Principles and pitfalls of software design for applied category theory

Evan Patterson

TOPOS INSTITUTE

Topos Berkeley Seminar

February 14, 2022

The AlgebraicJulia team

Besides me, the core contributors to the AlgebraicJulia ecosystem are:

James Fairbanks

University of Florida

Andrew Baas

Georgia Tech Research Institute

Sophie Libkind

Stanford University

Topos Institute

Owen Lynch

Utrecht University

Kris Brown

University of Florida

Micah Halter

Balena

What is AlgebraicJulia?

AlgebraicJulia is

- a family of open source software packages for applied category theory (ACT)
- written in the Julia programming language
- focused on technical computing for science and engineering

Organization of ecosystem:

- Catlab.jl: general framework for ACT
  - Semagrams.jl: interactive graphical editor for acsets
- Domain-specific packages, including
  - AlgebraicDynamics.jl: discrete and continuous dynamical systems
  - AlgebraicPetri.jl: Petri nets and epidemiology modeling
  - CombinatorialSpaces.jl: simplicial sets and the discrete exterior calculus (DEC)
  - Decapodes.jl: multiphysics simulation based on DEC

Aims of AlgebraicJulia

Present wave of ACT has been ongoing for 10+ years, featuring

- Monoidal categories and wiring diagrams
- Categorical databases
- Open systems via decorated/structured cospans
- ...and a lot more!

The goal of AlgebraicJulia is to make useful technologies out of this mathematics.

1. Build highly general software, based on CT abstractions, applicable to diverse domains
2. Instantiate these abstractions in specific scientific and engineering domains, in collaboration with domain experts
3. Interoperate between domains and applications using functorial constructions

Although it is a research project, AlgebraicJulia aims to be useful *today*.

Designing vs computing with categories

Category theory has impacted programming in at least two different ways:

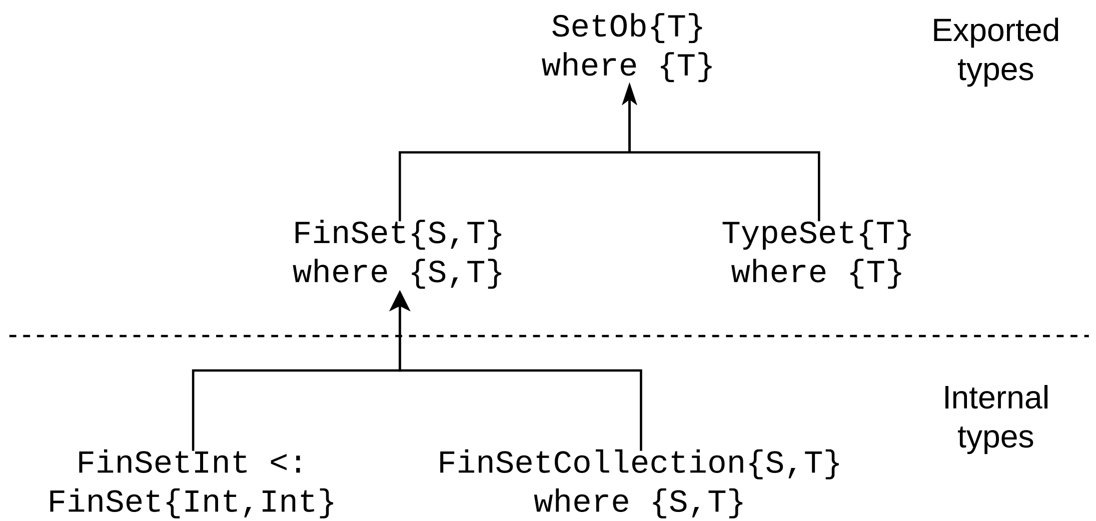
1. As a framework for **designing** programs and programming languages:
   - Categorical concepts model the *program* or *language*
   - In functional programming, types and functions are viewed as objects and morphisms in a category (e.g., Haskell as the "category" `Hask`)
   - Language features based on categorical constructions (e.g., monads in Haskell)
2. As a framework for **computing** grounded in data structures and algorithms:
   - Categorical concepts model the *subject-matter domain*
   - Ex: monoidal categories and string diagrams as a model of processes
   - Implies nothing about the host language

The two uses are not incompatible, but they are orthogonal.

Catlab and AlgebraicJulia are (mostly but not exclusively) about the second.

Example: sets and finite sets

Incomplete type hierarchy for **sets** in Catlab:



- No overlap with base Julia types `AbstractSet` and `Set` for unordered collections
- Although sometimes useful to treat Julia types as sets (`TypeSet`), no attempt is made to model the Julia programming language
- Emphasizes finite sets as a fundamental building block for more interesting objects
    - Especially algorithms for efficiently computing limits and colimits

Design space for computational category theory

Design space for computational CT is very large, encompassing:

- **Semantics**: Computations in specific categories and categorical structures
  - Grounded in specialized data structures and algorithms
  - Objects and morphisms given by *concrete data*, combinatorial or numerical
  - Combinatorial example: categorical databases
  - Numerical example: open dynamical systems based on ODEs
- **Syntax**: Computer algebra for CT
  - Categories given abstractly, e.g., presented by generators and relations
  - Algorithms for generic rewriting, symbolic or combinatorial
- **Proofs**: Proof assistants and automated theorem proving for formalized CT

Today, AlgJulia does a lot of semantics, some computer algebra, and no formal proofs.

Pitfall: mathematics $\ncong$ software

Mathematicians often hope to see math expressed in code in precisely the way they know and love it. However:

Mathematics can rarely be "isomorphic" to its software implementation.

This is true for many reasons:

- **Size**: Mathematical objects are very often infinite, but computers are finitary
- **Sameness**: Isomorphic objects, equivalent categories, ... are "the same" in math but in software these equivalences must be tracked and computed
  - Both objects and their presentations are important
- **Representation**: Most importantly, mathematics is not intrinsically algorithmic:
  - Data structures and algorithms are *additional content* going beyond the math
  - The same mathematical object (in the sense of strict identity) can be usefully represented by different data structures

As a result, implementing mathematics is a creative endeavor in its own right.

Example: categories of sets and finite sets

In Catlab, the category Set is not a single code construct but is more like the profunctor

$$\mathsf{FinSet} \longrightarrow\!\!\!\!\!\!\!+\;\; \mathsf{Set}$$

whose heteromorphisms are functions from finite sets to arbitrary sets.

Motivation:

1. functions between finite sets are given by finite data
2. functions from a finite set to a set are *also* given by finite data
3. functions between arbitrary sets must be given by rules/algorithms

Supported representations for functions `FinSet{Int}` $\to$ `TypeSet{T}` of type (2):

- vector with elements of type `T`
- arbitrary Julia function taking integers in $\{1, \ldots, n\}$ to objects of type `T`
- lazy composite of function of type (1) and function of type (2)

Each corresponds to a different Julia data type.

Example: Attributed C-sets

This viewpoint extends to our in-memory implementation of **categorical databases**, called *attributed C-sets* ("acsets"):

$$\begin{tikzcd}
S_0 \arrow[r, "S_\to", "\shortmid"{marking}] \arrow[d] & S_1 \arrow[d] \\
\mathsf{FinSet} \arrow[r, "\shortmid"{marking}] \arrow[ur, phantom, "\Downarrow"] & \mathsf{Set}
\end{tikzcd}$$

- **Schema** is profunctor $S = (S_\to \colon S_0 \nrightarrow S_1)$
- **Entities** and functional **relations** given by objects/morphisms of category $C = S_0$
- **Attribute types** given by objects of (discrete) category S_1
- **Data attributes** given by heteromorphisms $S_\to$

Cf. (Schultz et al 2017: "Algebraic databases")

C-sets in Catlab today

Even ignoring data attributes, C-sets (copresheaves on C) on a category C comprise a hopelessly broad set of mathematical structures to implement.

Today, Catlab supports finite-valued C-sets on a category C that is:

1. **Finite**: finitely many objects and morphisms
   - Examples: schemas for graphs, possibly reflexive and/or symmetric; whole-grain Petri nets; wiring diagrams
2. **Finitely presented**: finitely many objects but possibly infinitely many morphisms
   - Example: schema for discrete dynamical systems (free monoid on one generator)
   - In free case, infinite if and only if generating graph contains a cycle
   - In non-free case, determining finiteness is hard (in general, undecidable)

Currently, both classes are supported by the same mechanism: presentation of C by **generators and relations**.

Constructions that depend on finiteness are avoided.

C-sets in Catlab in the future

In the future, I would like to support C-sets on categories C that are not finitely presented but admit **finitary descriptions**. Examples:

- *2-computads*, the generators of free 2-categories

$$
\begin{tikzcd}
\bullet \arrow[r] \arrow[d] & \bullet \arrow[d] \arrow[d, bend left, ""{name=B}] \\
\bullet \arrow[r] \arrow[ur, Rightarrow] & \bullet
\end{tikzcd}
$$

- *2-opetopes*, a possible data structure for diagrams with (some) commuting cells
- *simplicial sets* and *semi-simplicial sets*, not truncated to finite dimension

What is a useful notion of "finitary description"?

This is an example of a design problem in computational category theory.

Principle: generic $\not\Rightarrow$ slow

Category theory offers very general abstractions for modeling.

- Abstraction can often cause performance overhead
- But, with the right techniques, it doesn't have to

Case study: attributed C-sets in Catlab

- C-sets are far more general than graphs
- Yet Catlab's graphs library (`Catlab.Graphs`), based on acsets, achieves performance comparable with state-of-the-art graphs packages (LightGraphs.jl)

This is achieved through a careful design utilizing Julia language features.

(Patterson, Lynch, Fairbanks, 2021: "Categorical data structures for technical computing")

Fast acsets: static vs dynamic

The key to good performance is appropriately separating phases:

- **Static**: computations that will be run many times should be **compiled** into performant machine code, e.g., for acsets:
  - Low-level accessors and mutators, associated with specific schema
  - Updating indices for fast reverse lookups, associated with schema + Julia type
- **Dynamic**: one-off computations should be evaluated at **runtime** to avoid overhead of compilation

These distinctions manifest differently in different languages:

- In *compiled* language like C/C++, strict separation between compile- and runtime
- In *interpreted* language like Python, the compilation phase is minimal
- In *just-in-time (JIT) compiled* language like Julia, static and dynamic phases are interwoven and compilation happens on-the-fly as needed

The latter is a powerful and flexible combination.

Fast acsets: types and metaprogramming

The low-level acsets API is implemented using "generated functions," a Julia-specific form of **metaprogramming**.

- Ordinary functions take and return data of specified types
- Generated functions take input types and return a Julia expression to be compiled
  - Expression may depend on static information (types)
  - Expression may not depend on runtime information (instances of types)

Thus, the ascet schema and index config must be packed into the Julia type for the ascet:

- Julia does not allow arbitrary dependence of types on values
- But it does allow values of certain primitive types in a sufficiently flexible way

Remark: A frontier in PL design is use of dependent types to *improve* performance by carefully managing the static/dynamic phase distinction.

Principle: separate syntax and semantics

The syntax-semantics distinction is basic to mathematical logic and computer science.

But category theory/categorical logic expands the reach of both:

- **Syntax** is about more than expression trees
  - From the categorical viewpoint, syntax is just another part of algebra
  - Example: string diagrams for morphisms in SMCs
  - Example: operads as algebraic gadgets for modular/hierarchical composition
- **Semantics** is about more than sets and functions
  - Functorial semantics allows theories to be interpreted in categories besides Set

Warning:

- I am not talking about Julia syntax (remember, we are not modeling the language)
- But rather bespoke syntax for the domain being modeled

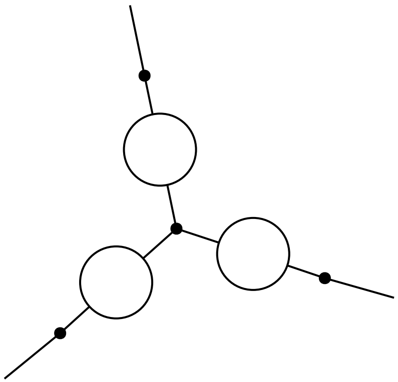
Julia syntax vs categorical syntax

Julia syntax: `@relation` macro for defining an undirected wiring diagram (UWD)

```
>>> using Catlab.Programs
    uwd = @relation (x,y,z) begin
        R(x,w)
        S(y,w)
        T(z,w)
     end;
```

Categorical syntax: UWD as combinatorial object (or visualization thereof)

```
>>> using Catlab.Graphics
    to_graphviz(uwd)
```



Julia syntax vs categorical syntax

Categorical syntax: UWD as combinatorial object (an acset)

```
>>> uwd
```

RelationDiagram{Symbol, Symbol} with elements Box = 1:3, Port = 1:6, OuterPort = 1:3, Junction = 1:4

Box	name
1	R
2	S
3	T

Port	box	junction
1	1	1
2	1	4
3	2	2
4	2	4
5	3	3
6	3	4

OuterPort	outer_junction
1	1
2	2
3	3

Junction	variable
1	x
2	y
3	z
4	w

One syntax, many semantics

Undirected wiring diagrams are a syntax for composing...

- Spans/data tables, as in a conjunctive query
- Pixel arrays/boolean tensor networks
- Structured cospans, such as
  - open graphs
  - open Petri nets (`AlgebraicPetri`)
  - open free diagrams in a category (`Decapods`)
- Open ODE systems via "resource sharing" (`AlgebraicDynamics`)

All of these are implemented in AlgebraicJulia (some more efficiently than others).

Syntax in Catlab

Catlab features two approaches to syntax:

1. **Generalized algebraic theories** (GATs)
   - "GATs = algebraic theories + dependent types"
   - Closely related to: essentially algebraic theories, finite limit theories
   - Theories and syntax presented in *biased* style
2. **"Combinatorial operads"**
   - *Unbiased* approach based on operads and operad algebras
   - Operad morphisms are combinatorial data structures, namely acsets
   - Important examples implemented in Catlab (`Catlab.WiringDiagrams`):
     - Directed wiring diagrams (DWDs)
     - Undirected wiring diagrams (UWDs)
     - Circular port graphs

Example: GAT for categories

Theory of categories (`Catlab.Theories`):

```
@theory Category{Ob,Hom} begin
  # Unicode aliases.
  @op begin
    (→) := Hom
    (⋅) := compose
  end

  # Type constructors.
  Ob::TYPE
  Hom(dom::Ob, codom::Ob)::TYPE

  # Term constructors.
  id(A::Ob)::(A → A)
  compose(f::(A → B), g::(B → C))::(A → C) ⊣ (A::Ob, B::Ob, C::Ob)

  # Equational axioms.
  ((f ⋅ g) ⋅ h == f ⋅ (g ⋅ h) ⊣ (A::Ob, B::Ob, C::Ob, D::Ob,
                                  f::(A → B), g::(B → C), h::(C → D)))
  f ⋅ id(B) == f ⊣ (A::Ob, B::Ob, f::(A → B))
  id(A) ⋅ f == f ⊣ (A::Ob, B::Ob, f::(A → B))
end
```

GATs and higher-dimensional algebra

GATs have some advantages:

- Correspond closely to typical definitions found in CT textbooks
- Easy to write down and to adapt to new situations
- Easy to define semantics in Julia types (`@instance` macro)

But for the higher-dimensional structures that are so prominent in ACT...

- Symmetric monoidal categories (SMCs)
- Hypergraph categories
- (Symmetric monoidal) double categories

...the biasedness of GATs is a huge inconvenience because it forces *arbitrary choices* when decomposing into primitive operations.

Motivating example: the *interchange law* for morphisms in an SMC:

```
(f ⊗ g) ⋅ (h ⊗ k) == (f ⋅ h) ⊗ (g ⋅ k)
    ⊣ (A::Ob, B::Ob, C::Ob, X::Ob, Y::Ob, Z::Ob,
       f::(A → B), h::(B → C), g::(X → Y), k::(Y → Z))
```

Principle: prefer combinatorial syntax

GAT expressions for morphisms in an SMC can be obtained by

- Human labor: user writes down the expression
  - Time-consuming and error-prone
  - Dealing with identities and swaps is particularly annoying
- Computer labor: function `to_hom_expr` converts DWD to morphism expression
  - Algorithmically complex and slow
  - Not all flavors of SMCs with extra structure are supported

Combinatorial operads and operad algebras bypass this problem entirely!

- Especially embraced in AlgebraicDynamics, maintained by Sophie Libkind

But figuring out these operads is not a mechanical process...

Open problem: What is a combinatorial operad, anyway?

Thanks!

Resources

- Website: `https://www.algebraicjulia.org`
    - Blog
    - Papers and talks
- GitHub: `https://github.com/AlgebraicJulia/`
    - Source code
    - Documentation
- Julia Zulip: `https://julialang.zulipchat.com`
    - `#catlab.jl` stream

Contributing

We welcome new contributors at all levels of experience.

Please reach out if interested!