates the events $1, 2, 3, \ldots$ (the primary ID column), and one column which lists the events (nouns) (a data column). Thus each row contains a natural number and an event. From this table one may compute the state of an urbit by applying each event sequentially.

One could then extend the database to have three columns: the primary ID column, a data column for events, and another data column for the state. Then in a given row, you have the number of the event, an event, and the new state of Arvo after applying that event. Note that present here is also an equation that relates the event and state in a given row. We ignore this for now.

Example 2.2. Consider a Gall agent whose state has two variables `a=@` and `b=@`. We may think of this as a database with four columns: a primary ID column which enumerates `poke`s, a data column which contains the `poke`s, and two data columns representing the state `[a b]` obtained by applying the `poke` to the previous state. Again we have an equation that relates entries in the rows.

Gall agent state is derived from Arvo state. One thing you could do with categorical databases is give a functorial description of the mapping from the Arvo table to a given Gall agent table.

One could also imagine a foreign key column that lists the numerical identifier for Arvo events that contained the pokes that updated the Gall agent.

Subscriptions between Gall agents can also be represented with foreign key columns.

Example 2.3. The Azimuth PKI is another example of a table. Here the primary ID column is given by 32 bit integers, corresponding to ship identities. Data columns are given by `life`, `rift`, public key, and addresses for ownership and proxies. There are no foreign key columns as there is only one table.

3 Categorical databases

3.1 Schemas and instances

The structure of a categorical database is called a *schema*, while data in that database is called an *instance*.

Definition 3. *A* schema S *is a category* $[\![S]\!]$.

1. $\mathrm{Ob}([\![S]\!])$ *are the* nodes *of* S.
2. *For all nodes* A, B, $\mathrm{Hom}_{[\![S]\!]}(A, B)$ *is the set of finite paths* $A \to B$*, modulo the path equivalences in* S *(i.e. the equations for composition).*

Definition 4. *An* S*-instance is a functor* $I : [\![S]\!] \to \mathbf{Set}$.

Given $A \in \mathrm{Ob}([\![S]\!])$, $I(A)$ should be thought of as a table, so there is one table in the database for each object in S. Then associated to each edge in $[\![S]\!]$ (which recall is a *path* in S modulo equivalence, not an edge) is a column shared among the two tables associated to the nodes joined by the path (a foreign key column)

Let's consider some simple examples.

Example 3.1. Let $\mathcal{C}$ be a category with a single object and single identity morphism. An instance of $\mathcal{C}$ is then a single table, and the only path is given by the identity morphism, which is associated to the primary ID column of the table.

Example 3.2. Consider the bookkeeping necessary to run a department store. We keep track of a set of employees and a set of departments. We do this with two tables: one for employees and one for departments.

For each employee e, we keep track of

a The **first** name of e,

b The **last** name of e,

c The **manager** of e,

d The **department** that e works in.

For each department d, we keep track of

A The **name** of d,

B the **secretary** of d, which is an employee.

a, b, and A are data columns (referring to names), and c, d, and B are foreign key columns (referring to managers and secretaries).

We note two rules here:

1. For every employee e, the manager of e works in the same department that e works in.

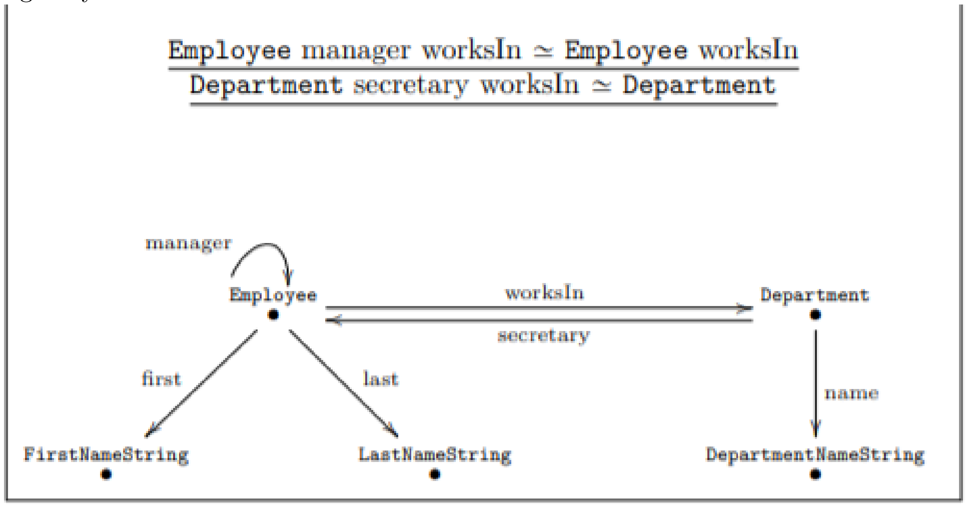
ID	first	last	manager	works in
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Emmy	Noether	103	q10

ID	**name**	**secretary**
q10	Sales	101
x02	Production	102

2. For every deparment d, the secretary of d works in department d.

These patterns can be formalized in the schema of the database as path equivalences as we will see shortly.

To do this, we will think of each data column as a specific kind of foreign key. That is, it is a foreign key to a *leaf table*, which has only one column and no additional data. So we consider three additional tables with one column each: one for first names, one for last names, and one for department names. Then our collection of tables has the property that every column is either a primary key or foreign key, so the notion of data column is subsumed under the notion of a foreign key column.



The five tables are seem as five vertices, which is also the number of primary ID columns. The six foreign key columns are seen as six arrows, each pointing form a table to a foreign tables. The two rules are seen as the statements written at the top, which are the *path equivalences* we've discussed. Thus we have an example of a database schema.

The particular table we populated is then an instance of this schema. The functor F from the above pictured category to the category of sets **Set** that instantiates these tables maps each node to the set of data in that table. For example, $F(employee)$ maps to the set of employee ID's $\{101, 102, 103\}$.

$F(department)$ maps to the set $\{Sales, Production\}$. The arrow $worksIn$ maps to the function that maps each employee ID to the department they work in, etc.

3.2 Functorial data migration

This model for databases is useful for a number of reasons, but one of the most important ones is how it suggests a mathematical encoding of database migration. Here we have two cases: the special case of adding rows to a database, and the general case of migrating a database schema to a new database schema.

Adding rows to a database can be thought of as a *natural transformation* between instances. Sometimes it is said that categories were defined in order to define functors, and functors were defined in order to define natural transformations. A natural transformation is essentially a functor for functors: a way of mapping one functor into another. For us, this will mean mapping one instance (a populated database) into a new one. This definition can look rather obtuse at first, but its also what you'd inevitably invent if you tried to define a mapping between functors.

Definition 5. *If F and G are functors $F, G : \mathcal{C} \to \mathcal{D}$, a* natural transformation *$\eta : F \to G$ is a family of morphisms η_A for $A \in \mathrm{Ob}(\mathcal{D})$ that satisfies the following:*

1. *For every $A \in \mathrm{Ob}(\mathcal{C})$, a morphism $\eta_A : F(A) \to G(A)$ between objects in $\mathcal{D}$.*
2. *For every morphism $f : A \to B$ in $\mathcal{C}$, $\eta_B \circ F(f) = G(f) \circ \eta_A$.*

This allows us to define a transformation between instances, such as one that adds a row to a table:

Definition 6. *Let $\mathcal{C}$ be a schema and $I, J : \mathcal{C} \to$ **Set** be instances on $\mathcal{C}$. A* morphism m from I to J, *denoted $m : I \to J$, is a natural transformation between those two functors.*

Because instances are just functors to set, the morphisms m_A for $A \in \mathrm{Ob}(\mathcal{C})$ are simply functions between sets $m_A : I(A) \to J(A)$.

Example 3.3. For instance, if we added an employee with ID 104, named Isaac Newton, working in the Sales department, to the table in the previous example, we'd have an injection from the set of IDs to the set $\{101, 102, 103, 104\}$, an injection from the set of first names to $\{David, Bertrand, Emmy, Isaac\}$, etc.

m is the an implementation of *functorial data migration* for the special case where the database schema (the underlying category) does not change.

Data migration between different schemas is given by functors between those schemas, which then induce natural transformations between instances of those schemas. This is a bit more complex so we do not cover it here, but this is how we would do things like have composable state for composable Gall agents.

Remark 3.4. The full formalism of functorial data migration is a bit more involved than this. Since I'm only aiming to get people interested in this and pointing out some instances where this way of thinking may be useful for Urbit, we're leaving out the heavy lifting.

Example 3.5. Adding an event to an Arvo event log or a `poke` to a Gall agent state might be able to be thought of as functorial data migration. Same with adding some new kind of state to a Gall agent.

Remark 3.6. Categorical databases are well-suited for typed data. Thinking of the **Haskell** example, where objects are types and arrows are programs, it is instructive to think about how altering data types in successive versions of state in e.g. a Gall agent meshes with functorial data migration. One of the main benefits of categorical databases is that they suggest a principled way to manage data migration that includes changes of type.

4 Simplicial databases

There are a number of interesting kinds of databases one can define using this categorical formalism. One idea I've been fond of for some time, but which is more of a stretch in terms of relevance to Urbit than what we've covered so far, is a categorical way of thinking of the data of the entire network using *simplicial databases*.

The Ames network of Arvo instances (which we'll just call the Arvo network) can be thought of as a collection of databases. For simplicity we pretend that key rotation is not a thing, and nobody ever needs to breach. We also assume that every ship has been booted, and the address space is only 32 bits (no moons or comets). So we're already in spherical cow territory, but I will make some remarks at the end about how these important details can be formalized in this model.

Each Urbit ship is itself a database. Each database contains some private data, but they also contain data intended to be shared with others via the pub/sub pattern that tries to keep that data identical across many nodes. For example, a ship hosting a chat channel sends each new message for that channel it receives out to every subscriber to that channel. Thus under ideal conditions, the state of the chat channel held by each ship that is a part of that channel is identical. Ships not subscribed to that channel will either not have any state associated to that channel at all, or an old version of the state if they were previously subscribed but then left.

The most coherent shared data is that held by blockchains. This is data that everyone on the network agrees upon, and consensus mechanisms are a way to ensure that the state held by every node in the blockchain is identical. For Urbit, this keeps the state of the `azimuth` Gall agent identical among every node. Federation, as with chat channels, is itself a form of consensus mechanism where everybody agrees that the host has the correct state and attempts to mirror that state. Programs like `git` and BitTorrent are another way to share state among nodes.

We'll start by summarizing the idea of a simplicial database in the context of Urbit, and then go a little bit into the formalism, followed by how I like to think of the network as a simplicial database.

4.1 Summary

We'd like a way to think about all these kinds of data sharing and synchronization among a network of databases under one mathematical roof.

We'll do this by constructing a high dimensional geometric object called a *simplicial set*. Roughly speaking, a simplicial set is a collection of points, line segments that join points, triangles that join together groups of three points that have lines connecting each pair, tetrahedrons that join together groups of four points that have lines and triangles between them, and so on upwards to higher dimensions. Each of these geometric objects is called a *simplex* - points are 0-simplices, line segments are 1-simplices, triangles are 2-simplices, tetrahedrons are 3-simplices, and so on.

To describe the Arvo network, we represent each ship as a point (0-simplex). Associated to that 0-simplex is the database (Arvo event log, or the current state of Arvo) held by that point.

If two ships share the same data, we draw a line (1-simplex) between their points. For example, if `zod` sends a DM to `nec`, both ships share the data of that DM and store it in their state somewhere. So this datum is associated both to the points associated to those ships in our simplicial set, as well as the line that joins them.

If three ships share the same data, such as with a group DM, then the datum of a message in that group DM session is associated to the points associated to the ships, the lines that join them, as well as a triangle (2-simplex) that joins all three.

Going up and up, the data of the PKI is held by a blockchain and thus is shared and synchronized among all nodes in the network. Thus this datum is associated to every point, every line, every triangle, and so on up to the highest dimension simplex in the simplicial set. Since we're only thinking about the 32-bit address space, this is then a $2^{32}-1$-simplex. Since there are only 2^{32} ships, this is also the highest dimensional simplex in the set.

Only state shared with and agreed upon by every ship is associated with the $2^{32}-1$-simplex. Furthermore, there is only one $2^{32}-1$-simplex. And all data associated to this top-dimensional simplex is also associated to all lower dimensional simplices. Since if everybody agrees on some datum, then e.g. that means that any pair of nodes agrees on it as well, so that datum is also associated to the line that joins them.

Example 4.1. Say we have a chat room hosted by A that has 10 members, including A. A message sent to this chat room is mirrored on each node. Thus that message is associated to each 0-simplex associated to the 10 members, each

1-simplex that joins each pair of members, each 2-simplex that joins any three of the members, and so on up to the 9-simplex that joins all members.

Remark 4.2. We should not think of our simplices as being located in any particular location in space. These are abstract entities that exist apart from any particular spatial embedding, and we only care about their connectivity. Thus this is more of a topological picture than a geometric one, but for the sake of visualization it is helpful to think of them as being geometric.

4.2 Formalism

The above geometrical picture can be formalized using categorical database theory. This last section can be left off on first reading for anyone not already familiar with category theory - it gets fairly technical.

Recall that a database is a system of tables which are connected together via foreign keys. This information is part of the schema for the database. We'll be keeping track of this information among a network of databases by making use of simplicial sets. Tables are connected together when the corresponding simplices are connected.

In order to give the precise definition, we need to dip into type theory.

Definition 7. *A* type specification *is a function between sets* $\pi : U \to \mathbf{DT}$*. The set* $\mathbf{DT}$ *is called the set of* data types *for* π*, and the set* U *is called the* domain bundle *for* π*. Given any element* $T \in \mathbf{DT}$*, the preimage* $\pi^{-1}(T) \subset U$ *is called the* domain *of* T*. An element* $x \in \pi^{-1}(T)$ *is called an* object of type T*.*

Example 4.3. I'm not sure how well this way of specifying types works with Hoon, but I haven't given it much thought and don't have much background in type theory. I think U is the power set of the set of nouns and $\mathbf{DT}$ is the set of all values of `type` (or maybe `spec`?). The `type` of `type` ought to specify π (but I'm not sure if this is unique).

For pedagogical purposes, it is easiest to think of `noun` as the only type for the time being. Then U is the set of nouns, and $\mathbf{DT}$ consists of a single element, `noun`.

Definition 8. *Let* $\pi : U \to \mathbf{DT}$ *denote a type specification. A* simple schema of type π *is a pair* (C, σ) *where* C *is a finite totally ordered set and* $\sigma : C \to \mathbf{DT}$ *is a function. We abbreviate* (C, σ) *with* σ*. We call* C *the* column list *for* σ *and* π *as the* type specification *for* σ*.*

Remark 4.4. The ordering on C does not matter too much and can be dropped if one so chooses. We use it since tables are displayed with a left-to-right ordering, and it leads to a more widely used definition of simplicial set.

We use a slight variation of the ordinary notion of simplicial sets found in algebraic topology.

Definition 9. *Let Δ denote the category of finite totally ordered sets, and let $\pi : U \to \mathbf{DT}$ be a type specification. Let $\mathcal{S}$ denote the category of simple schemas on π (note we have not defined what a morphism between simple schemas is - we're leaving it out in this note to keep things short). Then the* category of schemas on π*, denoted $\mathbf{Sch}^{\pi}$, is the category whose objects are functors $X : \mathcal{S}^{op} \to \mathbf{Set}$ and whose morphisms are natural transformations of functors.*

Remark 4.5. $\mathcal{S}^{op}$ is the *opposite category* of $\mathcal{S}$ - the category you get by taking $\mathcal{S}$ and reversing all arrows. The reason we use this is technical and outside our scope.

Remark 4.6. Given a category $\mathcal{C}$, the category whose objects are functors $\mathcal{C}^{op} \to \mathbf{Set}$ and whose morphisms are natural transformations of functors is called the *category of presheaves on $\mathcal{C}$*. The relationship between (pre)sheaves and consensus mechanisms in distributed networks is an active field of research outside the scope of this, but I'm happy to talk about it if you ask me.

Definition 10. *Let $X \in \mathbf{Sch}^{\pi}$ denote a schema. Given a simple schema $\sigma : C \to \mathbf{DT}$, the* σ-simplices *of X are the elements of the set $X(\sigma)$.*

I started writing out how my picture of the Arvo network in the summary fits into this formalism, but there's a number of aspects to it that don't seem to quite fit, and trying to hand wave past it may just be more confusing than convincing. I am reasonably certain that these obstacles are not show-stoppers, but working them out will take more time than I want to put into this at this point. The first and most obvious one is an n-simplex is associated to n columns, so that the $2^{32} - 1$-simplex has $2^{32} - 1$ columns. Since this is the simplex we'd like to associate to blockchain data such as the PKI, we probably want it to just be one column containing the noun that has the PKI state. You can get around this by taking the dual simplicial set, where instead there is only one 0-simplex and 2^{32} $2^{32} - 1$-simplices (one for each ship). This is a really common thing in math but probably isn't obvious that this is a sensible thing to do anyone who hasn't studied topology. There's numerous other issues, but at the end of the day this section was less intended as an actual mathematical formalism for Urbit itself and more as a sketch of how a network of databases for something Urbit-like could be modeled.

5 Conclusion

Some resources to read more:

Most of this stuff can be found at this portal for categorical informatics: https://math.mit.edu/ dspivak/informatics/

An introdutory textbook, Category Theory for Scientists http://math.mit.edu/ dspivak/CT4S.pdf

Actual software that makes use of categorical databases: https://categoricaldata.net/

The main references I referred to writing this:

Algebraic databases: https://math.mit.edu/ dspivak/informatics/CatData.pdf

Simplicial databases: https://math.mit.edu/ dspivak/informatics/SD.pdf

Functorial data migration: https://math.mit.edu/ dspivak/informatics/CD-FDM.pdf