Categorical Data Structures for Technical Computing

Evan Patterson[1], Owen Lynch[2], and James Fairbanks[3]

[1]Topos Institute, California, USA

[2]Universiteit Utrecht, Mathematics Department, Utrecht, The Netherlands

[3]University of Florida, Computer & Information Science & Engineering, Florida, USA

Many mathematical objects can be represented as functors from finitely-presented categories C to Set. For instance, graphs are functors to Set from the category with two parallel arrows. Such functors are known informally as C-sets. In this paper, we describe and implement an extension of C-sets having data attributes with fixed types, such as graphs with labeled vertices or real-valued edge weights. We call such structures *acsets*, short for *attributed* C*-sets*. Derived from previous work on algebraic databases, acsets are a joint generalization of graphs and data frames. They also encompass more elaborate graph-like objects such as wiring diagrams and Petri nets with rate constants. We develop the mathematical theory of acsets and then describe a generic implementation in the Julia programming language, which uses advanced language features to achieve performance comparable with specialized data structures.

Contents

Evan Patterson: evan@epatters.org, https://www.epatters.org
Owen Lynch: o.c.lynch@students.uu.nl, https://owenlynch.org
James Fairbanks: fairbanksj@ufl.edu, https://jpfairbanks.com

1 Introduction

Practicing data scientists commonly say that they spend at least half their time cleaning data and managing data pipelines, rather than fitting models. The inherent difficulties of data preparation are exacerbated by the multitude of ways in which data can be stored. Data can live in SQL databases, data frames, graphs, or any of the more specialized data structures scattered throughout the data science ecosystem. While each data structure may have its own advantages, it also comes with a new programming interface to learn, and this proliferation fragments the ecosystem and creates challenges for interoperability.

One solution to this problem has been to centralize around a single data structure: the data frame. A data frame is a column-oriented data table, which, unlike a matrix, can have columns with different types. The Python ecosystem has the popular package `pandas` [15]; R has the built-in data type `data.frame` [20] and its cousin `tibble` [17]. In Julia, many different packages implement the common interface of `Tables.jl`, which helps somewhat with interoperability.

Underlying all these packages, however, is the same abstract data model, with all its limitations. Importantly, the data frame model fails to capture *relations* between entities, a concept realized in SQL as the `FOREIGN KEY`. Of course, it is possible to maintain a collection of data frames that refer to each other according to an ad hoc convention, but such relationships are not formalized and thus cannot be manipulated conveniently or robustly through high-level abstractions. As a prototypical example, data frames do not properly capture the structure of a graph, which requires two interlinked data tables, one for vertices and one for edges.

A well-known abstraction that encompasses both data frames and graphs is the relational database. Relational databases have schemas that describe a collection of relations and the foreign keys that link them together. However, relational databases tend to be monolithic systems that are difficult to integrate with general-purpose programming languages. Relational query languages like SQL and logic programming languages like Prolog and Datalog suffer from the Achilles' heel of all standalone domain-specific languages (DSLs): one cannot step outside the DSL without abandoning the system entirely. Yet stepping outside the DSL is often necessary, since even the most expressive query languages have their limitations. Therefore, it becomes a perennial question how much logic to put into the query and how much to put into postprocessing outside the database: a "two-language problem" for data processing. Furthermore, databases are typically designed to be treated as global mutable state, making it unnatural to use them as disposable objects in a localized context. For example, graphs can be modeled as databases, but while a program could easily have thousands or millions of graphs extant at a given time, existing in different scopes, one would not wish to achieve this using a million SQL database instances.

Yet there is no essential reason that the idea of "a collection of tables tied together by a schema, along with indices" should require monolithic systems, persistence, or global scope. The earliest papers on relational databases already possessed a mathematical model, based on relational algebra and first-order logic, that was independent of any conventions about implementation [8]. More recently, Spivak recognized that the relational data model could be elegantly represented as a functor from a finitely-presented category to the category of sets [26]. More elaborate categorical models accommodating data attributes were developed later [27, 25].

In this paper, we present an efficient in-memory implementation of categorical databases. Depending on the schema, the resulting data structure can act as a data frame, a graph, or any of a multitude of other structures, previously regarded as too niche to have dedicated implementations. Using advanced features of the Julia programming language, our implementation achieves performance competitive with state-of-the-art graph libraries, despite the fact that our graph library is a thin wrapper around a much more general system.

We call our data structures "acsets," an abbreviation of "attributed C-sets." We define acsets in a direct, practical manner but show that they can be reformulated as well-understood objects from categorical algebra. From this mathematical picture, we derive operations to translate between acsets based on different schemas, combine acsets together, query acsets, and algorithmically create and manipulate schemas. For instance, we can generically compute finite limits and colimits of acsets on a given schema. We emphasize, however, that for most purposes, knowledge of category theory is not required to use acsets.

Our impetus for developing acsets originated with the needs of Catlab and other packages in the AlgebraicJulia ecosystem [18]. While implementing various pieces of applied category theory, we realized that many of the data structures we needed were captured, at least partially, by C-sets (copresheaves), which we could implement generically. But this abstraction was not completely satisfactory, because it did not account for attributes: data with a fixed, external meaning such as real numbers or strings of text. This eventually led to us the formalism of acsets and to a more systematic software implementation.

Contributions Our main contribution is an efficient, flexible implementation of categorical databases as in-memory data structures in a general-purpose programming language, supporting key constructions of applied category theory, including decorated or structured cospans. Our implementation, the first of its kind, takes advantage of metaprogramming features of the Julia language to attain performance comparable with specialized, state-of-the-art graph libraries, while being far more general. In a more theoretical vein, we introduce a variant of Spivak et al's functorial data model, intermediate in complexity between those in [27] and [25], and we derive its basic mathematical properties.

Outline The paper begins, in Section 2, with an informal overview of acsets that should be accessible to a general audience of computer scientists and software engineers. We then review the mathematical theory of C-sets, of which acsets are an elaboration, in Section 3. Readers familiar with the relevant mathematics may omit this section. In Section 4, we develop the theory of attributed C-sets, showing that acsets are slice objects in the category of C-sets and deriving consequences of this fact. Finally, in Section 5, we discuss the implementation of acsets and benchmark it against `LightGraphs.jl`, a state-of-the-art graph library written in Julia [4], lending empirical support to our claim that acsets can simultaneously achieve generality and performance.

2 Using Attributed C-sets

In this section, we aim to convey an intuitive understanding of acsets. We first explain how two common data structures, data frames and graphs, are special cases of acsets. To illustrate the breadth of the formalism, we also give a short tour of more exotic, yet useful, acsets.

2.1 Data Frames and Graphs

As discussed in the Introduction, data frames are a popular answer to the question of how we should store our data. Table 1 shows a tiny data frame with two columns, `a` and `b`.

`a`	`b`
3	0.2
2	0.0
2	0.0
1	0.9

Table 1: Example of a data frame

For the purposes of this paper, we regard a *data frame* to be a collection of 1-dimensional arrays, called *columns*, all of the same length. A *row* in a data frame consists of the values of all columns at a given integer index. Individual columns can be retrieved by name, and individual rows can be retrieved by index. Data frames are typically stored *column-wise* (as a list of columns) rather than *row-wise* (as a list of rows) to permit efficient data access and iteration.

We will build acsets as an extension of data frames, so we take for granted an implementation of data frames in Julia. In this notional implementation, `DataFrame{(:a,:b),Tuple{Int,Float64}}` is the type of a data frame with two columns named `:a` and `:b` and having the types `Int` and

Float64. We can construct and inspect the above data frame as follows, accessing the data both row-wise and column-wise.

```
> df = DataFrame(a=[1,2,2,3], b=[0.2,0.,0.,0.9]);
> df.a # first column, called `a`
[1,2,2,3]
> df[2,:] # second row
(a=2,b=0.)
```

As motivated in the Introduction, we will eventually wish to regard a graph as two interlinked data frames. However, before turning to that strategy, we consider how graphs are typically implemented. Perhaps the simplest way to implement a graph in Julia would be to use the *edge list* data structure:

```
struct EdgeList
  vertices::Int
  edges::Int
  src::Vector{Int}
  tgt::Vector{Int}
end
```

An object `g` of type `EdgeList` defines a graph, where

1. the vertices in the graph are the consecutive numbers `1:g.vertices`,
2. the edges in the graph are the consecutive numbers `1:g.edges`, and
3. the source of edge `j` is `g.src[j]` and the target of edge `j` is `g.tgt[j]`.

In particular, the invariant `length(g.src) == length(g.tgt) == g.edges` should be maintained. Just as for data frames, we store `src` and `tgt` as two separate vectors, rather than as a single vector of pairs.

A common variant of the edge list is the *adjacency list*. It stores the inverse images of the source and target maps, which, to use the jargon of databases, index the edges incoming to or outgoing from each vertex.

```
struct AdjacencyList
  vertices::Int
  edges::Int
  src_index::Vector{Vector{Int}}
  tgt_index::Vector{Vector{Int}}
end
```

The interpretation of an object `g` of type `AdjacencyList` is that for each vertex `i`, the list of edges with source `i` is `g.src_index[i]` and the list of edges with target `i` is `g.tgt_index[i]`. These indices are useful for traversing the graph, as they enable rapid iteration through the neighbors of any given vertex.

However, in the adjacency list format, it is less convenient to iterate through the edges of the graph and retrieve their sources and targets. Therefore, it can be useful to have both the `src` and `src_index` fields. The trouble now becomes that when modifying the graph, careful bookkeeping is required to ensure that all the fields remain consistent with each other. A useful feature of our implementation of acsets is automatically generating code to deal with such indices. Correct and efficient implementation of this bookkeeping is a significant task when implementing new mutable data structures.

2.2 Towards Attributed C-sets

There is an essential difference between graphs and data frames. One can permute the vertices of a graph, and as long as the source and target maps are updated accordingly, the meaning of the graph does not change. However, in a data frame, if one were to exchange every occurrence of the number 6.0 with the number 2.0, then the meaning of the data has changed drastically.

This distinction is crucial to understanding acsets, so it is worth introducing some informal terminology. We call the connectivity data stored in a graph "combinatorial data" and the kind of data stored in a data frame "atomic data" or "attribute data."

In data science and scientific computing, we routinely encounter datasets involving both combinatorial and atomic data. For instance, suppose that we want to represent a network of roads. We could store a graph where each vertex (road junction) has associated x and y coordinates, and each edge (road) has a length, which could be different from the Euclidean distance between its endpoints due to bends and hills. A straightforward implementation of the corresponding data structure might be:

```
struct MessyRoadMap
  vertices::Int
  edges::Int
  coords::DataFrame{(:x,:y), Tuple{Float64,Float64}}
  lengths::DataFrame{(:length,), Tuple{Float64}}
  src::Vector{Int}
  tgt::Vector{Int}
  src_index::Vector{Vector{Int}}
  tgt_index::Vector{Vector{Int}}
end
```

Organizing the fields more systematically, in our implementation of acsets this data structure appears as:

```
struct OrganizedRoadMap{T}
  tables::NamedTuple{(:vertex,:edge),
    Tuple{
      DataFrame{(:x,:y), Tuple{T,T}},
      DataFrame{(:src,:tgt,:length), Tuple{Int,Int,T}}}}
  indices::NamedTuple{(:vertex,:edge),
    Tuple{
      DataFrame{(:src,:tgt), Tuple{Vector{Int},Vector{Int}}},
      DataFrame{(),Tuple{}}}}
end
```

If `i` is the index of a vertex in an object `g` of this reorganized type, then to access `i`'s x-coordinate, we write `g.tables.vertex.x[i]`. Similarly, the source of an edge `j` is `g.tables.edge.src[j]`. The edges incoming to a vertex `i` are given by `g.indices.vertex.tgt[i]`.

This way of organizing the data is evidently not at all specific to road maps. The relevant features of the road map are:

1. There are two "combinatorial types": vertices and edges. We write these in mathematical notation as V and E. In an instantiation of a graph, we will assign finite sets to V and E.
2. There are two maps from edges to vertices, which we write as $\mathrm{src}\colon E \to V$ and $\mathrm{tgt}\colon E \to V$.
3. There is a single "attribute type" T. In an instance of a road map, we will assign a Julia type to T, typically some numeric type.
4. There are "data attribute" maps $x\colon V \to T$, $y\colon V \to T$, and $\mathrm{length}\colon E \to T$.
5. The maps src and tgt are indexed.

Items 1 and 2 comprise the combinatorial data of the road map and items 3 and 4 the atomic data. In our implementation, combinatorial data is always represented by integers, whereas atomic data is represented by type parameters and may be filled by any Julia type. Note that the final item 5 is not like the others; it is required for efficiency, but we could have omitted the index and the data structure would still contain the same information. For this reason, when we describe signatures formally, we will omit indices. Instead, the indices can be chosen at compile time based on the anticipated workload. Only keys that are frequently queried should be indexed.

In our implementation in Catlab, we generate a data type like `OrganizedRoadMap` by writing down a formal specification of the schema and then passing this specification to a function that programatically generates the data structure for the corresponding acset:

```
@present TheoryRoadMap(FreeSchema) begin
  (V,E)::Ob
  (src,tgt)::Hom(E,V)

  T::Data
  (x,y)::Attr(V,T)
  length::Attr(E,T)
end

@acset_type RoadMap(TheoryRoadMap, index=[:src,:tgt]) # creates RoadMap type
```

$$\begin{tikzcd} E \arrow[rr, bend left, "src"] \arrow[rr, bend right, "tgt"'] \arrow[dr, "length"'] & & V \arrow[dl, bend right, "x"'] \arrow[dl, bend left, "y"] \\ & \mathrm{T} & \end{tikzcd}$$

The elements in the schema typed by `Ob` and `Hom` describe the combinatorial data and the elements typed by `Data` and `Attr` describe the atomic data. The schema is also displayed as a diagram to the right of the code.

Graphs and data frames can now be treated as special cases of this machinery, one involving only combinatorial data and the other involving only atomic data. The schema for graphs is:

```
@present TheoryGraph(FreeSchema) begin
  (V,E)::Ob
  (src,tgt)::Hom(E,V)
end
```

$$\begin{tikzcd} E \arrow[r, bend left, "src"] \arrow[r, bend right, "tgt"'] & V \end{tikzcd}$$

The schema for `DataFrame{(:a,:b),Tuple{T,T}}`, a data frame with two columns of the same type, is:

```
@present TheoryAB(FreeSchema) begin
  Sample::Ob
  T::Data
  (a,b)::Attr(Sample,T)
end
```

$$\begin{tikzcd} \textbf{Sample} \arrow[d, bend right, "a"'] \arrow[d, bend left, "b"] \\ \mathrm{T} \end{tikzcd}$$

Defining data structures is all well and good, but how does one use them? To illustrate, we show a simple function that generates a road map whose underlying graph is a path graph.

```
function make_path(coords::Vector{Tuple{Float64, Float64}})
  # Create an empty roadmap
  path = RoadMap{Float64}()

  # This is a convenient function that calculates the Euclidean distance between two
  # vertices in the road map. Notice that we can reference attributes using indexing
  # and that the system knows that these attributes belong to vertices, not edges.
  dist(i,j) = sqrt((path[i,:x] - path[j,:x])^2 + (path[i,:y] - path[j,:y])^2)

  x, y = coords[1]
  # add_part! mutates path to add a part, returning the index of the added part.
  # The named arguments to this function assign the attributes of that part.
  src = add_part!(path, :V, x=x, y=y)

  for i in 2:length(coords)
    x, y = coords[i]
    tgt = add_part!(path, :V, x=x, y=y)
```

```
      add_part!(path, :E, src=src, tgt=tgt, length=dist(i,j))
      src = tgt
    end
    path
  end
```

While there are also higher-level functions on acsets, the low-level accessors and mutators are always available. Moreover, they are *fast*, as demonstrated by the benchmarks in Section 5, so the user is not constrained by what the high-level interface exposes. Coupled with the ability of Julia to make hand-written loops as efficient as "vectorized" code, users can easily write high-performance algorithms that are unanticipated by the core library.

2.3 Beyond Graphs: Wiring Diagrams and Other Graph-like Structures

Objects similar to graphs but possessing extra or different structure occur frequently in computer science. To create custom data structures for each graph variant by hand would cause an explosion of software complexity, yet without custom data structures, it is not possible to efficiently utilize the extra mathematical structure. In this section, we show that many graph-like structures are unified by the concept of an acset and thus can be manipulated through a uniform, general software interface.

Each of the following three examples are accompanied by figures, which are too large to be displayed inline. The reader is encouraged to contemplate the figures spread over the next several pages before returning to the main text for an explanation.

Example 1. Figure 1 shows four different kinds of graphs with ports, or "port graphs," and the generators of their schemas. The port graphs differ along two axes: untyped versus typed and circular versus directed. In a typed port graph, the types of incident ports and wires must agree. This requirement is expressed by the equation

$$\mathrm{src} ⨟ \mathrm{ptype} = \mathrm{wtype} = \mathrm{tgt} ⨟ \mathrm{ptype}$$

for circular port graphs, and

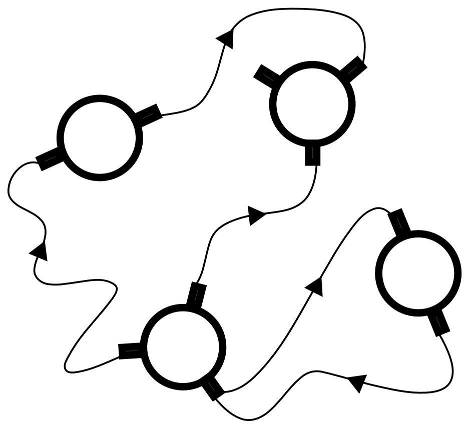
$$\mathrm{src} ⨟ \mathrm{optype} = \mathrm{wtype} = \mathrm{tgt} ⨟ \mathrm{iptype}$$

for directed port graphs, which assert that certain triangles in Figures 1b and 1d commute. Untyped port graphs have no such requirement. In a circular port graph, the boxes have only kind of port, **Port**, although the wires are directed. In a directed port graph, the ports are split into input ports (**InPort**) and output ports (**OutPort**), and the wires must go from input ports to output ports. By convention, input ports are drawn on the left and output ports on the right, so that the directions of the wires can be inferred from the incident ports.

Example 2. A *whole-grained Petri net* consists of species, transitions, inputs to transitions, and outputs from transitions [12]. We visualize Petri nets in Figure 2, where, by tradition, the species are drawn as circles and the transitions as squares. Petri nets are often augmented with a set of tokens for each species. This is accomplished by adding a new object **Tok** to the schema, along with a map from tokens to species. This trick of representing a many-to-one relationship via a map is a common one when modeling data with relational algebra. Note that the schema for a whole-grained Petri net is isomorphic to the schema for a directed bipartite graph, to which we return in Example 6.

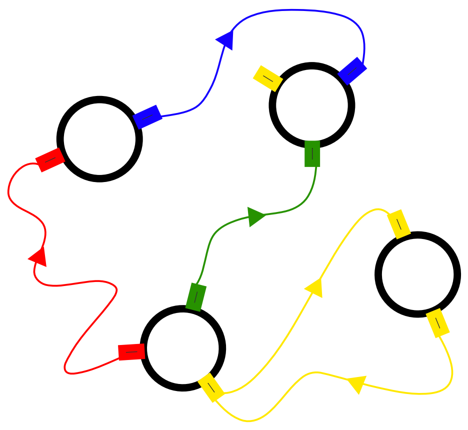
Example 3. Schemas for typed and untyped *undirected wiring diagrams* are pictured in Figure 3. Loosely speaking, undirected wiring diagrams represent patterns of composition for systems with an outer boundary. Imagine placing an entire wiring diagram inside one of the inner circles, and then erasing the inner circle to get a new wiring diagram. This operation makes undirected wiring diagrams into an operad, a construction made precise in [23].

We hope that these examples have convinced the reader that expressing graph-like data structures as acsets is both natural and general. The remainder of this paper will be mainly concerned with the mathematics and implementation of acsets, so the reader who is primarily interested in the practical use of this technology may proceed directly to the software in Catlab.



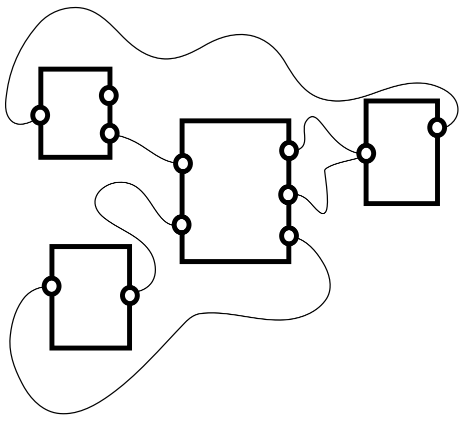
$$\begin{tikzcd} \textbf{Wire} \arrow[r, shift left, "\text{src}"] \arrow[r, shift right, "\text{tgt}"'] & \textbf{Port} \arrow[r, "\text{box}"'] & \textbf{Box} \end{tikzcd}$$

(a) Circular port graph



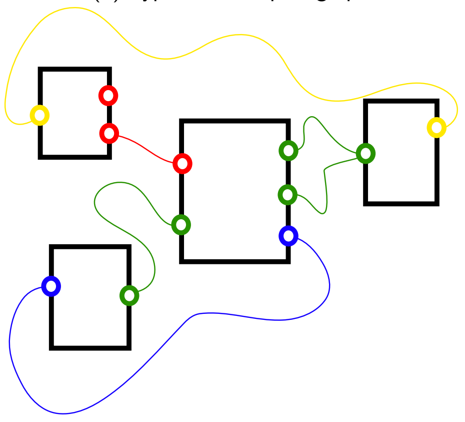
$$\begin{tikzcd} \textbf{Wire} \arrow[r, shift left, "\text{src}"] \arrow[r, shift right, "\text{tgt}"'] \arrow[d, "\text{wtype}"'] & \textbf{Port} \arrow[r, "\text{box}"'] \arrow[dl, bend left, "\text{ptype}"] & \textbf{Box} \\ \texttt{Type} & & \end{tikzcd}$$

(b) Typed circular port graph



$$\begin{tikzcd} & \textbf{InPort} \arrow[dr, "\text{ibox}"] & \\ \textbf{Wire} \arrow[ur, "\text{tgt}"] \arrow[dr, "\text{src}"'] & & \textbf{Box} \\ & \textbf{OutPort} \arrow[ur, "\text{obox}"'] & \end{tikzcd}$$

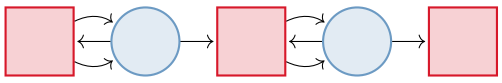
(c) Directed port graph



$$\begin{tikzcd} & \textbf{InPort} \arrow[dr, "\text{ibox}"] \arrow[d, dashed, "\text{iptype}"] & \\ \textbf{Wire} \arrow[ur, "\text{tgt}"] \arrow[dr, "\text{src}"'] \arrow[r, "\text{wtype}"] & \texttt{Type} & \textbf{Box} \\ & \textbf{OutPort} \arrow[ur, "\text{obox}"'] \arrow[u, dashed, "\text{optype}"] & \end{tikzcd}$$

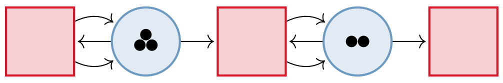
(d) Typed directed port graph

Figure 1: Port graphs



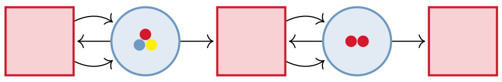
$$\begin{tikzcd}
\mathbf{Input} \arrow[d] \arrow[dr] & \\
\mathbf{Species} & \mathbf{Transition} \\
\mathbf{Output} \arrow[u] \arrow[ur] &
\end{tikzcd}$$

(a) Petri net



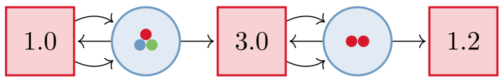
$$\begin{tikzcd}
& \mathbf{Input} \arrow[d] \arrow[dr] & \\
\mathbf{Tok} \arrow[r] & \mathbf{Species} & \mathbf{Transition} \\
& \mathbf{Output} \arrow[u] \arrow[ur] &
\end{tikzcd}$$

(b) Petri net with tokens



$$\begin{tikzcd}
& \mathbf{Input} \arrow[d] \arrow[dr] & \\
\mathbf{Tok} \arrow[r] \arrow[d] & \mathbf{Species} & \mathbf{Transition} \\
\mathtt{Type} & \mathbf{Output} \arrow[u] \arrow[ur] &
\end{tikzcd}$$

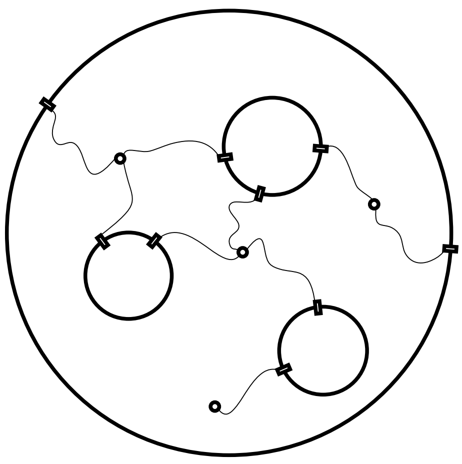
(c) Petri net with typed tokens



$$\begin{tikzcd}
& \mathbf{Input} \arrow[d] \arrow[dr] & \\
\mathbf{Tok} \arrow[r] \arrow[d] & \mathbf{Species} & \mathbf{Transition} \arrow[d] \\
\mathtt{Type} & \mathbf{Output} \arrow[u] \arrow[ur] & \mathtt{Rate}
\end{tikzcd}$$

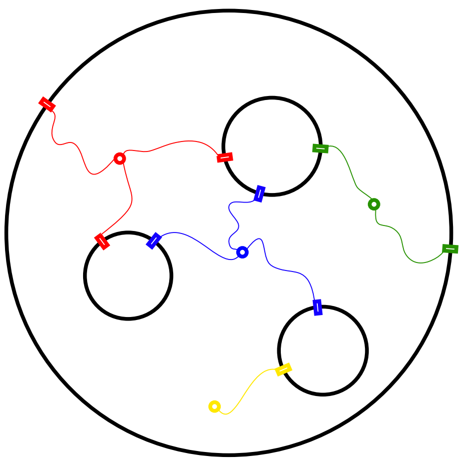
(d) Petri net with typed tokens and rates

Figure 2: Whole-grained Petri nets



$$\begin{tikzcd}
& & \mathbf{Box} \\
\mathbf{OuterPort} \arrow[r] & \mathbf{Junction} & \mathbf{Port} \arrow[l] \arrow[u]
\end{tikzcd}$$

(a) Undirected wiring diagram



$$\begin{tikzcd}
& & \mathbf{Box} \\
\mathbf{OuterPort} \arrow[r] \arrow[dr, bend right] & \mathbf{Junction} \arrow[d] & \mathbf{Port} \arrow[l] \arrow[u] \arrow[dl, bend left] \\
& \mathtt{Type} &
\end{tikzcd}$$

(b) Typed undirected wiring diagram

Figure 3: Undirected wiring diagrams

3 Review of C-sets

Before turning to the theory of attributed C-sets, we review the concept of a C-set, also known as a copresheaf on a category C. In this section and the next, we assume acquaintance with the basic concepts of category theory, namely categories, functors, and natural transformations.

3.1 Definition and Examples

Definition 1. Given a small category C, a C-*set* is a functor $F\colon \mathsf{C} \to \mathsf{Set}$. For the purposes of computing, we sometimes restrict to *finite* C-set, which are functors $F\colon \mathsf{C} \to \mathsf{FinSet}$.

Example 4. Let $\mathsf{Gr} = \{E \rightrightarrows V\}$ be the category with two objects E and V and two parallel morphisms, $\mathrm{src}, \mathrm{tgt}\colon E \to V$. Then a Gr-set is a graph.

When implementing mathematics on a computer, everything must represented by a finite amount of data. In the case of C-sets, this means that we work only with *finitely presented* categories C. The next several definitions explain this idea, starting with the simplest way to present a category, as a free category.

Definition 2. There is a forgetful functor $|{-}|\colon \mathsf{Cat} \to \mathsf{Set}^{\mathsf{Gr}}$ that takes a small category to its underlying graph. We define $\langle - \rangle\colon \mathsf{Set}^{\mathsf{Gr}} \to \mathsf{Cat}$ to be the left adjoint to this functor, which takes a graph G to the *free category* $\langle G \rangle$ on that graph. Concretely, an object of $\langle G \rangle$ is a vertex of G and a morphism $u \to v$ of $\langle G \rangle$ is a (possibly empty) path in G from vertex u to vertex v.

Definition 3. A *parallel pair* in a category C is a diagram in C of shape $\{0 \rightrightarrows 1\}$, or equivalently a pair of morphisms (f, g) in C such that $\operatorname{dom} f = \operatorname{dom} g$ and $\operatorname{codom} f = \operatorname{codom} g$.

In the following definition, parallel pairs will represent the equations between morphisms that are imposed in a finitely presented category.

Definition 4. If G is a graph and R is a set of parallel pairs in $\langle G \rangle$, let $\langle G|R \rangle$ be the initial category equipped with a map $q_R\colon \langle G \rangle \to \langle G|R \rangle$, called the *quotient map*, such that $q_R(f) = q_R(g)$ for all $(f, g) \in R$. We say that the category $\langle G|R \rangle$ is *presented* by the *generators* in G and *relations* in R. In particular, a category C is *finitely presented* by a graph G and relations R if G and R are both finite and $\mathsf{C} \cong \langle G|R \rangle$.

To say that $\langle G|R \rangle$ is the *initial category* with a quotient map $q_R\colon \langle G \rangle \to \langle G|R \rangle$ is to say that for any other category C with a map $q'_R\colon \langle G \rangle \to \mathsf{C}$ such that $q'_R(f) = q'_R(g)$ for all $(f, g) \in R$, there exists a unique functor $F\colon \langle G|R \rangle \to \mathsf{C}$ such that $q'_R = q_R \mathbin{⨟} F$. An explicit construction of the category $\langle G|R \rangle$ is given by Borceux [3, Proposition 5.1.6], among other sources.

Example 5. Consider the graph G given by

$$\begin{tikzcd} E \arrow[loop left, "\mathrm{inv}"] \arrow[r, shift left, "\mathrm{src}"] \arrow[r, shift right, "\mathrm{tgt}"'] & V \end{tikzcd}$$

and the set of parallel pairs

$$R = \{(\mathrm{inv}^2, 1_E), (\mathrm{inv} \mathbin{⨟} \mathrm{src}, \mathrm{tgt}), (\mathrm{inv} \mathbin{⨟} \mathrm{tgt}, \mathrm{src})\}.$$

Then $\mathsf{SymGr} = \langle G|R \rangle$ is a finitely presented category such that SymGr-sets are *symmetric graphs*. In a symmetric graph F, every edge $e \in F(E)$ is paired with an edge $F(\mathrm{inv})(e)$ going in the opposite direction, so that the set $\{e, F(\mathrm{inv})(e)\}$ can be interpreted as an "undirected edge" between $F(\mathrm{src})(i)$ and $F(\mathrm{tgt})(i)$.

Example 6. Let $\mathsf{BipartiteGr}$ be the category freely generated by the graph

$$\begin{tikzcd} & V_a & \\ E_{ab} \arrow[ur, "\mathrm{src}_a"] \arrow[dr, "\mathrm{tgt}_b"'] & & E_{ba} \arrow[ul, "\mathrm{tgt}_a"'] \arrow[dl, "\mathrm{src}_b"] \\ & V_b & \end{tikzcd} .$$

Then a $\mathsf{BipartiteGr}$-set F is a *bipartite graph* where $F(V_a)$ and $F(V_b)$ are the vertices in each partition and E_{ab} and E_{ba} are the edges going from V_a to V_b and from V_b to V_a, respectively.

Example 7. Let Dyn be the category freely generated by

$$\begin{tikzcd} X \arrow[loop right, "\text{succ}"] \end{tikzcd} .$$

Then a Dyn-set is a discrete dynamical system. When viewed as a monoid, Dyn is isomorphic to the natural numbers $(\mathbb{N}, +, 0)$.

3.2 The Category of C-sets

Recall that for two categories C and D, the functor category D^{C} is the category whose objects are functors $F\colon \mathsf{C} \to \mathsf{D}$ and whose morphisms are natural transformations $\alpha\colon F \Rightarrow G$. The category C-Set is simply $\mathsf{Set}^{\mathsf{C}}$ (or $\mathsf{FinSet}^{\mathsf{C}}$ when we restrict our attention to finite C-sets). This category enjoys excellent mathematical properties as a result of being a functor category.

Proposition 1. *Limits and colimits in* D^{C} *are computed pointwise in* D*. More precisely, if* J *is a small category and* D *has all (co)limits of shape* J*, then* D^{C} *has all (co)limits of shape* J*. Moreover, the limit of a diagram* $K\colon J \to \mathsf{D}^{\mathsf{C}}$ *is computed via the formula*

$$\left(\varprojlim_{j\in J} K(j)\right)(c) = \varprojlim_{j\in J}(K(j)(c))$$

for all $c \in \operatorname{ob}\mathsf{C}$*, and similarly for colimits.*

The proof of this standard result can be found in [1, Sections 8.5-8.6] or [22, Proposition 3.3.9]. Because Set has all (small) limits and colimits, and FinSet has all finite limits and colimits, the proposition establishes that $\mathsf{C}\text{-}\mathsf{Set} = \mathsf{Set}^{\mathsf{C}}$ and $\mathsf{FinSet}^{\mathsf{C}}$ have the corresponding limits and colimits. The computational content of the proposition is illustrated by the following examples.

Example 8. The product $G_1 \times G_2$ of graphs G_1 and G_2 has edge set

$$(G_1 \times G_2)(E) = G_1(E) \times G_2(E)$$

and vertex set

$$(G_1 \times G_2)(V) = G_1(V) \times G_2(V).$$

Both equations follow from the formula in Proposition 1. The edge $(e, e') \in (G_1 \times G_2)(E)$ in the product graph has source vertex $(G_1(\mathrm{src})(e), G_2(\mathrm{src})(e'))$, and similarly for the target.

Example 9. The coproduct of two discrete dynamical systems D_1 and D_2 has state space

$$(D_1 \sqcup D_2)(X) = D_1(X) \sqcup D_2(X).$$

The successor map $(D_1 \sqcup D_2)(\mathrm{succ})$ is defined by

$$(D_1 \sqcup D_2)(\mathrm{succ})(x) = \begin{cases} D_1(\mathrm{succ})(x) & \text{if} \quad x \in D_1(X) \\ D_2(\mathrm{succ})(x) & \text{if} \quad x \in D_2(X) \end{cases}.$$

We are interested in limits and colimits because applications of C-sets frequently involve operations that can be expressed using limits or colimits; we will see later that being able to compute pushouts is essential for composing structured cospans. The pointwise formula leads to a generic algorithm for computing limits and colimits in functor categories, which we have implemented for finite C-sets.

3.3 Queries and Data Migration

Given two schemas C and D along with a functor $f\colon \mathsf{C} \to \mathsf{D}$, the induced *pullback functor* $f^*\colon \mathsf{Set}^{\mathsf{D}} \to \mathsf{Set}^{\mathsf{C}}$ from D-sets to C-sets is given by precomposition with f.

Example 10. Let $1 = \{*\}$ be the terminal category and let $f\colon 1 \to \mathsf{Gr}$ be the functor sending $*$ to V. Then $f^*\colon \mathsf{Set}^{\mathsf{Gr}} \to \mathsf{Set}^1 \cong \mathsf{Set}$ is the forgetful functor sending a graph to its set of vertices.

Example 11. Let ι be the inclusion functor from the category Gr into SymGr. The pullback functor ι^* sends a symmetric graph to its underlying graph, forgetting the involution on edges.

For any functor f, the pullback functor f^* has left and right adjoints, denoted Σ_f and Π_f.

$$\begin{tikzcd}
\mathsf{Set}^C \arrow[r, bend left=40, "\Sigma_f"] \arrow[r, bend right=40, "\Pi_f"'] & \mathsf{Set}^D \arrow[l, "f^*" description] \arrow[l, phantom, shift left=3, "\perp"] \arrow[l, phantom, shift right=3, "\perp"]
\end{tikzcd}$$

The simplest way to think about these adjunctions is through the natural bijections they define between hom sets. That is,

$$\mathrm{Hom}(X, f^*(Y)) \cong \mathrm{Hom}(\Sigma_f(X), Y)$$
$$\mathrm{Hom}(f^*(X), Y) \cong \mathrm{Hom}(X, \Pi_f(Y))$$

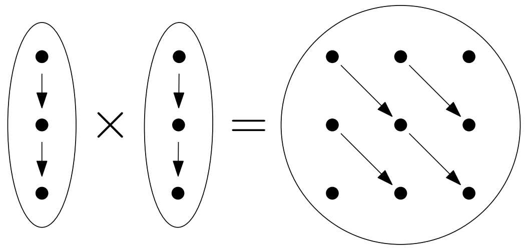
for all C-sets X and D-sets Y.

Returning to Example 10, we can use these isomorphisms between hom sets to see that $\Sigma_f(X)$ is the graph with vertex set X and no edges (the *discrete* graph), and $\Pi_f(Y)$ is the graph with vertex set Y and precisely one edge between each pair of vertices (the *codiscrete* graph).

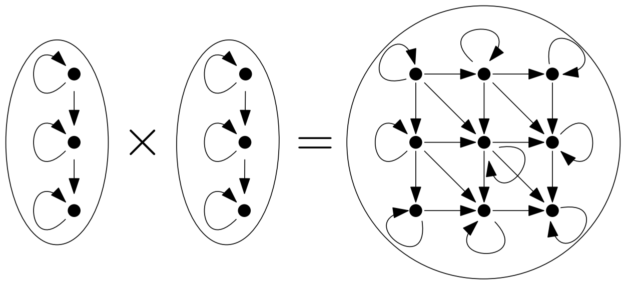
Example 12. A *reflexive graph* is a graph where every vertex is equipped with a distinguished self-loop; more precisely, it is a ReflGr-set, where

$$\mathsf{ReflGr} := \left\langle \begin{tikzcd} E \arrow[r, shift left=2, "\mathrm{src}"] \arrow[r, shift right=2, "\mathrm{tgt}"'] & V \arrow[l, "\mathrm{refl}" description] \end{tikzcd} \;\middle|\; \begin{array}{l} \mathrm{refl} \mathbin{⨾} \mathrm{src} = 1_V \\ \mathrm{refl} \mathbin{⨾} \mathrm{tgt} = 1_V \end{array} \right\rangle.$$

Starting from the evident inclusion $\iota\colon \mathsf{Gr} \to \mathsf{ReflGr}$, the functor Σ_ι freely adds reflexive edges to each vertex of a graph. This transformation is useful because the product of two line graphs does not produce a mesh (Figure 4a), but the product of two reflexive line graphs does (Figure 4b).



(a) Product of two non-reflexive graphs



(b) Product of two reflexive graphs

Figure 4: Categorical products of graphs

For any functor f between finitely presented categories, the pullback functor f^* is straightforwardly implemented. Computing the pushforward functors Σ_f and Π_f can, in principle, be reduced to computing limits and colimits of sets, but the reduction itself can be difficult to compute [5] and the resulting sets can be infinite.

This concludes our brief review of C-sets. C-sets, along with the dual concept of *presheaves* (functors $\mathsf{C}^{\mathrm{op}} \to \mathsf{Set}$), are among the best-studied notions of category theory. A more thorough treatment may be found in textbooks such as [1, 21, 22].

4 Theory of Attributed C-sets

In Section 2, we contrasted two kinds of information that can be contained in a dataset. The first kind, combinatorial data, is modeled well by C-sets. However, the second kind, atomic data, is not modeled appropriately because the isomorphism class of a C-set abstracts away from the actual elements of the sets. The only information it retains is how the elements are related to each other.

One approach to augmenting C-sets with attribute data, elaborated in [26], is to consider the slice category $\mathsf{Set}^{\mathsf{C}}/D$ over a fixed C-set D. An object of $\mathsf{Set}^{\mathsf{C}}/D$ is a functor $F\colon \mathsf{C} \to \mathsf{Set}$ together with a natural transformation $\alpha\colon F \to D$, whose components $\alpha_c : F(c) \to D(c)$ are assignments of data to the elements of $F(c)$. For instance, data frames can be modeled in this framework. Consider a data frame with column types $X_1, \dots, X_n$, which are simply sets. Perhaps $X_1 = \mathbb{Z}$ and $X_2 = \mathbb{R}$. Then let $\mathsf{1} = \{*\}$ be the terminal category and define the $\mathsf{1}$-set D by $D(*) := \prod_{i=1}^n X_i$. Of course, $\mathsf{Set}^{\mathsf{1}} \cong \mathsf{Set}$, so we have $\mathsf{Set}^{\mathsf{1}}/D \cong \mathsf{Set}/\prod_{i=1}^n X_i$. An object of this slice category is a set U together with a function $f\colon U \to \prod_{i=1}^n X_i$. We interpret it as a data frame by saying that each element $x \in U$ is a row whose ith column value is $(\pi_i \circ f)(x)$, where π_i is the ith projection function. Note that the morphisms in this category must preserve column values, so an isomorphism class of data frames *does* preserve the atomic data, not just the number of rows.

However, for a general category C and choice of data attributes, the construction of the C-set D becomes complicated. It is not as simple as sending each object to the product of the attribute types for that object, because morphisms in C effectively induce extra attributes on their domain objects. For this reason we turn to another definition of acsets, which ultimately reduces to slice categories but is more convenient to work with.

The idea is that we will identify C with one half of a larger category $|S|$, where the rationale for the notation $|S|$ will become clear shortly. The half of $|S|$ identified with C specifies the structure of the combinatorial data; the other half specifies the structure of the atomic data. As an example, the schema for decorated graphs is pictured in Figure 5. By "decorated graph," we mean a graph whose vertices and edges are decorated with data of a specific type.

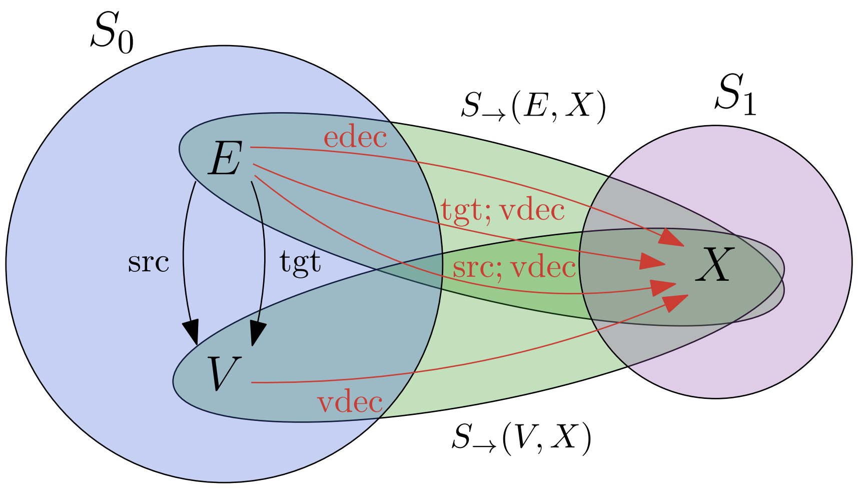


Figure 5: Schema for decorated graphs

More formally, we make the following definition.

Definition 5. A *schema* is a small category $|S|$ with a map $S\colon |S| \to \mathsf{2}$, where $\mathsf{2}$ is the interval category $\{0 \to 1\}$, Unless otherwise stated, we assume that $S^{-1}(1)$ is a discrete category.

We denote the preimages $S^{-1}(0)$ and $S^{-1}(1)$ by S_0 and S_1, respectively, and for $c \in S_0$ and $X \in S_1$, we write $S_{\to}(c, X) := \mathrm{Hom}_{|S|}(c, X)$. As in the setting of C-sets, we typically require that $|S|$ be finitely presented.

Observe that $S_{\to}$ defines a functor $S_0^{\mathrm{op}} \times S_1 \to \mathsf{Set}$. This gives an alternative definition of a schema as two categories S_0 and S_1 together with a functor $S_{\to} \colon S_0^{\mathrm{op}} \times S_1 \to \mathsf{Set}$. Such a functor is called a *profunctor* and is denoted by $S_{\to} \colon S_0 \nrightarrow S_1$. Loosely speaking, a profunctor is a kind of "bipartite category." The category $|S|$ in the original definition is recovered as the *collage* of the profunctor.

Without further ado, we can define an attributed C-set.

Definition 6 (Main Definition)**.** An *acset* on a schema S with *typing map* $K \colon S_1 \to \mathsf{Set}$ is a functor $G \colon |S| \to \mathsf{Set}$ that restricts to $G|_{S_1} = K$. It is *finite* if G restricts to a functor $G|_{S_0}$ into FinSet.

A *morphism of acsets* $G_1, G_2 \colon |S| \to \mathsf{Set}$ with typing map K is a natural transformation $\alpha \colon G_1 \to G_2$ such that $\alpha|_{S_1}$ is the identity transformation on K. The category of K-typed acsets on the schema S and morphisms between them is denoted Acset_K^S.

For readers familiar with the double category of profunctors, $\mathbb{P}\mathrm{rof}$, an equivalent but perhaps more elegant definition of acsets and acset morphisms is as follows.

Definition 7 (Alternative Definition)**.** An *acset* on a schema S with typing $K \colon S_1 \to \mathsf{Set}$ is a functor $F \colon \mathsf{C} \to \mathsf{Set}$ together with a 2-cell α in $\mathbb{P}\mathrm{rof}$ of form

$$\begin{tikzcd} S_0 \arrow[r, "S_{\to}", "\shortmid" marking] \arrow[d, "F"'] \arrow[dr, phantom, "\Downarrow \alpha"] & S_1 \arrow[d, "K"] \\ \mathsf{Set} \arrow[r, "\mathrm{Hom}_{\mathsf{Set}}"', "\shortmid" marking] & \mathsf{Set} \end{tikzcd} .$$

A *morphism of acsets* (F, α) and (G, β) on a schema S with typing K is a natural transformation $\gamma \colon F \Rightarrow G$ such that

$$\begin{tikzcd} S_0 \arrow[r, "\mathrm{Hom}_{S_0}", "\shortmid" marking] \arrow[d, "F"'] \arrow[dr, phantom, "\Downarrow \gamma"] & S_0 \arrow[r, "S_{\to}", "\shortmid" marking] \arrow[d, "G"'] \arrow[dr, phantom, "\Downarrow \beta"] & S_1 \arrow[d, "K"] \\ \mathsf{Set} \arrow[r, "\mathrm{Hom}_{\mathsf{Set}}"', "\shortmid" marking] & \mathsf{Set} \arrow[r, "\mathrm{Hom}_{\mathsf{Set}}"', "\shortmid" marking] & \mathsf{Set} \end{tikzcd} = \begin{tikzcd} S_0 \arrow[r, "S_{\to}", "\shortmid" marking] \arrow[d, "F"'] \arrow[dr, phantom, "\Downarrow \alpha"] & S_1 \arrow[d, "K"] \\ \mathsf{Set} \arrow[r, "\mathrm{Hom}_{\mathsf{Set}}"', "\shortmid" marking] & \mathsf{Set} \end{tikzcd} .$$

Our definition of an acset is closely related to other definitions of categorical databases with attributes in the literature [27, 25]. Definition 7 is a simpler but less expressive variant of the "algebraic databases" introduced by Schultz et al [25], replacing *algebraic profunctors*—product-preserving profunctors into multisorted algebraic theories—with profunctors into discrete categories. This means that our data model excludes operations on data attributes, although of course such operations can still be performed in ordinary Julia code. On the other hand, our definition is slightly richer than that of Spivak and Wisnesky [27], including a notion of *attribute type* rather than treating each attribute as having its own independent data type. This addition is convenient in our implementation, as attribute types correspond to type parameters in the generated Julia data type (see Section 5). In summary, our notion of acset is intermediate in complexity between those in [27] and [25].

4.1 Acset_K^S as a Slice Category

In this section, we show that the category Acset_K^S is a slice category in $\mathsf{Set}^{\mathsf{C}}$, where $\mathsf{C} = S_0$. As a result, Acset_K^S inherits useful categorical properties from $\mathsf{Set}^{\mathsf{C}}$, such as completeness and cocompleteness.

Theorem 2. *The category* Acset_K^S *is isomorphic to the slice category* $\mathsf{Set}^{\mathsf{C}}/D$ *for some* C*-set* D*.*

Proof. The proof happens in two major steps. The first step is to establish that acsets are fully captured by the diagram in Figure 6; the second is to recognize the possibility of constructing D from a certain Kan extension.

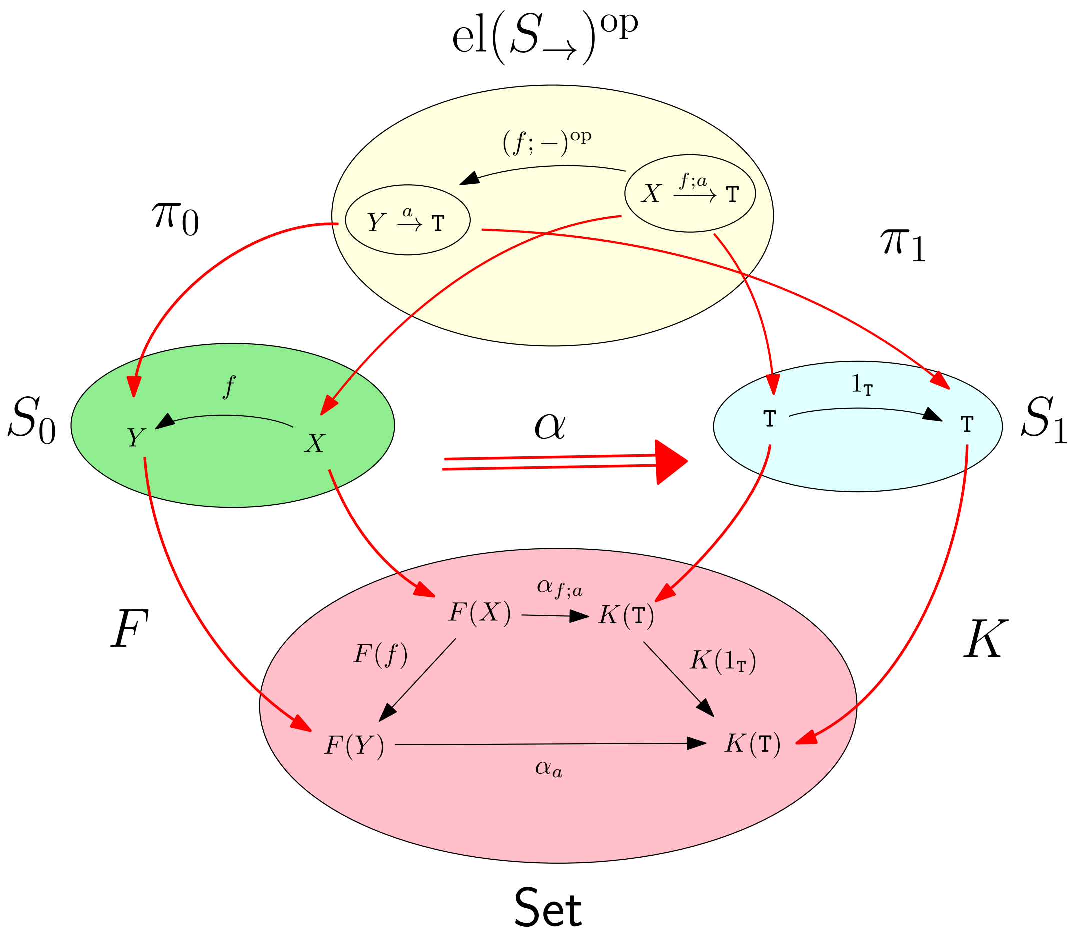


Figure 6: An acset as a natural transformation α from $\pi_0 \mathbin{⨾} F$ to $\pi_1 \mathbin{⨾} K$

Recall the alternative definition of the schema S as a functor $S_{\to}\colon S_0^{\mathrm{op}} \times S_1 \to \mathsf{Set}$. As S_1 is discrete, $S_1 = S_1^{\mathrm{op}}$ and we can reinterpret $S_{\to}$ as a functor $S_{\to}\colon (S_0 \times S_1)^{\mathrm{op}} \to \mathsf{Set}$. Let $\mathrm{el}(S_{\to})$ be the category of elements of $S_{\to}$, whose objects are attributes $Y \xrightarrow{a} \mathtt{T}$, and let be $p\colon \mathrm{el}(S_{\to}) \to (S_0 \times S_1)^{\mathrm{op}}$ the canonical projection, sending $Y \xrightarrow{a} \mathtt{T}$ to the pair $(Y, \mathtt{T})$. Then for any acset G in Acset_K^S, consisting of a functor $G : |S| \to \mathsf{Set}$ with $G|_{S_0} =: F$ and $G|_{S_1} = K$, we have a diagram

$$\begin{tikzcd} & \mathrm{el}(S_{\to})^{\mathrm{op}} \arrow[dl, "\pi_0 \circ p"'] \arrow[dr, "\pi_1 \circ p"] & \\ S_0 \arrow[rr, Rightarrow, "\alpha"] \arrow[dr, "F"'] & & S_1 \arrow[dl, "K"] \\ & \mathsf{Set} & \end{tikzcd}.$$

The natural transformation α is defined by letting its component at an element $a \in \mathrm{el}(S_{\to})^{\mathrm{op}}$, which is an attribute from $(\pi_0 \circ p)(a) \in S_0$ to $(\pi_1 \circ p)(a) \in S_1$, be the function

$$\alpha_a := G(a)\colon F((\pi_0 \circ p)(a)) \to K((\pi_1 \circ p)(a)).$$

This construction is depicted schematically in Figure 6. The naturality of α follows from the functorality of G. That is, $\alpha_{f;a} = F(f); \alpha_a$ because $G(f;a) = G(f); G(a) = F(f); G(a)$. Moreover, any such natural transformation α defines a functor $G : |S| \to \mathsf{Set}$ with $G|_{S_1} = K$. This correspondence gives an isomorphism between the category of acsets and the category of pairs (F, α). In the latter category, the morphisms $(F_1, \alpha_1) \to (F_2, \alpha_2)$ are natural transformations $F_1 \to F_2$ making a certain diagram commute, so that attributes are preserved. For further intuition, see Definition 7 of the category of acsets in terms of the double category of profunctors.

Next, we rewrite the above diagram in a slightly different form:

$$\begin{tikzcd} \mathrm{el}(S_{\to})^{\mathrm{op}} \arrow[rr, "K \circ \pi_1 \circ p"] \arrow[dr, "\pi_0 \circ p"'] & \arrow[d, phantom, "\alpha \Uparrow"] & \mathsf{Set} \\ & S_0 \arrow[ur, "F"'] & \end{tikzcd}$$

Diagrams of this form are precisely the context for right Kan extensions. Letting $\tilde{p} := \pi_0 \circ p$ and $\tilde{K} := K \circ \pi_1 \circ p$, the right Kan extension of $\tilde{K}$ along $\tilde{p}$, denoted $\mathrm{Ran}_{\tilde{p}}\, \tilde{K}$, is known to exist [22, Corollary 6.2.6]. By its defining universal property, the natural transformations α in the above diagram are in isomorphism with natural transformations $\hat{\alpha}$ in the diagram below:

$$\begin{tikzcd} \mathrm{el}(S_{\to})^{\mathrm{op}} \arrow[rr, "\tilde{K}"] \arrow[dr, "\tilde{p}"'] & & \mathsf{Set} \\ & S_0 \arrow[ur, bend left, "\mathrm{Ran}_{\tilde{p}}\, \tilde{K}"] \arrow[ur, bend right, "F"'] \arrow[ur, phantom, "\hat{\alpha} \Leftarrow"] & \end{tikzcd}$$

Let $D = \mathrm{Ran}_{\tilde{p}}\, \tilde{K}$. We have shown that an object of Acset_K^S is precisely a functor $F\colon S_0 \to \mathsf{Set}$ together with a natural transformation $\hat{\alpha}\colon F \Rightarrow D$. Moreover, tracing carefully through the argument shows that a morphism of acsets is exactly a morphism in the slice category Set^{S_0}/D, which completes the proof. □

A similar result is stated by Spivak and Wisnesky [27, Proposition 9.1.2], although the proof there is more immediate due to differences in formalism.

In the very special case where $\mathsf{C} = \{*\}$ is the terminal category, the C-set D in Theorem 2, which is just a set, is the product $\prod_{* \xrightarrow{a} \mathtt{T}} K(\mathtt{T})$ of the attribute types of all attributes in the schema. This is the example of a data frame discussed at the beginning of the section. In general, however, the right Kan extension $D = \mathrm{Ran}_{\tilde{p}}\, \tilde{K}$ used in the proof is large, complicated, and impractical to instantiate in software. Thus, it should be understood mainly as a theoretical device for understanding the category of acsets.

Indeed, it is useful to know that the category of acsets is a slice category of a presheaf category, as such categories have many desirable features. Because presheaf categories are elementary toposes and slices of toposes are again toposes (the "fundamental theorem of toposes" [16, Theorem 17.4]), the category of acsets is a topos. In particular, all finite limits and colimits exist, and can be computed by known formulas. Also, there is a geometric morphism between Set^{S_0} and the category of acsets on S. Thus, in an abstract sense, the important properties of the category of acsets are determined by the above construction and already known; the main innovation is in realizing the many useful applications thus enabled. Much of Section 5 will be dedicated to showing how the properties of slice categories of presheaf categories translate into practical capabilities for designing and implementing software.

4.2 Functorality of the Attributed C-set Construction

Careful examination of the construction of Acset^S_K shows that it is functorial in both S and K.

Functorality in S is given simply by precomposition. A *morphism of schemas* $S \to S'$ is a functor $L\colon |S| \to |S'|$ that restricts to functors $L_0 : S_0 \to S'_0$ and $L_1 : S_1 \to S'_1$. Given such a functor L and a typing map $K\colon S'_1 \to \mathsf{Set}$, there is a functor $\Delta_L\colon \mathsf{Acset}^{S'}_K \to \mathsf{Acset}^{S}_{L_1 ⨟ K}$ which sends $G\colon S' \to \mathsf{Set}$ to $L ⨟ G\colon S \to \mathsf{Set}$.

Example 13. Recall the schema for weighted graphs. For this schema, $S_1 \cong \{*\}$ has a single object, so a typing map is simply a choice of a set. Functorality of the acset construction in the typing map allows us to lift a map $\mathbb{R} \to \mathbb{C}$ to a map from $\mathbb{R}$-weighted graphs to $\mathbb{C}$-weighted graphs.

Moreover, the inclusion of the schema for "weighted sets" into the schema for weighted graphs, pictured below, induces a morphism from weighted graphs to weighted sets that sends a weighted graph to its weighted set of edges.

$$\begin{tikzcd}
\bullet \arrow[d] \arrow[rr, dotted, hook, bend left] & & E \arrow[d] \arrow[r, shift left, "\mathrm{src}"] \arrow[r, shift right, "\mathrm{tgt}"'] & V \\
\mathtt{T} \arrow[rr, dotted, hook, bend left] & & \mathtt{T} &
\end{tikzcd}$$

For functorality in K, suppose that $\gamma\colon K \Rightarrow L$ is a natural transformation. Then the Kan extension in the proof of Theorem 2 yields a map $\hat{\gamma}\colon P := \mathrm{Ran}_{\pi_0 ⨟ F}(\pi_1 ⨟ K) \to \mathrm{Ran}_{\pi_0 ⨟ F}(\pi_1 ⨟ L) =: Q$. The theory of slices of toposes [16, Chapter 11] then produces three functors:

$$\begin{tikzcd}
\mathsf{Set}^{S_0}/P \arrow[rr, bend left, "\hat{\Sigma}_{\hat{\gamma}}"] \arrow[rr, bend right, "\hat{\Pi}_{\hat{\gamma}}"'] & \perp \\ \hat{\Delta}_{\hat{\gamma}} \\ \perp & \mathsf{Set}^{S_0}/Q \arrow[ll]
\end{tikzcd}$$

It is important to note that these three functors are *different* from the three functors given by functorality in S_0. Namely, these functors do not migrate acsets between different schemas, and instead manipulate the data of an acset.

Example 14. In the case that the schema has only one object, no morphisms, and one attribute, these functors are particularly simple. Let P and Q be sets, and let $\gamma\colon P \to Q$ be any function. We then have adjunctions:

$$\begin{tikzcd}
\mathsf{Set}/P \arrow[rr, bend left, "\hat{\Sigma}_{\hat{\gamma}}"] \arrow[rr, bend right, "\hat{\Pi}_{\hat{\gamma}}"'] & \perp \\ \hat{\Delta}_{\hat{\gamma}} \\ \perp & \mathsf{Set}/Q \arrow[ll]
\end{tikzcd}$$

The functors $\hat{\Sigma}_{\hat{\gamma}}$ and $\hat{\Delta}_{\hat{\gamma}}$ have simple descriptions: $\hat{\Sigma}_{\hat{\gamma}}$ composes $X \xrightarrow{f} P$ with γ, and $\hat{\Delta}_{\hat{\gamma}}$ pulls back $Y \xrightarrow{g} Q$ along γ. To describe $\hat{\Pi}_{\hat{\gamma}}$, it is best to exploit the isomorphism $\mathsf{Set}/P \cong \mathsf{Set}^P$, viewing P as a discrete category. For more information about adjoint triples arising from slices, a standard reference is [14, Chapter 1].

Example 15. Consider again weighted graphs. If G is a $\mathbb{Z}$-weighted graph and $f\colon \mathbb{Z} \to \mathbb{R}$ is any function, then the functor $\hat{\Sigma}_{\hat{f}}$ "maps f over G" to produce a $\mathbb{R}$-weighted graph with the same underlying graph as G.

The other two functors are not as straightforward. When γ is injective, the functor $\hat{\Delta}_{\hat{\gamma}}$ can be interpreted as a "filtering transformation," which deletes the parts of the acset having attributes that are not in the domain of γ. For example, if $f\colon [0, 200] \hookrightarrow \mathbb{R}_{\geq 0}$ is the inclusion, then $\hat{\Delta}_{\hat{f}}$ takes a $\mathbb{R}_{\geq 0}$-weighted graph and deletes any edges with weights greater than 200 to produce a $[0, 200]$-weighted graph. More generally, when γ is not injective, parts can be copied as well as deleted. For the unique function $f\colon \{0, 1\} \to \{*\}$, the functor $\hat{\Delta}_{\hat{f}}$ maps a graph with weights of singleton type (i.e., an unweighted graph) to the graph where each edge is duplicated, one copy weighted by 0 and the other by 1.

In general, $\hat{\Delta}_{\hat{\gamma}}$ performs a combination of these two transformations. Again, $\hat{\Pi}_{\hat{\gamma}}$ is not so intuitive, especially in the context of acsets. The reader is encouraged to try examples for themselves and use the adjointness property to see what $\hat{\Pi}_{\hat{\gamma}}$ must be.

4.3 Structured Cospans with Attributed C-sets

Acsets are usefully combined with structured cospans, a formalism for open systems introduced by Baez and Courser [2], building on Fong's decorated cospans [9]. Categories of structured cospans generalize categories of cospans, which we now review.

Definition 8. Given a category C with pushouts, the *cospan category* $\mathrm{Csp}(\mathsf{C})$ has the same objects as C and has as morphisms from a to b, the isomorphism classes of cospans $a \to x \leftarrow b$ in C. Morphisms $a \to x \leftarrow b$ and $b \to y \leftarrow c$ are composed by pushout:

$$\begin{tikzcd}
 & & x +_b y & & \\
 & x \arrow[ur] & \ulcorner & y \arrow[ul] & \\
a \arrow[ur] & & b \arrow[ul] \arrow[ur] & & c \arrow[ul]
\end{tikzcd}$$

The identity morphism for $c \in \mathsf{C}$ is the cospan $c \xrightarrow{1_c} c \xleftarrow{1_c} c$.

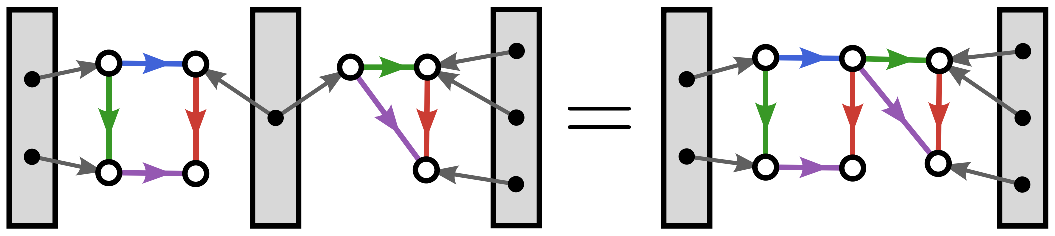
The definition involves one technical point, which is that a pushout, like any colimit, is defined only up to canonical isomorphism. The most direct way to solve this problem is to work with isomorphism classes of cospans, rather than with cospans themselves, as done above. Two cospans $a \xrightarrow{f} c \xleftarrow{g} b$ and $a \xrightarrow{f'} c' \xleftarrow{g'} b$ are *isomorphic* if there exists an isomorphism $h\colon c \to c'$ in C making the following diagram commute:

$$\begin{tikzcd}
 & c \arrow[dd, "h"] & \\
a \arrow[ur, "f"] \arrow[dr, "f'"'] & & b \arrow[ul, "g"'] \arrow[dl, "g'"] \\
 & c' &
\end{tikzcd} .$$

Structured cospans extend the concept of a cospan to the setting where the apex of the cospan has more structure than the feet. Thus, the feet of the cospan will come from a category C while the apex will belong to a different category D, which is related to C by a functor $L\colon \mathsf{C} \to \mathsf{D}$.

Definition 9. Given categories C and D, where D has all pushouts, along with a functor $L\colon \mathsf{C} \to \mathsf{D}$, the *$L$-structured cospan category* ${}_L\mathrm{Csp}(\mathsf{D})$ has the same objects as C and has as morphisms from a to b, the isomorphism classes of cospans $L(a) \to x \leftarrow L(b)$, where $a, b \in \mathsf{C}$ and $x \in \mathsf{D}$. As before, morphisms are composed by pushout.

Most combinatorial examples of structured cospans, including many examples in [2], can be constructed using categories of acsets and data migration functors. A common pattern, illustrated


(a) Composition of Open Weighted Graphs (Example 16)

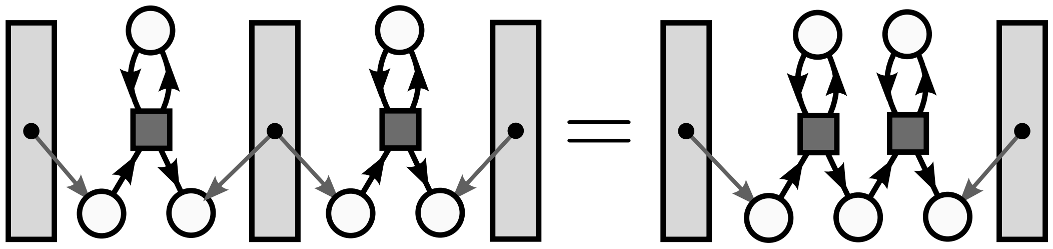

(b) Composition of Open Petri Nets (Example 17)

Figure 7: Examples of structured cospan composition for categories of weighted graphs and Petri nets.

by the following examples, takes the category D to be an arbitrary category of acsets on a schema S and takes C to be the category of acsets on a schema comprising a chosen object c of S along with any data attributes of c in S, under the assumption that c has no outgoing morphisms in S. The functor $L : \mathsf{C} \to \mathsf{D}$ is then the left adjoint to the forgetful functor $\mathsf{D} \to \mathsf{C}$.

Example 16. Consider the inclusion of the terminal category 1 into the schema for weighted graphs that sends the unique object of 1 to V. Pullback along this inclusion is the forgetful functor that sends a weighted graph to its set of vertices, and the left adjoint L to this forgetful functor sends a set to the *discrete weighted graph* having that set as vertices and no edges. The L-structured cospans are open weighted graphs, which are composed by gluing together vertices, as pictured in Figure 7a.

Example 17. Returning to the whole-grained Petri nets of Example 2, consider the inclusion of the terminal category 1 into the "SITOS schema" (Figure 2a) that sends the unique object of 1 to S. Pullback along this inclusion sends a Petri net to its set of species, and the left adjoint sends a set to the Petri net having that set as species and no transitions. Structured cospans with respect to this functor are open, whole-grained Petri nets, which are composed by gluing together species, as pictured in Figure 7b. Structured cospans of Petri nets with rates, where each transition has an associated rate constant, can be handled similarly.

Because left adjoints preserve colimits, the conditions needed to define categories of structured cospans are automatically satisfied. The fact that so many common examples of structured cospans are in fact acsets with data migration functors enables a uniform implementation of these examples.

4.4 Limits and Colimits

In this section we give explicit formulas for computing limits and colimits in categories of acsets. Since, by the main theorem 2, the category Acset_K^S is a slice category in the category of C-sets, the limit and colimit formulas may be derived from the corresponding formulas for C-sets, reviewed in Section 3.2, and for slice categories, reviewed here.

Proposition 3. *For any category* D *and object* $d \in \mathsf{D}$*, the forgetful functor* $\Pi : \mathsf{D}/d \to \mathsf{D}$ *creates colimits in the slice category* D/d*. In particular, if* D *has colimits of shape* J*, then for any diagram* $F \colon \mathsf{J} \to \mathsf{D}/d$*, we have* $\Pi(\varinjlim_{\mathsf{J}} F) \cong \varinjlim_{\mathsf{J}} F \fatsemi \Pi$.

For a proof (of the dual statement), see [22, Proposition 3.3.8]. In conjunction with the colimit formula for C-sets, we deduce:

Corollary 4. *The category* Acset_K^S *has all finite colimits, which are computed pointwise. To be precise, if* $F \colon J \to \mathsf{Acset}_K^S$ *is a finite diagram of acsets on the schema* S*, then for each object* $c \in S_0$*, we have* $\mathrm{ev}_c(\varinjlim_J F) = \varinjlim_J F \fatsemi \mathrm{ev}_c$*, where the evaluation functor* $\mathrm{ev}_c \colon \mathsf{Acset}_K^S \to \mathsf{Set}$ *gives the set associated with the object* c*.*

Limits in slice categories are less straightforward: they can be reduced to limits in the base category, but only by enlarging the diagram. Given a diagram $F : \mathsf{J} \to \mathsf{D}/d$ in a slice category, construct a diagram $\bar{F} : \mathsf{J}_\bullet \to \mathsf{D}$ in the base category as follows. Let $\mathsf{J}_\bullet$ be the category given by freely adjoining a terminal object to J, i.e., a new object $\bullet$ and for each object j in J, a unique morphism $j_\bullet : j \to \bullet$. Then define $\bar{F} : J_\bullet \to \mathsf{D}$ by setting $\bar{F}(\bullet) = d$, $\bar{F}(j_\bullet) = F(j)$ for each $j \in \mathsf{J}$, and $\bar{F}|_{\mathsf{J}} = F \fatsemi \Pi$, where $\Pi : \mathsf{D}/d \to \mathsf{D}$ is the forgetful functor. The following result appears in the literature as, for example, [28, Theorem 17.2].

Proposition 5. *A diagram* $F : \mathsf{J} \to \mathsf{D}/d$ *has a limit in the slice category* D/d *if and only if the enlarged diagram* $\bar{F} : \mathsf{J}_\bullet \to \mathsf{D}$ *has a limit in* D*, in which case the limit cone in* D *uniquely lifts to a limit cone in* D/d*.*

Corollary 6. *The category* Acset_K^S *has all finite limits, which may be reduced to limits in* $\mathsf{Set}^{\mathsf{C}}$ *by the above procedure.*

For example, the product of two acsets (F_1, β_1) and (F_2, β_2) in $\mathsf{Set}^{\mathsf{C}}/D \cong \mathsf{Acset}_K^S$ is computed as a pullback in $\mathsf{Set}^{\mathsf{C}}$:

$$\begin{tikzcd} & F_1 \times_D F_2 \arrow[dl] \arrow[dr] & \\ F_1 \arrow[dr, "\beta_1"'] & & F_2 \arrow[dl, "\beta_2"] \\ & D & \end{tikzcd} .$$

5 Implementation of Attributed C-sets

In this section we sketch the more interesting aspects of the implementation of attributed C-sets, especially those that take advantage of unique features of Julia or Catlab.

5.1 The `AttributedCSet` Data Structure

At the core of Catlab is a system to specify and manipulate generalized algebraic theories (GATs) and presentations of their models [6], which uses Julia's LISP-style metaprogramming capabilities. Using the `@theory` macro, we can define the theory of a schema to be the theory of a category extended with two new GAT types, for atomic data (`Data`) and data attributes (`Attr`). We also declare a syntax system for schemas using the `@syntax` macro, which generates new Julia types to be used in symbolic expressions and in schemas presented by generators and relations.

```
@theory Schema{Ob,Hom,AttrType,Attr} <: Category{Ob,Hom} begin
  Data::TYPE
  Attr(dom::Ob,codom::AttrType)::TYPE

  compose(f::Hom(A,B), g::Attr(B,X))::Attr(A,X) ⊣ (A::Ob, B::Ob, X::AttrType)

  (compose(f, compose(g, a)) == compose(compose(f, g), a)
    ⊣ (A::Ob, B::Ob, C::Ob, X::AttrType, f::Hom(A,B), g::Hom(B,C), a::Attr(C, X)))
  compose(id(A), a) == a ⊣ (A::Ob, X::Ob, a::Attr(A,X))
end

@syntax FreeSchema{ObExpr,HomExpr,DataExpr,AttrExpr} Schema begin
end
```

We can now give finite presentations of schemas using the `@present` macro, as we saw in Section 2. For example, the schema for weighted graphs is:

```
@present TheoryWeightedGraph(FreeSchema) begin
  (V,E)::Ob
  (src,tgt)::Hom(E,V)
  X::AttrType
  dec::Attr(E,X)
end
```

The final step in defining an acset is to create the Julia type for the data structure.

```
@acset_type WeightedGraph(TheoryWeightedGraph, index=[:src,:tgt])
```

Inspecting the resulting type `WeightedGraph`, we see that a struct has been generated, suitable for storing the data of a weighted graph.

```
> dump(WeightedGraph{Int})
WeightedGraph{Int64} <: StructACSet{
    Catlab.Theories.SchemaDescType{
      (:V, :E), (:src, :tgt), (:X,), (:dec,),
      (src = 2, tgt = 2, dec = 2), (src = 1, tgt = 1, dec = 1)
    },
    Tuple{Int64},
    (src = true, tgt = true, dec = false),
    (src = false, tgt = false, dec = false)
  }
  obs::StaticArrays.MVector{2, Int64}
  homs::NamedTuple{(:src, :tgt), Tuple{Vector{Int64}, Vector{Int64}}}
  attrs::NamedTuple{(:dec,), Tuple{Vector{Int64}}}
  hom_indices::NamedTuple{(:src, :tgt), Tuple{Vector{Vector{Int64}}, Vector{Vector{Int64}}}}
  hom_unique_indices::NamedTuple{(), Tuple{}}
  attr_indices::NamedTuple{(), Tuple{}}
  attr_unique_indices::NamedTuple{(), Tuple{}}
```

This struct subtypes the abstract type `StructACSet`, with parameters that describe the schema of a WeightedGraph. This allows us to write functions that accept a `StructACSet` and dispatch on the type parameters of `StructACSet` to generate custom code for accessing our weighted graph. We discuss this more in the next section.

5.2 Code Generation

If every low-level operation on an acset had to process the type-level description of the schema, the resulting overhead would make it impossible for acsets to compete with handwritten data structures. Fortunately, it is possible to avoid this runtime penalty using *generated functions*, a useful metaprogramming feature supported by the Julia language. Like a macro, a generated function runs arbitrary Julia code to generate a Julia expression that is then evaluated. However, while a macro operates on the *syntax* of its arguments, a generated function operates on the *types* of its arguments and returns a Julia expression for the body of the function that is specific to those types. Because the type of an acset fully describes its structure, this type contains enough information to generate specialized, fast functions for each operation. After the first run, this specialized code is cached, so that subsequent applications of the function need not generate the code again.

It is best to illustrate the usefulness of generated functions with an example that does not involve acsets, as the types of acsets are rather complicated. Instead, we use the classic example of unrolling the loop in a map operation on vectors of fixed length. In the following code, the type `StaticVector{n,T}` represents a vector of length `n` with element type `T`.

```julia
@generated function map(f::Function, v::StaticVector{n,T}) where {n,T}
  if n == 0; error("Type inference for empty vectors not implemented") end
  quote
    SVector{$n}($([:(f(v[$i])) for i in 1:n]...))
  end
end
```

When we call the function `map` on an argument of type, say, `StaticVector{3,Int}`, it is as if `map` were defined as:

```julia
function map(f::Function, v::StaticVector{3,Int})
  SVector{3}(f(v[1]), f(v[2]), f(v[3]))
end
```

In a proper implementation of statically-sized vectors, such as in `StaticArrays.jl`, much of the performance boost comes from static vectors being allocated on the stack rather than the heap. The loop unrolling would then enable generically sized static vectors to be competitive with code special-cased for particular sizes. Thus, for example, Euclidean geometry packages need not treat the two-dimensional and three-dimensional cases specially for the sake of performance.

Similarly, as the benchmarks below show, generated functions enable a generic implementation of acsets to be competitive with special-cased implementations of specific acsets. Not only does this obviate the need for a great deal of domain-specific libraries, it opens the door to specialized data structures that would previously have required too much effort to be worth writing. In particular, it removes the need for packages to use dictionaries to support arbitrary, user-defined data attributes, such as in `MetaGraphs.jl`; instead, the user can simply create a new, specialized data structure that has precisely the fields needed for the application at hand.

Almost all basic operations on acsets are implemented as generated functions. This yields a greater savings of programmer effort than it may initially appear due to the support for *indexing*, where both forward and the inverse image maps are stored to enable fast lookups. Operations such as adding and removing elements and changing the values of morphisms interact with the indices in subtle ways. By writing the accessors and mutators using generated functions, this bookkeeping can be handled once and for all.

5.3 Low-level Operations

We now demonstrate how to write high-performance algorithms using the low-level interface to acsets. As an example, we implement depth first search on a graph. The following code searches a graph `g` in a depth-first manner, starting from a vertex `s`. It returns the array of parent vertices from the search, indexed by vertex.

```julia
function dfs_parents(g::Graph, s::Int)
  n = nparts(g, :V) # number of vertices in the graph
  parents = zeros(Int, n)
  seen = zeros(Bool, n)
  S = [s]
```

```
    seen[s] = true
    parents[s] = s
    while !isempty(S)
      v = S[end]
      u = 0
      outedges = incident(g, v, :src) # all edges with source vertex v
      outneighbors = subpart(g, outedges, :tgt) # all vertices with edge from v
      for n in outneighbors
        if !seen[n]
          u = n
          break
        end
      end
      if u == 0
        pop!(S)
      else
        seen[u] = true
        push!(S, u)
        parents[u] = v
      end
    end
    return parents
  end
```

In this code, the call to `incident` returns the list of edges outgoing from a given vertex `v`, which uses the index for the morphism `src` maintained by the acset. The subsequent call to `subpart` gives the list of vertices with an incoming edge from `v`. This pattern of data access is repeated in a loop. Relational databases that only provide high-level, query-based access tend to perform poorly on graph algorithms, including many search algorithms, that are highly iterative or recursive [7].

5.4 Categorical Operations

One use for acsets is to supply the "structure" in structured cospans, as explained in Section 4.3. To support this application, we must be able to compute pushouts. In this section, we give a brief overview of how pushouts of acsets are computed, in the special case of coequalizers.

Recall from Section 4.4 that colimits of acsets are computed pointwise. Thus, before we discuss how to compute coequalizers in a category of acsets, we first discuss how to compute a coequalizer in the category of finite sets (cf. [24, §4.6]). For the purposes of this discussion, we will use the following data structures for finite sets of form $\{1, \dots, n\}$ and functions between them. Note that this code is significantly simpler and less generic than what is implemented in Catlab.

```
struct FinSet
  n::Int
end
struct FinFunction
  dom::FinSet
  codom::FinSet
  values::Vector{Int}
  function FinFunction(values::Vector{Int}, codom::Int)
    @assert all(1 <= v <= codom for v in values)
    new(FinSet(length(values)), FinSet(codom), values)
  end
end
```

We want to take the coequalizer of a parallel pair of morphisms, given by the following type.

```
struct ParallelPair
  f::FinFunction
  g::FinFunction
  function ParallelPair(f::FinFunction, g::FinFunction)
    @assert f.dom == g.dom && f.codom == g.codom
    new(f,g)
  end
end
```

Given a parallel pair $f, g\colon A \to B$, we must compute the projection map $p\colon B \to C$ onto the object C comprising the coequalizer. The algorithmic idea is to utilize the *union-find* or *disjoint-sets* data structure, which stores an equivalence relation on B, and then join the equivalence classes of $f(a)$ and $g(a)$ for all $a \in A$. In the code below, the `IntDisjointSets` data structure stores an equivalence relation on the set of integers $\{1, \dots, n\}$, and the function `union!` efficiently merges two equivalence classes.

```
function colimit(pair::ParallelPair)
  f, g = pair.f, pair.g
  m, n = f.dom.n, f.codom.n
  sets = IntDisjointSets(n)
  for i in 1:m
    union!(sets, f(i), g(i))
  end
  # Extract the map out of the IntDisjointSets data structure.
  quotient_projection(sets)
end
```

For example, the connected components of a graph can be extracted from the coequalizer of the source and target maps $V \to E$. To compute a coequalizer of acset morphisms, one must first have a Julia data structure for acset morphisms. This amounts to a named tuple of morphisms in FinSet, one for each object in the schema. Next, given a parallel pair of acset morphisms, one applies the above colimit function for each object, and then follows the proof that colimits are computed pointwise to construct the colimit acset. Note that this also requires implementing the universal property of the colimit. The details can be found in the Catlab source code.

5.5 Benchmarks

In the Section 5.2, we claimed that the use of generated functions made acsets competitive with hand-written data structures. In this section we provide some evidence for that claim by benchmarking against `LightGraphs.jl`, a state-of-the-art Julia package for graphs [4]. LightGraphs boasts performance competitive with graph libraries written in C++ and vastly superior to the popular Python graph library NetworkX [13].

The results of the benchmarks are shown in Figure 8, normalized by the time of the state of the art in Julia (LightGraphs/MetaGraphs). Code to generate these benchmarks is available on GitHub [19]. Performance faster than the state of the art is highlighted in green, and performances worse than 2x the state of the art is highlighted in red. In “Graph” and “SymmetricGraph”, we benchmark the operations of checking whether a graph has an edge, iterating through the edges of a graph, iterating through the neighbors of a vertex, and constructing a path graph, for directed and symmetric graphs, respectively. Our data structure for symmetric graphs is intrinsically worse than an “undirected” graph data structure, so we cannot expect to be as fast in the symmetric graph benchmarks, unfortunately. In “GraphConnComponents” and “SymmetricGraphConnComponents”, we compute the connected components of path graphs, complete graphs, star graphs, and the Tutte graph. In “WeightedGraph” and “LabeledGraph”, we modify and iterate over the weights and labels of weighted and labeled graphs, respectively. Finally, in “RandomGraph”, we construct random graphs having various distributions, and in “Searching” we traverse the random graphs that we generated previously.

The benchmarks show that without too much optimization effort, and with no graph-specific code in the acset core, acsets achieve a performance comparable to LightGraphs, often within a factor of two. However, when we compare with `MetaGraphs.jl`, a commonly used package for

Category	Benchmark	Normalized Time
Graph	iter-neighbors	1.37
	iter-edges	0.31
	make-path	0.31
	has-edge	0.54
SymmetricGraph	iter-neighbors	6.19
	iter-edges	0.40
	make-path	9.53
	has-edge	0.78
GraphConnComponents	path-graph	0.43
	complete100	0.04
	path500	0.38
	star-graph	1.02
SymmetricGraphConnComponents	path-graph-components	1.11
	star-graph-components	0.45
	complete100	1.38
	path500	0.82
	tutte	1.21
LabeledGraph	indexed-lookup	0.67
	make-discrete	0.72
	iter-labels	0.01
	make-discrete-indexed	0.45
WeightedGraph	sum-weights	0.001
	increment-weights	0.0001
RandomGraph	expected_degree_graph-10000-10	1.55
	watts_strogatz-10000-10	1.87
	erdos_renyi-10000-0.001	1.10
Searching	dfs_erdos_renyi-10000-0.001	1.13
	bfs_erdos_renyi-10000-0.001	1.25

Figure 8: Performance Benchmarks for graph algorithms. The normalized time is the runtime using our ACSet implementation divided by the runtime using the LightGraphs.jl implementation. Numbers less than one represent performance improvement and numbers greater than one represent performance degradation.

attaching data attributes to a LightGraphs graph, we find that acsets provide very large speedups. This is possible because, as we have seen, our implementation of acsets can generate a graph data structure specialized to any particular pattern of vertex and edge attributes. On the other hand, MetaGraphs handles all cases at once by attaching dictionaries to each vertex and each edge, which results in an inefficient layout of the data in memory.

We conclude from these benchmarks that it is reasonable to use acsets for performance-sensitive tasks in scientific computing. Although further performance gains are surely possible and may be the subject of future work, the system is already usable for most in-memory workloads. Indeed, that acsets achieve fairly good performance in situations where there are viable alternatives (e.g., LightGraphs or DataFrames), as well as in many situations where there are no reasonable alternatives at all, gives us high hopes for the future of this categorical approach to data structures.

6 Summary and Outlook

In this paper, we have laid the theoretical and computational groundwork for the use of acsets as a practical data structure for technical computing. The advantages of this approach are threefold. Acsets provide a unifying abstraction for many existing data structures, including graphs and data frames; acsets enable the rapid development of new data structures for relational data; and the category theory underlying acsets is well understood, enabling the implementation of powerful, general operations such as limits, colimits, and functorial data migration.

Many directions for future work remain. One of the most obvious is replacing the category Set with other categories admitting useful computational representations. For instance, we could replace Set with Par, the category of sets and partial maps, offering one approach to graphs with “dangling edges” (edges which may have undefined sources or targets). Other interesting categories include Rel, the category of sets and relations; $\mathsf{Vect}_{\mathbb{R}}$, the category of real vector spaces and linear maps; $\mathsf{LinRel}_{\mathbb{R}}$, the category of real vector spaces and linear relations; and Markov, the category of measurable spaces and Markov kernels. For all these large categories, we would work in the skeletization of a finitary subcategory when implementing a data structure, using, for example, the category $\mathsf{Mat}_{\mathbb{R}}$ of real-valued matrices instead of $\mathsf{Vect}_{\mathbb{R}}$.

Generalizing in a different direction, we could consider monoidal schemas and monoidal functors from the schema to the cartesian monoidal $(\mathsf{Set}, \times)$. The corresponding data structures would generalize from vectors to matrices and higher-order tensors, e.g., $F(c_1 \otimes c_2 \xrightarrow{f} d)$ would be a $F(c_1) \times F(c_2)$ matrix with values in $F(d)$. This approach would be useful for modeling multivariate observations where both the number of samples and the dimension of each sample are not known in advance, and might be compared with the Python package `xarray` [11].

All these variants could in principle be implemented in Julia using methods similar to those presented in this paper. However, more theoretical work would be needed to understand the implications for the many categorical constructions supported by acsets.

Another direction for future work would be to improve the symbolic reasoning capabilities around acset mutation. The relations in an acset schema are known to be respected by category-theoretic constructions like limits and colimits but are not automatically verified in user code. For example, when using symmetric graphs (Example 5), any limit or colimit of symmetric graphs is guaranteed to yield a valid symmetric graph, but it is the user’s responsibility to ensure that any direct mutation of the acset data structure preserves the equations governing the edge involution. At least in special circumstances, it should be possible to check whether a given pattern of acset mutation preserves the invariants implied by the relations. This could be done statically, i.e., during the code generation phase, to avoid a runtime penalty. Verification would likely be impractical for arbitrary mutations, but could be carried out for certain basic mutations. Then, provided that all higher-level code used only these basic mutations, the invariants would be guaranteed to hold. For instance, the simple function that adds an edge to a symmetric graph along with its reversal could be checked for correctness. Provided that higher-level code added edges using only this function, the validity of any symmetric graph thus constructed would be guaranteed.

A third direction for future work is creating and manipulating acset schemas using high-level operations. For instance, given a schema S, one could construct another schema $S^{\rightarrow}$, the “arrow schema” on S, whose instances are two instances of S together with an acset morphism between them. In principle, it is also possible to take limits and colimits of schemas themselves. Complex data structures could then be constructed compositionally from simpler ones in a principled manner.

To conclude, we might compare the usage of acsets with that of another, highly popular family of parameterized data types: algebraic data types, formalized mathematically as initial algebras of polynomial functors [10]. The acset data model is ubiquitous in its guise as relational databases but rarely implemented as in-memory data structures. On the other hand, many programming languages, especially functional ones, support some form of algebraic data types, yet algebraic data types are quite uncommon in database systems. Given the clear benefits of each paradigm, it is unclear to us why this separation should hold. Our implementation of acsets in Julia, although efficient and usable, would enjoy improved ergonomics if it were a built-in programming language feature; likewise, algebraic data types would be a powerful feature in databases. The two families of data structures are useful in different circumstances: algebraic data types excel at representing syntax trees and other recursive data, while acsets effectively represent graph-like structures and tabular data. Looking forward, we expect to see more implementations of acsets in both programming languages and libraries. We hope that acsets will eventually be recognized as having the same fundamental status as algebraic data types, and will join data frames and graphs as a standard abstraction for data science and scientific computing.

Acknowledgments The authors thank David Spivak for suggestions regarding the proof of Theorem 2 and Andrew Baas, Kris Brown, Micah Halter, and Sophie Libkind for contributing to the development of AlgebraicJulia and providing useful feedback on the manuscript.

Funding Statement The authors were supported by DARPA Awards HR00112090067 and HR00111990008, ARO Award W911NF-20-1-0292, and AFOSR Awards FA9550-17-1-0058 and FA9550-20-1-0348. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Steve Awodey. *Category Theory.* Second Edition. Oxford Logic Guides. Oxford, New York: Oxford University Press, 2010.

[2] John C. Baez and Kenny Courser. “Structured cospans”. In: *Theory and Applications of Categories* 35.48 (2020), pp. 1771–1822. arXiv: 1911.04630.

[3] Francis Borceux. *Handbook of categorical algebra 1: Basic category theory.* Vol. 1. Cambridge University Press, 1994. DOI: 10.1017/CBO9780511525858.

[4] Seth Bromberger, James Fairbanks, and other contributors. *JuliaGraphs/LightGraphs.jl: an optimized graphs package for the Julia programming language.* 2017. DOI: 10.5281/zenodo.889971.

[5] M. R. Bush, M. Leeming, and R. F. C. Walters. “Computing left Kan extensions”. In: *Journal of Symbolic Computation* 35.2 (2003), pp. 107–126. DOI: 10.1016/S0747-7171(02)00102-5.

[6] John Cartmell. “Generalised algebraic theories and contextual categories”. In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243. DOI: 10.1016/0168-0072(86)90053-9.

[7] Yijian Cheng, Pengjie Ding, Tongtong Wang, Wei Lu, and Xiaoyong Du. “Which category is better: Benchmarking relational and graph database management systems”. In: *Data Science and Engineering* 4.4 (2019), pp. 309–322. DOI: 10.1007/s41019-019-00110-3.

[8] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[9] Brendan Fong. “Decorated Cospans”. In: *Theory and Applications of Categories* 30.33 (2015), pp. 1096–1120. arXiv: 1502.00872.

[10] Makoto Hamana and Marcelo Fiore. "A foundation for GADTs and inductive families: dependent polynomial functor approach". In: *Proceedings of the seventh ACM SIGPLAN workshop on Generic Programming*. 2011, pp. 59–70. DOI: 10.1145/2036918.2036927.

[11] S. Hoyer and J. Hamman. "xarray: N-D labeled arrays and datasets in Python". In: *Journal of Open Research Software* 5.1 (2017). Publisher: Ubiquity Press. DOI: 10.5334/jors.148.

[12] Joachim Kock. "Elements of Petri nets and processes". In: *arXiv preprint* (2020). arXiv: 2005.05108.

[13] Timothy Lin. *Benchmark of popular graph/network packages v2*. 2020. URL: https://www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2 (visited on 05/18/2021).

[14] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer, 1994. DOI: 10.1007/978-1-4612-0927-0.

[15] Wes McKinney. "Data Structures for Statistical Computing in Python". In: Python in Science Conference. Austin, Texas, 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[16] Colin McLarty. *Elementary categories, elementary toposes*. Oxford University Press, 1992.

[17] Kirill Müller and Hadley Wickham. *tibble: Simple Data Frames*. 2018. URL: https://CRAN.R-project.org/package=tibble.

[18] Evan Patterson, Micah Halter, Owen Lynch, James Fairbanks, Andrew Baas, Kris Brown, Sophie Libkind, and other contributors. *AlgebraicJulia/Catlab.jl: v0.13.5*. Version v0.13.5. Dec. 2021. DOI: 10.5281/zenodo.5771194. URL: https://doi.org/10.5281/zenodo.5771194.

[19] Evan Patterson, Owen Lynch, and James Fairbanks. *Catlab Benchmarks*. Jan. 2022. URL: https://github.com/AlgebraicJulia/Catlab.jl/blob/d6b569039584ba756c9687c54190370fceca2a9d/benchmark/README.md.

[20] R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2020. URL: https://www.R-project.org/.

[21] Marie La Palme Reyes, Gonzalo E. Reyes, and Houman Zolfaghari. *Generic figures and their glueings: A constructive approach to functor categories*. Polimetrica, 2004.

[22] Emily Riehl. *Category Theory in Context*. Mineola, New York: Dover Publications, 2016.

[23] Dylan Rupel and David I. Spivak. "The operad of temporal wiring diagrams: formalizing a graphical language for discrete-time processes". In: *arXiv preprint* (2013). arXiv: 1307.6894.

[24] David E. Rydeheard and Rod M. Burstall. *Computational category theory*. Prentice Hall, 1988.

[25] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. "Algebraic Databases". In: *Theory and Applications of Categories* 32.16 (2016), pp. 547–619. arXiv: 1602.03501.

[26] David I. Spivak. "Functorial Data Migration". In: *Information and Computation* 217 (2012), pp. 31–51. DOI: 10.1016/j.ic.2012.05.001. arXiv: 1009.1166.

[27] David I. Spivak and Ryan Wisnesky. "Relational foundations for functorial data migration". In: *Proceedings of the 15th Symposium on Database Programming Languages*. Extended version on arXiv. 2015, pp. 21–28. DOI: 10.1145/2815072.2815075. arXiv: 1212.5303.

[28] Oswald Wyler. *Lecture notes on topoi and quasitopoi*. World Scientific, 1991. DOI: 10.1142/1047.