

Visual Logic

as a universal interface for knowledge

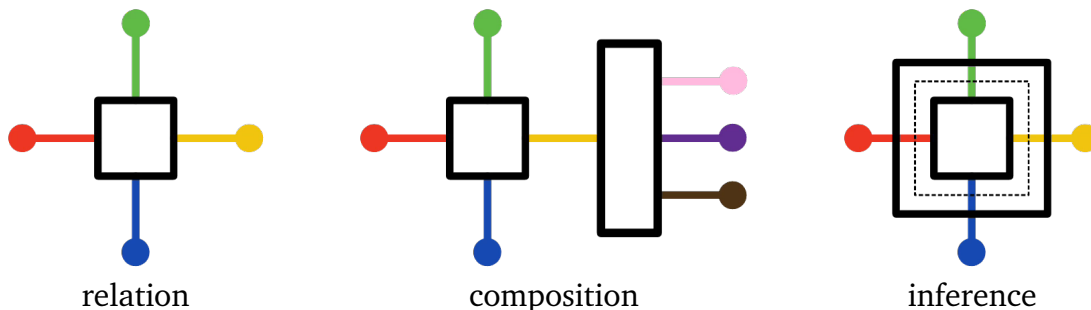
holdea.co

Introduction

Logic can empower us all, once it is simple to grasp and express and share.

Every kind of thinking shares the same general form: every thought expresses some *relation* of some types of things. Both in exploring nature, and creating society, we determine the world by forming **connections**.

Yet connections can be **visualized** in a simple way, as “nodes”. We make thoughts by *composing* relations, and we reason from thought to thought by *inferring* from relation to relation; this can be visualized as “nesting” one node inside another.



In this innocent idea lies great possibility: we can make a **universal interface for knowledge** — an infinite canvas for humanity — simply by seeing and holding thoughts as connections. The idea is to empower all people to explore and create.

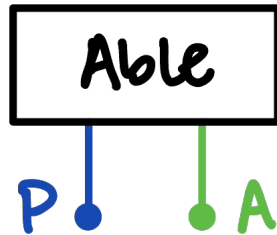
The growth plan is to leverage the immense present demand for data science: most structured information is stored in *relational databases*, in which a table is a relation of its column types; so, visual logic is a **simple and complete data tool**. All data activity — exploration, creation, management, and integration — can be expressed through simple interactions with tables as nodes.

Upon completing a PhD in Mathematics [2], I have formed the company **Holdea**. With cofounder Justin Pottenger, we are now developing the visual query tool, and we are seeking clients interested in a simple all-in-one data interface. Let’s meet!

cb@holdea.co, jp@holdea.co

Visualizing a Database

We can visualize a data type as a **wire**, and a table as a **node** that connects wires. For example, below is a table of people and their abilities.

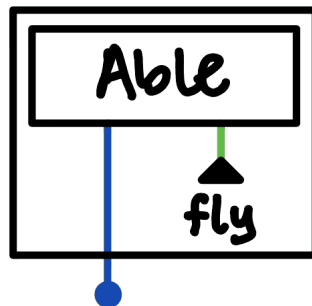


new “meta-view”

Person	Ability
Ben	juggle
Dalia	sing

ordinary “row-view”

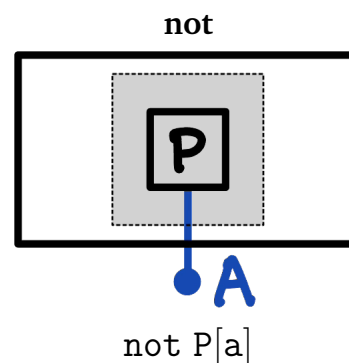
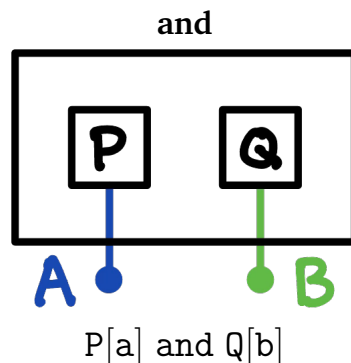
Why? Because this node is a *building block* for queries. In the simplest case, we can filter a table by plugging things into the wires.



Person
Margot
Sherman
Lilli

In general, a query is the same as a *predicate* in logic, such as “can fly a plane” — the results are the things which make the predicate true. And in fact, all predicates can be constructed out of just four basic operations, visualized as follows.

First, *conjunction* and *negation*: “and” is seen as placing nodes side-by-side; while “not” is seen as alternating between two opposite background colors.

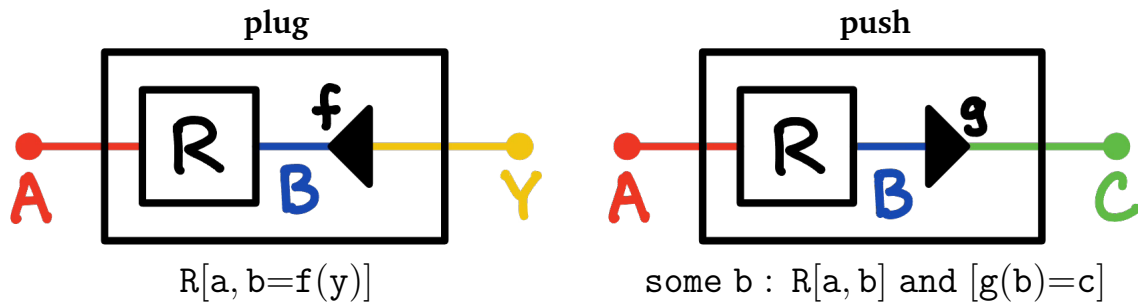


For example, the conjunction “Animal a is an herbivore and Plant b has vitamin K”, and the negation “Animal a is not an herbivore”.

These visual forms reflect our simple intuitions of considering multiple things, and considering the opposite of something.

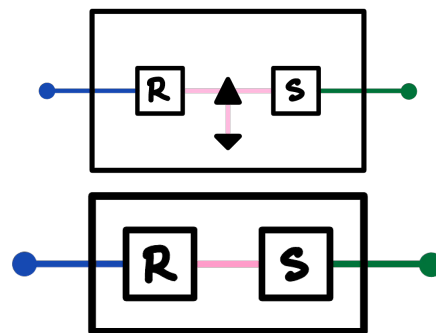
For the second half of logical constructors, *functions* can operate on predicates, i.e. programs can operate on tables of data.

Given a table R with columns A, B , we can *substitute* or “plug” a program $f : Y \rightarrow B$ into a column; or we can form the *image* or “push” the table by a program $g : B \rightarrow C$. Below the visualizations are the formulas of $f^*R[a, y]$ and $g!R[a, c]$, respectively.



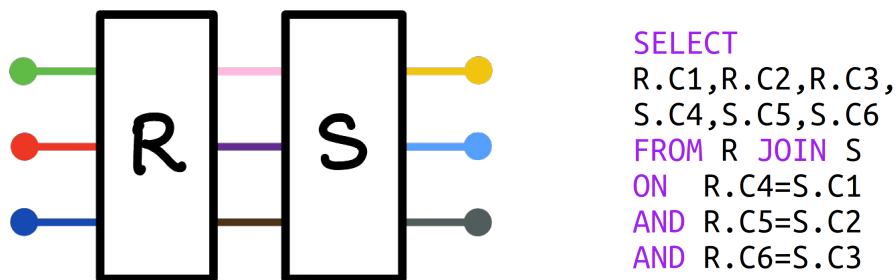
As a special case, a thing $b : B$ is the output of a program with no input. And often, we use the program g that *deletes* the thing, $g(b)=*$, drawn as whitespace; so in this case, we form the relation $[\text{some } b : R[a, b]]$ on things of type A .

In particular, we *compose* relations by “and, plug, some”: conjoin the relations, match the middle variable, and then sum. As the basic way we build connections, we can visualize composition simply as *connecting wires*.

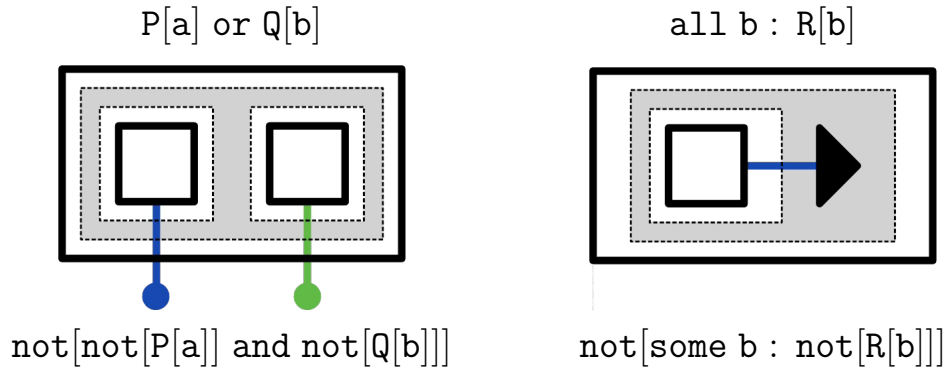


$$RS[a, c] = \text{some } b : R[a, b] \text{ and } S[b, c]$$

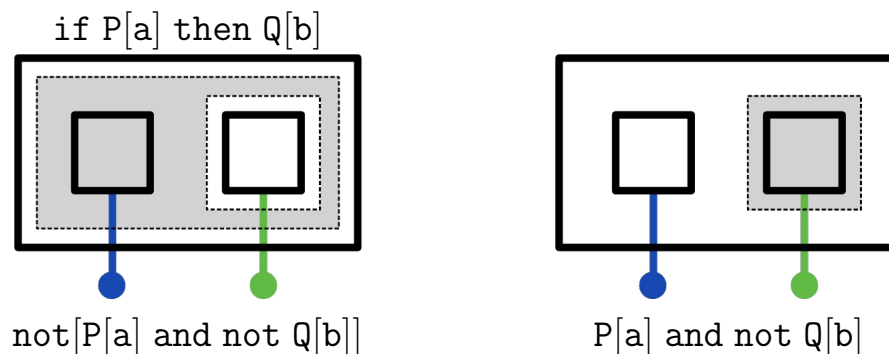
This is far more intuitive and expressive than in the Standard Query Language: all the millions of lines of “select-join” code can be left to the computers.



Relations and composition, made from “and” and “some”, form the part of logic known as *regular logic*. By combining with negation, we form the *opposite half* of logic: disjunction “or”, and universal “all”. [1]



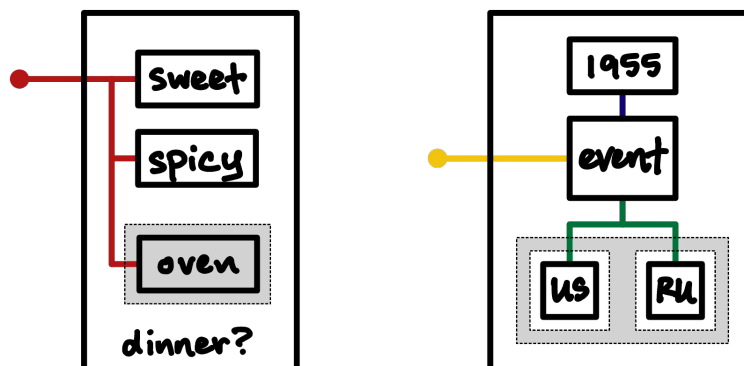
The other two basic operations are *implication* and *difference*.



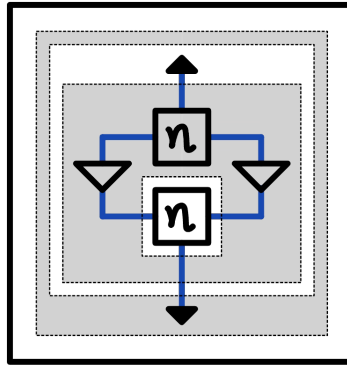
So, we can express all of **first-order logic** from just “and, not, plug, some”.

This provides a lot; but **higher-order logic** includes our ability to see a *thought* as itself being a type of *thing* which we can think about. In databases, **aggregations** input tables rather than rows. This ability can be visualized as well (see appendix), providing complete expressiveness for all data operations.

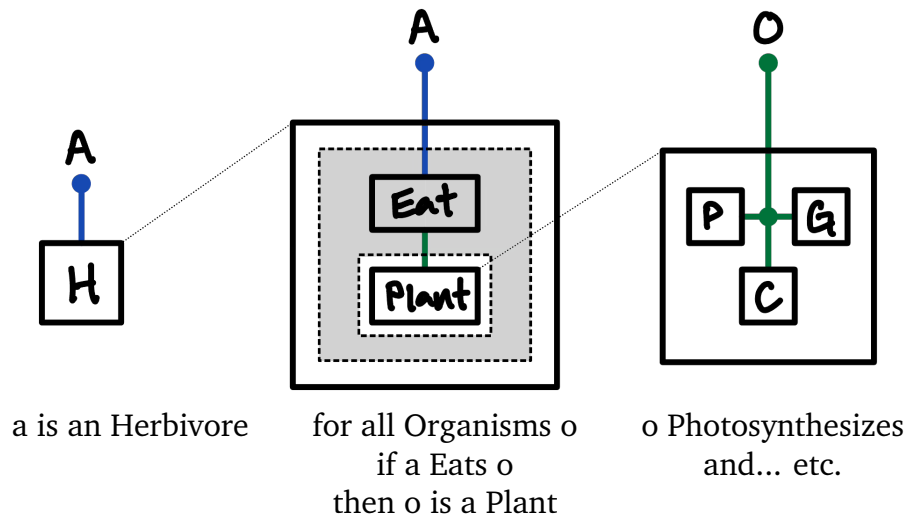
All our thoughts, in every sphere of life, we form out of these simple operations. They can be as simple as “what’s for dinner?” or as complex as world history;



or a basic notion, like *continuity* — read from outer to inner: for all distances e , there is some distance d so that if inputs are d -near, then outputs are e -near.



For us to grasp complex relations, we naturally view concepts in *layers of detail*. To learn the concept of Herbivore, for example, we can *expand* the herbivore node, to display its definition. Then we could ask “what is a Plant?” and expand *that* node — the visual logic interface can provide limitless exploration of knowledge.



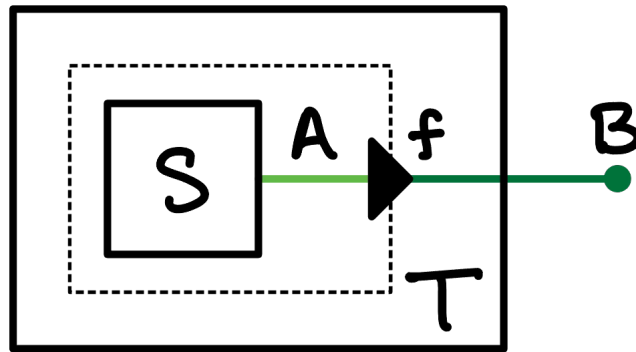
In computation, this is known as **modularity**. Any thought, no matter how complex, can be expanded to its full definition, or collapsed to a single node and built upon.

So far, this presents visual logic as a **query language**. Yet *exploring* knowledge is only one aspect; we also need to *create* and *update* knowledge.

Updates and Migrations

Things change — as a **process** $f : A \rightarrow B$ transforms a type of thing into another, we can make **inferences** from A-thoughts to B-thoughts; this can be visualized as *nesting* a source node S into a target node T , along the process f from A to B .

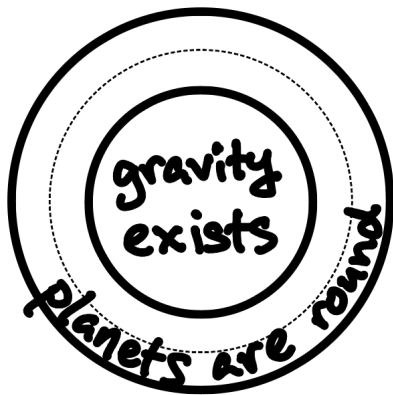
Below is the rule: if we know S of $a : A$, then we know T of $f(a) : B$.



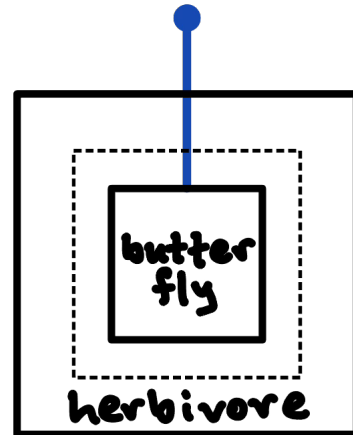
$$S[a] \vdash T[f(a)]$$

In a database, f is some **program** which modifies data, and the inference is an **update rule**: if a is a row in table S , then $f(a)$ is a row in table T .

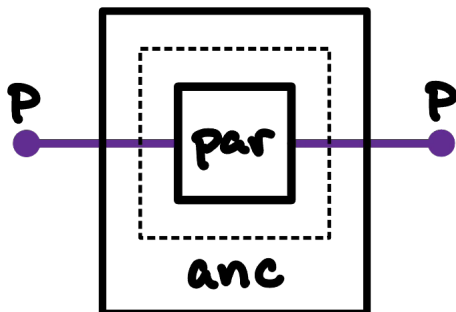
Here are examples where the things stay the same, as in $f(a) = a$.



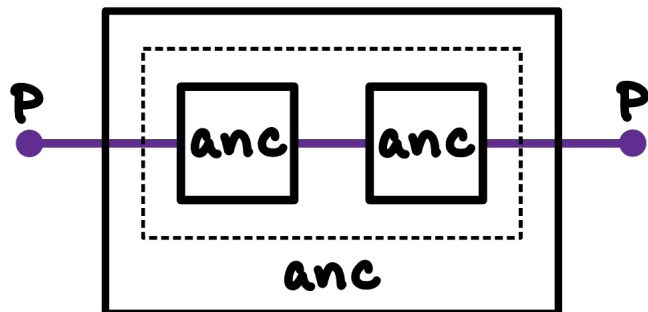
if gravity exists,
then planets are round.



if Animal a is a butterfly,
then a is an Herbivore.

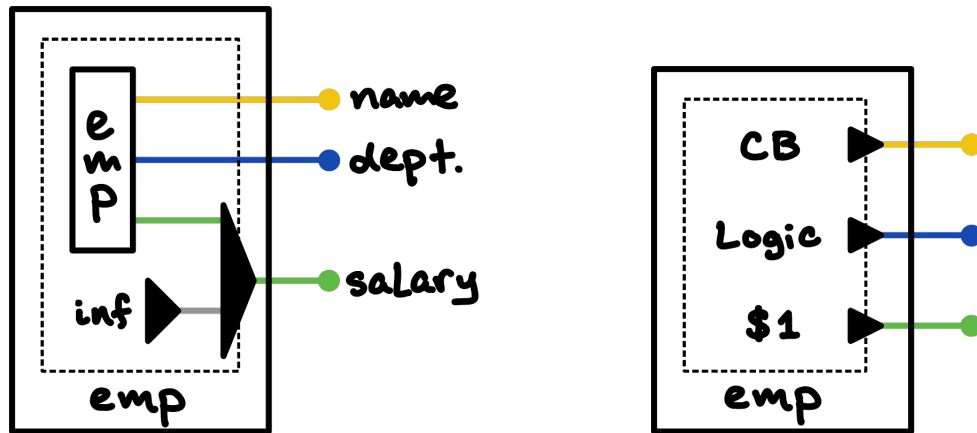


parent \vdash ancestor



ancestor of an ancestor \vdash ancestor

For a business example, suppose a company should adjust wages for inflation. This can be automated by a simple rule from and to the employee table (at left).



The source can have any conditions for when to execute, such as $\text{date}() = \text{Jan } 1$.

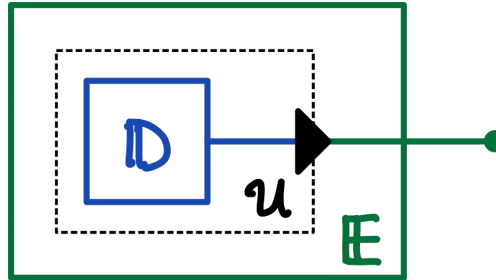
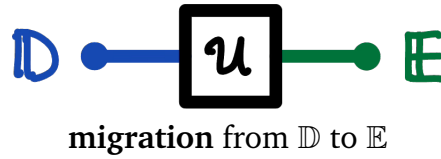
In particular, *creating* new knowledge is making rules with no preconditions. Filling a row b in a table T (at right) is asserting “if true, then we know T of $b()$ ”.

So, a database **schema** (a *logic*, or an *ontology*) is a system of tables and rules; and an **instance** (the actual *data* of a database) is a system of rules “from true”.

In full generality, we can form any system of rules between any pair of logics — this is known a **migration** between databases, and could be called a *transference* between logics. To see this higher view, we can “zoom out” to *the logic of logics*. (This is the main new idea of the author’s doctoral thesis [2].)

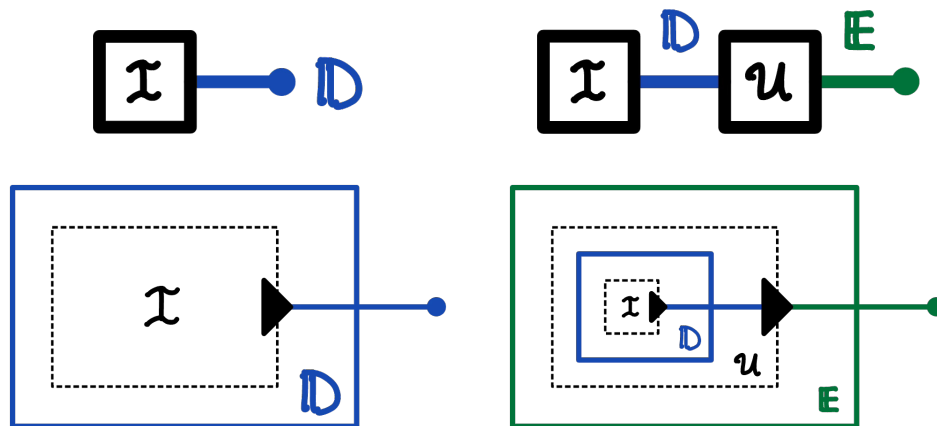
As this topic is more advanced, the next page can be skipped. We include it for completeness, to emphasize that visual logic encompasses *every* kind of data activity, including communication across databases.

In “metallogic”, a type is a whole logic (database), and a relation \mathcal{U} from \mathbb{D} to \mathbb{E} is a *transference* or **transfer**: a system of processes (programs) from types $\mathbb{D}:\mathbb{D}$ to types $\mathbb{E}:\mathbb{E}$, and inferences (updates) from relations (tables) in \mathbb{D} to those in \mathbb{E} .



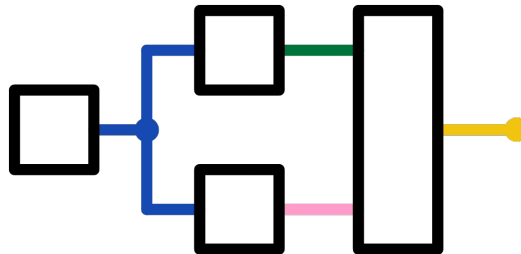
This is a migration between databases, or an “extract, transform, load” workflow; and this includes any **database update**: the logic \mathbb{E} could be the same as \mathbb{D} , or it could be \mathbb{D} plus some new tables, which are populated by the rules of \mathcal{U} .

Below is an instance \mathcal{I} of schema \mathbb{D} , and the application of an update.

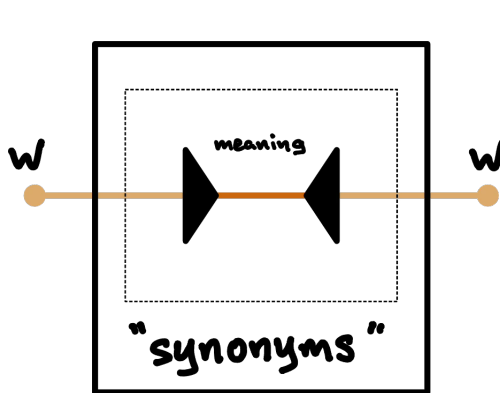


In an updated instance, each datum has a whole **history**, which can be queried.

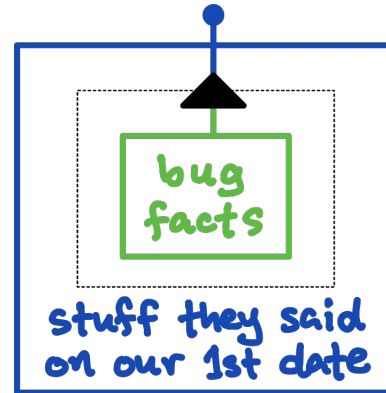
From the bird’s-eye view, we can easily see and manage *branching* updates, and we can *recombine* those branches; this encompasses all kinds of **data integration**.



We create all kinds of logics, of all kinds of worlds; and we share and grow by relating our logics. So transfers are *everywhere*: **learning**, updating one's own logic, or **communicating**, one person updating their logic in response to someone;

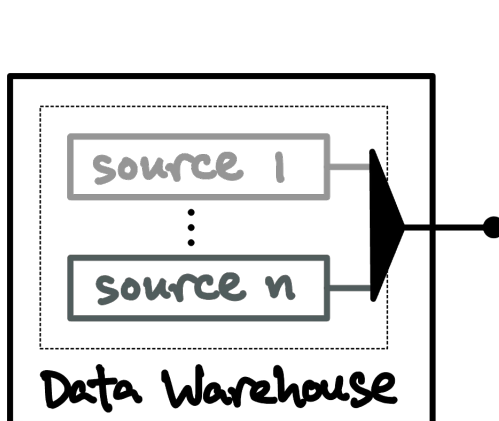


learning a new word

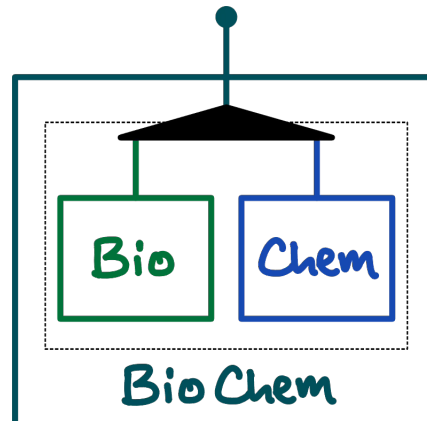


meeting a new friend

or **integrating** our knowledge — vital for most human endeavors.



Any company managing multiple data sources needs systematic migrations.



Some whole sciences can be understood as developing integrations.

Of course, such large-scale unification requires *interoperability*; so the interface must support many languages and systems. This is a large yet well-defined task.

As may be clear by now, the uses of a logical interface cannot be summarized, because logic underlies every aspect of human life. You can organize your thoughts, learn a new subject, run a business, design a social contract, imagine a new world. Visual logic is a canvas, meant to empower and connect humanity.

The development of the interface is now underway, by our new company Holdea. If this project interests you, reach out to us at cb/jp@holdea.co. Thank you.

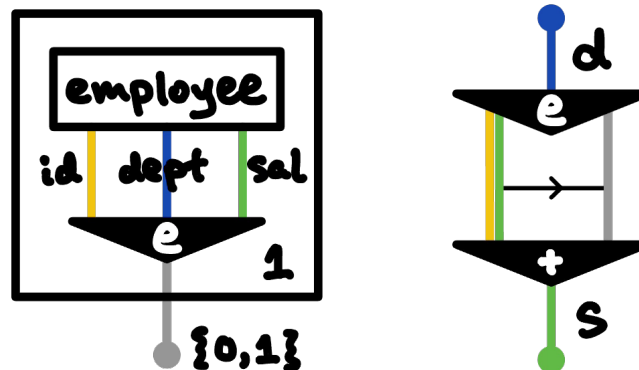
References

- [1] Filippo Bonchi, Alessandro Di Giorgio, Nathan Haydon, and Pawel Sobocinski. Diagrammatic algebra of first order logic, 2024. Available at <https://arxiv.org/abs/2401.07055>.
- [2] Christian Williams (now CB Wells). *The Metalanguage of Category Theory*. PhD thesis, University of California, Riverside, 2023. Available at <https://escholarship.org/uc/item/84j4z67h>.

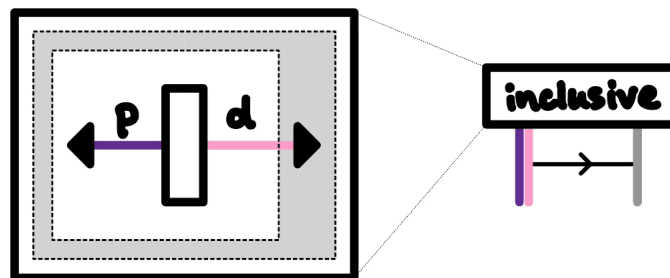
Appendix

In data science, an *aggregate function* inputs a whole table rather than one row, such as summing many values, or returning the highest value, or counting rows.

Converting a relation into a type can be visualized: we can see a table as its truth function, and “bend wires backward” to form a family of tables. For example, below is the query: “for each department, show the total salary of its employees”.



This forms the basis of *higher-order logic*, the complete language which encompasses both “thinking” and “thinking about thinking”. It goes well beyond aggregation — as a brief (and oversimplified) example, we could define a table of People to be Inclusive if all Demographics are represented.



Just by leaving the table name *blank*, we define a property of all tables of that type. We can reason about our systems, to ensure they are safe, effective, democratic, etc.