Diagrammatic notations for interactive theorem proving

SHARDUL CHIPLUNKAR and CLÉMENT PIT-CLAUDEL, EPFL, Switzerland

Diagrams are ubiquitous in the development and presentation of proofs, yet surprisingly uncommon in computerized mathematics. Instead, authors and developers rely almost exclusively on line-oriented notations (textual abbreviations and symbols). How might we enrich interactive theorem provers with on-the-fly visual aids that are just as usable? We answer this question by identifying a key challenge: designing declarative languages for composable diagram templates, that provide good-looking implementations of common patterns, and allow for rapid prototyping of diagrams that remain stable across transformations and proof steps.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Human-centered computing → Visualization systems and tools**.

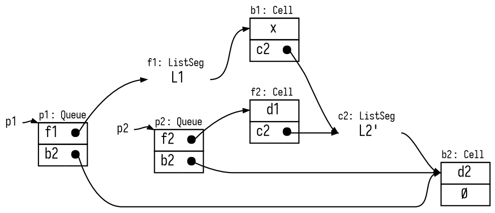
Additional Key Words and Phrases: diagrams, interactive theorem proving, diagramming languages

1 INTRODUCTION

Diagrams abound in proofs: in research papers, textbooks, documentation of proof developments, presentation slides, whiteboards, and classroom materials. Diagrams often illustrate or even constitute parts of the proof, especially in contexts where they are more natural, human-friendly representations than mathematical symbols or words. Figure 1 shows a particularly striking example in the context of a separation logic proof.

```
(sep (MCell f2 d1 c2) (sep (MCell p1
f1 b2) (sep (MCell p2 f2 b2) (sep
(MCell b1 x c2) (sep (MListSeg c2 b2
L2') (sep (MCell b2 d2 null)
(MListSeg f1 b1 L1)))))))
```

(a) As a Coq term, without notations.



(b) As a diagram with boxes, text, and arrows.

$$(f_2 \mapsto \langle d_1, c_2\rangle \quad \star\, p_1 \mapsto \langle f_1, b_2\rangle \quad \star\, p_2 \mapsto \langle f_2, b_2\rangle \quad \star\, b_1 \mapsto \langle x, c_2\rangle$$
$$\star\, c_2 \mapsto \langle\!\langle b_2, L_2'\rangle\!\rangle \quad \star\, b_2 \mapsto \langle d_2, \mathrm{null}\rangle \quad \star\, f_1 \mapsto \langle\!\langle b_1, L_1\rangle\!\rangle)$$

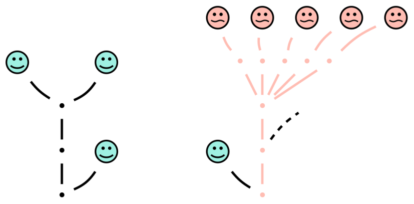
(c) In mathematical notation.

Fig. 1. Memory state while concatenating double-ended queues, adapted from [Charguéraud 2014].

But when proofs are performed with a computer instead of on paper, diagrams are glaringly absent. The term presented as (1a) by default in Coq is turned into the more readable form (1c) by using Coq's textual notation system to capture common patterns, but this is still a far cry from the diagrammatic representation like Figure 1b one would use in practice to understand and develop a proof. If diagrams are an important component of the human development of proofs, why are

Authors' address: Shardul Chiplunkar, shardul.chiplunkar@epfl.ch; Clément Pit-Claudel, clement.pit-claudel@epfl.ch, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.

https://doi.org/10.5075/epfl-SYSTEMF-305144



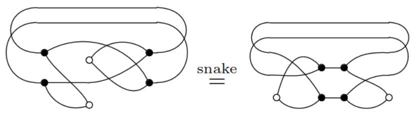
(a) Diagrams can represent the objects which the proofs are about, as in the case of hydra diagrams, a running example we will use throughout this paper. Hydra diagrams were introduced as an object of mathematical study in [Kirby and Paris 1982] and the figure to the left is from the book *Hydras & Co.* by [Castéran et al. 2022].

```
Inductive Hydra: Set := node: Hydrae -> Hydra
with Hydrae: Set := hnil: Hydrae | hcons: Hydra -> Hydrae -> Hydrae
```

```
(node (hcons (node (hcons (node (hcons (node hnil) (hcons (node hnil)
  hnil))) hnil)) (hcons (node hnil) hnil)))
```

(b) For reference, the Coq definition of hydras in *Hydras & Co.* as a mutually inductive type, and the term corresponding to the diagram to the left above (Figure 2a). Intuitively, a *hydrae* is a set of hydras as a linked list constructed with hcons and hnil, and a *hydra* is a node with a set of children (hydrae). A hydra with no children, i.e. (node hnil), is drawn as a smiley.

(c) Or, diagrams can represent entire proof states, such as for separation logic in Figure 1b.



(d) Or, they can also constitute the proof itself, with rewriting rules for diagrams that correspond to axioms of the underlying systems; recent work on diagrammatic calculi with string diagrams for Boolean satisfiability is a compelling example (figure taken from [Gu et al. 2023]).

Fig. 2. Diagrams can serve a few different roles in representing a proof or the objects it deals with.

they missing from proof assistants? How might we enrich proof assistants with diagramming capabilities?

In this paper, we attempt to approach these questions by first looking at alternative representations that *are* widely used in proof assistants, namely, *textual notations* or *macros* (§ 2). We identify some properties of these notation systems that we believe explain their success, and then briefly review some existing diagramming tools (not necessarily linked to proofs) (§ 3), noticing whether and how the visual counterparts of these properties manifest. With this background, we present some preliminary experiments in diagrammatic notations in which we try to demonstrate some of our desired properties (§ 4). Lastly, we summarize our proposed approach to building a diagramming system for interactive theorem proving and the main questions we hope to address (§ 5).

1.1 What is a diagram?

Before we proceed further, we will limit the scope of what we consider to be a diagram with a few qualifiers (albeit not a rigorous definition). A diagram is a *schematic representation* of entities, sub-entities, and their relationships: it follows a *schema* that determines what primitive components can be used in the diagram, what they represent, and how they can be combined, akin to a grammar for a language. This distinguishes it from a *pictorial* representation that aims to be photorealistic or to approximate reality, such as a polygon mesh. A diagram only includes elements that directly contribute to understanding or capture some information about the object. A diagram is also interpreted schematically. E.g., a binary tree drawn with a computer, and the same drawn by hand with slight imperfections, nevertheless represent the same object following the same schema. We also observe that we often want diagrams to be rendered rigidly: elements may need to be aligned or a tightly constrained size, spacing may need to uniformly reflect conceptual organization, etc.

2 WHAT MAKES TEXT NOTATIONS SUCCESSFUL?

One might think that the experience of interactive theorem proving is somehow different, and does not require alternative representations for complex objects—but we know that it does, because users heavily rely on notations, implicit arguments, and other devices to make proof goals more readable. For instance, the Coq keyword `Notation` (used to define a new text notation) is used 234 times in the sources of the *Hydras* book, among 5,397 definitions and theorems, or about once every 23 definitions/theorems. This idea can be taken very far to enable representations quite different from the original; e.g., `lean-verbose` [Massot 2023] consists of 642 lines of Lean macros originally developed to accompany 2,502 lines of notes and exercises in an introductory semester-long course on proof assistants, that 'translates' a proof script to look more like a natural language derivation.

Lean macros and Coq notations are two examples of more generally allowing the user to define domain-specific representation languages for proofs or goal states. What explains their widespread success? We make five observations: that they describe how notations should be *composed* based on the *structure* of conceptual objects, rather than describing individual notated terms; that they strike a good *balance* between cost (or ease of use) and benefit; that they are *canonical* representations, in direct correspondence with the objects being represented; that they allow for *continuous*, i.e. minimal and easy to follow, changes in the representation of an object over proof steps; and that common specification *idioms* are implemented by the system in a way that makes them look good.

1. Structure-driven compositionality. Notations are compositional: their parts have independent meaning and can be combined in systematic ways. Further, the composition of notations is driven by the structure of the representee, or in other words, the notation of the whole is composed of the notations of its parts mirroring how the whole is composed of those parts. Thus, instead of converting individual objects into prettified notations, this lets the user specify a grammar for notating *any* object in the domain, and the system instantiates it as needed. This productive grammar is sometimes described as the inverse of the analogously systematic definition of a parser.

2. Balance between costs and benefits. The language for specifying notations is simple to get started with and conveniently integrated into the proof assistant, allowing for rapid prototyping. Very useful notations can be specified in just a couple lines while more complex notations can be layered on progressively [Pit-Claudel and Bourgeat 2021]. No additional compilation steps are needed, and once defined, there is no noticeable overhead of using them. This makes notations worth it, not just for teaching, presentations, or other settings where the developer is willing to put in additional effort for clarity, but even for ordinary proofs developed by and for a single person.

3. Canonicity. Modulo what the user chooses to ignore in the specification, a notation is in direct, transparent correspondence with the underlying object. Equivalent objects have identical notations. Moreover, the same conceptual transformation applied to two objects produces the same transformation of their notations. Notations try to avoid introducing degrees of freedom that the user would have to quotient out to understand what is being denoted.

4. Continuity. When an object evolves over proof steps, its representation changes in such a way that the user can easily tell what has changed, and no more than it has to. (This is distinct from canonicity because the canonical representation of the next step of an object in a proof could conceivably be entirely different from the original, but the limited flexibility of textual notations avoids this in most cases.) The structure-driven nature of notations often automatically makes them continuous, because when a part of an object changes, only the corresponding part of the notation changes, but this is harder to achieve when global constraints are at play, such as when reflowing text or re-laying out a diagram.

5. Idioms that look good. Identifying some common idioms used in many different textual notations, the system shoulders the burden of implementing them with good defaults such that user-defined notations that use these idioms will look reasonably good with little additional effort. In Coq, for instance, some idioms the notation system provides are:

- *associativity* and *precedence*, to disambiguate, say, $a \star b \star c \rightsquigarrow d$ after defining $\star$ and $\rightsquigarrow$;
- *printing format*, to control line breaking and indentation; and
- *recursive notations*, to specify notations like `[a, b, ..., c]` for arbitrarily long literal lists.

Without such idioms, a developer of a set of related notations would have to either use verbose, less readable notations, e.g. with extra parentheses and other disambiguating syntax, or expend significant effort implementing their own versions, likely less pretty or capable.

We believe these properties of textual notation systems are key to their usability and widespread success. The fact that alternative textual representations are widely used in proof assistants, and diagrammatic notations in traditional 'paper' proofs, prompts us to draw inspiration from the former to enable the latter in proof assistants, too. With this perspective, we turn to a brief survey of existing diagramming tools as inspiration or potential candidates for use in a proof assistant.

3 A GUIDED TOUR OF DIAGRAMMING TOOLS

Many diagramming tools exist today, ranging widely in aims and domains. We take a look below at a few that may have important lessons or serve as starting points. Along the way, we try to identify visual counterparts of the properties from § 2. We disclaim upfront that this survey is necessarily incomplete given the sheer number of existing tools, about which we hope to foster exchange.

A recent tool with aims similar to ours is Penrose [Ye et al. 2020] for mathematical diagrams. Penrose takes three pieces of input: a DSL definition for a "domain" (e.g. set theory), a collection of "substance" statements in that DSL (e.g. definitions of some sets and subset relations between them), and a constraint-based, declarative "style" specification (e.g. subsumed sets are to be drawn as contained circles). It then produces as output a diagram representing the substance in the specified style. This is exactly what we meant by **structure-driven compositionality** previously. Additionally, Penrose provides some **idioms** for use in stylesheets, such as to enforce the containment of a shape within another or to optimize an unknown parameter subject to constraints, that make diagrams specified with these idioms have the desired visual characteristics without much effort from the user. On the other hand, the tool explicitly does not aim for **canonicity** or **continuity**. Diagrams are typically underspecified, and Penrose produces multiple valid instances so that the user can choose the 'best' one for their use case, or even discover examples they hadn't foreseen. Producing incremental diagrams corresponding to successive changes to the substance file is thus not a primary intended application. Rather, Penrose is most useful when producing diagrams is, in itself, the user's goal, whether for exploration or to include in static presentation materials, which justifies the additional **cost** of heavy use of optimization.

We are not the first to point out the need for visualization in proofs, either as part of a proof presentation framework like Alectryon [Pit-Claudel 2020] for Coq, or built into the IDE like ProofWidgets 4 [Nawrocki et al. 2023] for Lean. Fundamentally, these tools provide an interface to standard web and graphics technology: scripting, canvas drawing, SVG, CSS, etc. This makes it possible to produce quite sophisticated visualizations, like an animated Rubik's cube [ibid.]. ProofWidgets in particular demonstrates the demand for and usefulness of diagrammatic notations and smoothly integrates them into the proof assistant workflow. However, these tools do not aim to provide their own **language** or **idioms** atop the underlying technologies for specifying visualizations. They let the user build diagrams—that may in fact have our desired properties, or may just be ad hoc pictures—but hold no opinion about *what* makes a useful diagram, nor *how* to make such diagrams easier to build. ProofWidgets and Alectryon extensions are promising tools when the developer is willing to put in significant **effort** to, essentially, write a custom diagramming program, and have seen some success already (such as with commutative diagrams in Lean's `mathlib`), but are not meant to be lightweight notation systems.

An important domain to study is *data and graph visualization* tools, including the highly successful, general-purpose D3.js [Bostock et al. 2011] and GraphViz [Gansner and North 2000], as well as more specialized ones like Mermaid.js [Mermaid contributors [n. d.]] (software documentation) and Cytoscape [Shannon et al. 2003] (network data, particularly biological). What these tools have in common is that they preserve the separation between the content (raw data or a graph) and its rendering generated from a **compositional structure-driven** (data-driven) grammar, and that they provide a limited set of possible renderings with customizable styling. For instance, D3 has a few dozen libraries of visualization components, like force-directed layouts and animation transition interpolators, that the user can style freely and compose as permitted by the grammar. These are perfect instances of common **idioms** identified and implemented at the system level to produce pretty results by default (with the caveat that D3 consists almost entirely of idioms and is designed for **high-effort, high-reward** use, not rapid prototyping). Likewise, GraphViz exposes different graph layout algorithms best suited for different contexts, with global objectives like minimizing the number of edge crossings or aligning nodes according to a hierarchy. It hence leaves the question of representation **canonicity** to the user, whereas once a visualization is defined in D3, the representation it produces of any data is canonical almost by definition, being strictly data-driven. Lastly, **continuity** is an interesting consideration: some visualizations, like scatterplots, are easy to change continuously when a new data point is added, whereas for others, like force-directed graph layouts, the expected behavior is to globally recompute the visualization even for a structured, compositional change in the data. Graph layout or D3-style primitives will nonetheless likely be subproblems a proof diagramming tool would have to solve.

PGF/TikZ [Tantau 2023] and Asymptote [Bowman and Hammerlindl 2008] are two prominent examples of a larger category of *programmable drawing* tools, also including libraries in many general-purpose languages, like Diagrama [Florence [n. d.]] in Racket. These tools provide vector-graphics–like primitives such as curves, stroke & fill settings, and linear transformations, along with some general programming features, like variables and loops. They also sometimes provide default implementations of common **idioms**; for example, TikZ defines "nodes", and "anchors" named by cardinal directions for every node that connectors can attach to. However, programs written in TikZ and its kin do not typically bear much resemblance to the conceptual **structure** of the object their outputs depict. For TikZ in particular, this has given rise to a plethora of libraries and graphical interfaces on top of the base language. Some bridge the gap for specific domains, like forest [Živanović 2017], that uses macros to convert tree-like input into tree diagrams. Others sidestep the problem by providing a traditional manual drawing interface that just happens to produce TikZ output, like GeoGebra [GeoGebra Contributors [n. d.]] (geometry) or TikZiT [Kissinger 2020] (string diagrams, etc.). The existence of these tools highlights the need for a more **compositional** language-based approach to diagramming, but also for extensibility, because it is a strength of TikZ (and TeX) that a sophisticated library with a custom input language like forest can be built on top of it at all.

3.1 What do we want from diagrammatic notations?

Drawing from our brief review above, we are ready to advance a tentative hypothesis for what properties we might want of a diagrammatic notation system for proof assistants.

Conceptual diagrams should be specified in a declarative language (as opposed to graphics specified as imperative drawing instructions) that lets us compose and operate on diagrams according to a schema reflecting the structure of the underlying objects. The system should instantiate such **structure-driven, compositional specifications** as concrete renderings of objects.

Defining a notation should be easy enough, and the resulting diagrams should be pretty enough, that a proof developer finds it **worth it to quickly prototype notations** for themselves even in developments not presented to anyone else. To this end, a diagram specification language may need to incorporate some amount of constraint-based specification, graph layout, and general-purpose programming. The system should identify **common idioms** used in diagrams and provide implementations that are pretty by default, i.e. that have the expected visual characteristics without much additional effort from the user.

The stability of a diagrammatic representation should be a fundamental concern: equivalent objects or operations should have immediately identifiable diagrams or diagrammatic transformations (**canonicity**), and the evolution of the diagram of an object over small step-by-step changes in a proof should be easy to follow (**continuity**). However, we do not know the final state of the proof when we start diagramming. A usable solution might use animations to communicate changes while maintaining stability or leave extra space in anticipation.

4 PRELIMINARY EXPERIMENTS WITH HYDRA DIAGRAMS

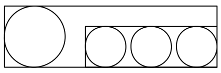
We present two preliminary case studies in diagramming the hydras from the book *Hydras & Co.* [Castéran et al. 2022]. Each aims to illustrate different properties we want of diagramming systems, as identified in § 3.1, but we are only advocating for the *general realization* of such properties in diagrammatic notations, not for our specific implementation choices. We are *not* proposing a diagram specification language or a notation system. Nonetheless, the (Racket) source code for our implementation is available on request.

Before we begin, recall the mutually inductive type definition for hydras in Coq and the example from Figure 2b. Note that syntactically, the children are ordered, but semantically, they're not.

4.1 Trees (similar to Figure 2a)

We start with the observation that tree diagrams for hydras are recursively compositional in much the same way that hydras are themselves: a node in a tree diagram has children that are themselves trees. In fact, we can even define a *hydrae diagram* for a set of children, where the recursive diagrammatic counterpart of `hcons` is to juxtapose a child hydra diagram beside existing children's diagrams. Thus the overall structure of our diagrammatic notation definition for hydra diagrams will mirror the original type definition.

Beyond the shapes to be drawn, we also need to keep track of the *bounding box* of each (sub-)diagram to align and juxtapose them, and the *endpoints* for the connecting lines between a node and its children. These computations, too, fit nicely into the recursive diagram definition. Putting it all together:


(a) Horizontal, bottom-aligned juxtaposition, with bounding boxes.

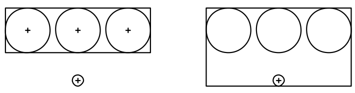

(b) Vertical, center-aligned juxtaposition, with bounding boxes, with exposed endpoints ('+') before and after the operation.

Fig. 3. Hydra diagram juxtapositions, without connecting lines.

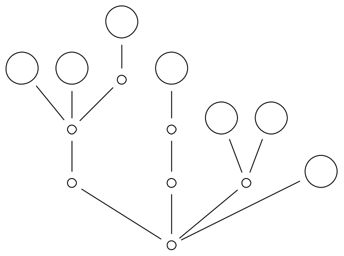
- The diagram of `hnil` is empty. Its bounding box is empty and it exposes no endpoints.
- The diagram of `(hcons h hs)` is the horizontal, bottom-aligned juxtaposition of the diagrams of `h` and `hs`. Its bounding box has width equal to the sum of the widths of its

two component diagrams, plus spacing, and height equal to the greater of their heights. It exposes all endpoints exposed by its components. See Figure 3a.

- The diagram of `(node hs)` is:
  - When the diagram of `hs` is empty, a circle, with a bounding box as big as its diameter.
  - Else, the vertical, center-aligned juxtaposition of a small dot and the diagram of `hs`. The total bounding box is computed as for `hcons`. See Figure 3b. Here, the endpoints become relevant: we draw connecting lines (not pictured) between the dot and each endpoint of the hydrae diagram (leaving some margin at the endpoints as a stylistic preference).

  The only endpoint exposed is the circle or dot's center.

Figure 3 illustrates juxtaposition operations without the connecting lines. Figure 4a shows an example of a hydra diagram produced by the above procedure.


(a) As a tree.

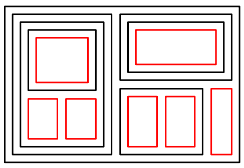

(b) As nested boxes.

Fig. 4. Two diagrammatic notations for the same hydra.

4.2 Boxes (as in Figure 4b)

Next, we attempt a different diagrammatic representation of hydras, albeit still with a recursive compositional structure, using nested boxes. Each box represents a node and contains the boxes for its children. A node with no children is represented by an empty box, colored red for style.

This diagram schema has many more degrees of freedom than the trees we saw previously. There are many different possible layouts of child boxes inside a parent box, even when we only consider layouts that completely fill available space; and then many possible relative distributions of space among the children; and we are still free to choose the size of the outermost box. Note that we ignore the (semantically irrelevant) ordering of children. Consequently, more implementation decisions are left to us, so what we present is only one set of choices among many reasonable ones.

We start by noticing that the layout of a box and its direct children can be neatly described using a *grid.* A grid is specified by relative proportions of the rows and columns and the internal spacing. Then, for each child, we specify its location and span in terms of grid cells. The layout so far has been specified purely in relative terms; to actually draw it, we pass as arguments a total width and height, and the absolute positions and dimensions of all the children are computed and passed on to them. We also define that to *transpose* a grid layout is to simply treat rows as columns and vice versa, without changing the width or height, nor rotating or reflecting any elements. This is all illustrated in Figure 5.

Now, we wish to specify a diagrammatic notation for hydras as nested grid layouts, so we must take some decisions to resolve the implementation freedom mentioned earlier. We proceed by informally describing a few desired æsthetic properties and how we implement them algorithmically.

- Leaves of the diagram should be roughly the same absolute size regardless of what total height and width the diagram is drawn at. But we cannot control the size of the leaves directly, so we take a different approach, assigning a *weight* to every (sub-)diagram. All leaves have unit weight; the weight of a node is proportional to the sum of the weights of its children; and space is allocated among siblings per their weight.
- The above still does not nail down an arrangement of the siblings, and we would like the arrangement to not be too visually complex. We find that recursively dividing a box into

```
(define gl
  (new grid-layout% ; columns  rows
    [grid-defn      '((1 2 1)  (3 1))]
    ; start row, start col, row span, col span
    [item-defns `((0 1 1 2 ,box))]
    [gap 16] [padding 10]))
(send gl draw! 400 150)
(send (send gl transpose) draw! 400 150)
```

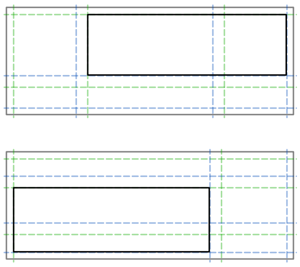


Fig. 5. A grid specification and its rendering.

quads until there are enough to lay out all the children produces grids that are both simple enough, and for which it is easy to ensure that space is allocated proportionally to weight.

- Boxes should not be too long or too wide. Again, we cannot control the sizes of boxes directly, but we can transpose a grid layout at any level in the hydra depending on the computed aspect ratio. We can also switch to a linear layout if the aspect ratio is too extreme.

4.3 Evaluation

How does this experiment stand up to the desired properties we stated earlier about diagramming systems? First of all, these are certainly **structure-driven, compositional specifications** for producing diagrams from the underlying objects. However, this is a Racket mini-library rather than a notation system, and we have not yet explored how we might design a true notation *language* (e.g. using Racket macros), nor how we might integrate with a proof assistant.

Our alternate notation for hydras (as nested boxes) highlights three features we would expect from a fully-fledged diagrammatic notation system. First, it is feasible to switch between alternate notations, as the notation is not built into the object. Second, we are able to let our informal stylistic preferences guide an algorithmic resolution of the additional degrees of freedom, building on top of the abstractions provided by the system. Third, these diagrams underscore the separation between abstract and concrete: a nested boxes diagram can be rendered in any (sufficiently large) size, adapting its layout at each level of hierarchy, all from one high-level declarative specification.

Another outcome of this experiment is that we confirm the utility of some **common diagrammatic idioms**, offering evidence that structured diagramming may be more broadly applicable:

- the family of aligned juxtaposition operators, `PQappend`, that juxtapose their argument list of diagrams along the `P` axis (e.g. horizontal) with the `Q` positions aligned (e.g. top-aligned);
- the notion that each (sub-)diagram exposes a set of endpoints for connectors, and that the endpoints of a composition of diagrams can be a function of their endpoints;
- intuitive specifications of grid layouts that can be nested, transposed, and arbitrarily resized;
- two kinds of 'drawable' object interfaces, one that can try to fit its concrete rendering within given dimensions, and one that only reports on the dimensions it ultimately occupied.

Two idioms we haven't yet devised a clean interface for, that might yet be common and sometimes desirable in diagrams, are the automatic layout of many siblings in a grid, and making full use of available space for trees by "packing" (c.f. the `forest` TikZ package [Živanović 2017], keeping track of the spans of diagrammatic elements with more fidelity than just rectangles).

The 'library' components above (inasmuch as we can make a distinction between library and user code at this early stage) took 178 significant lines of Racket code (LOC). Then, the 'user-defined notations' for tree diagrams took 18 LOC, and for nested box diagrams, 76 LOC. This **cost** is proportionate to the diagrams' flexibility.

Our last remark is about the **continuity** of the two notations for hydras. (We paid no explicit attention to continuity in our prototypes, so the following are more observations than evaluations.) Figure 6 shows a typical evolution of a hydra in both notations without assuming the layout of the final state at the start. We see that it is easier to follow the evolution of the trees than of the boxes as there are fewer visual changes unrelated to the conceptual ones (nested box notation is also less **canonical** than tree notation); but the boxes can maintain a fixed overall size, while the sizes of the trees are dictated by their structure.

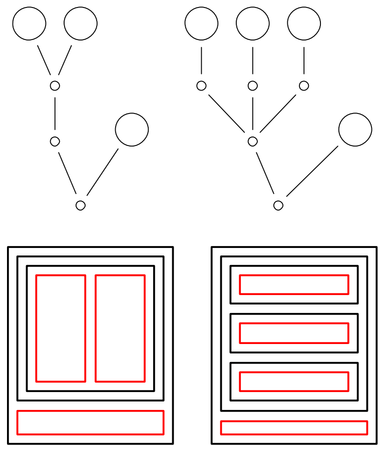


Fig. 6. The evolution of a hydra following its mathematical theory.

5 CONCLUSION & FUTURE WORK

Diagrams are essential to the human development of proofs, so we would like our computerized proof assistants to have them, too, as diagrammatic notations. For inspiration, we turned to an existing mode of representation in proof assistants—text notations—and to existing diagramming tools with varying aims. Our observations helped us hypothesize tentative criteria for a good diagrammatic notation system: **structure-driven, compositional specifications** in a declarative style should turn objects into conceptual diagrams; these specifications should be **low-cost** and allow for rapid prototyping and use; **canonical** notations should give equivalent objects and transformations equivalent diagrammatic counterparts, and **continuous** notations should make changes easy to follow; and the system should provide default implementations of **common diagrammatic idioms** that look good. We then experimented with two styles of diagrams for a particular domain (hydras) to try to illustrate some of these properties.

Many fundamental questions remain unanswered. For one, recall Figure 2 from our introduction—one of those figures is not like the others. So far, we have been working under the assumption that the objects we are representing have tree-like representations. String diagrams, however, are not trees, but general graphs, and we do not have good answers for them yet. But existing tools like GraphViz do, and we plan to study and incorporate their techniques.

The other major open questions concern the *workflow* in a proof assistant using these tools. What can we learn from projects like Lean ProofWidgets [Nawrocki et al. 2023] about usability and seamless prover integration? Apart from visualizing output, should we also visualize input, or allow for diagrammatic input, like [Andersen et al. 2020]? Should the user be able to perform computations in purely diagrammatic calculi? How should we allow the user to selectively elide irrelevant parts of a large diagram or define ad hoc extensions to the grammar? What are the usability considerations for our diagram language design, and how should similar existing designs, such as in Penrose [Ye et al. 2020] and GraphViz [Gansner and North 2000], inform our work?

REFERENCES

Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 222:1–222:28. https://doi.org/10.1145/3428290 https://dl.acm.org/doi/10.1145/3428290.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-driven Documents. *IEEE Transactions on Visualization & Computer Graphics (Proceedings of InfoVis)* (2011). http://vis.stanford.edu/papers/d3.

John C. Bowman and Andy Hammerlindl. 2008. Asymptote: A Vector Graphics Language. *TUGboat* 29 (2008). https://www.math.ualberta.ca/~bowman/publications/asyTUG.pdf.

Pierre Castéran, Jérémy Damour, Karl Palmskog, Clément Pit-Claudel, and Théo Zimmermann. 2022. Hydras & Co.: Formalized Mathematics in Coq for Inspiration and Entertainment. In *Journées Francophones Des Langages Applicatifs:*

JFLA 2022. https://hal.science/hal-03404668.

Arthur Charguéraud. 2014. Lecture Notes in Separation Logic. https://www.lri.fr/~marche/MPRI-2-36-1/2014/seplogic0.pdf.

Spencer Florence. [n. d.]. Diagrama: A Diagram Drawing Library. https://docs.racket-lang.org/diagrama/index.html.

Emden R. Gansner and Stephen C. North. 2000. An Open Graph Visualization System and Its Applications to Software Engineering. *Software: Practice and Experience* 30, 11 (2000), 1203–1233. https://doi.org/10.1002/1097-024X(200009)30:11%3C1203::AID-SPE338%3E3.0.CO;2-N https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

GeoGebra Contributors. [n. d.]. GeoGebra. https://www.geogebra.org/.

Tao Gu, Robin Piedeleu, and Fabio Zanasi. 2023. A Complete Diagrammatic Calculus for Boolean Satisfiability. *Electronic Notes in Theoretical Informatics and Computer Science – Proceedings of MFPS XXXVIII* 1 (Feb. 2023). https://doi.org/10.46298/entics.10481 https://entics.episciences.org/10481.

Laurie Kirby and Jeff Paris. 1982. Accessible Independence Results for Peano Arithmetic. *Bulletin of the London Mathematical Society* 14, 4 (July 1982), 285–293. https://doi.org/10.1112/blms/14.4.285 http://doi.wiley.com/10.1112/blms/14.4.285.

Aleks Kissinger. 2020. TikZiT. https://tikzit.github.io/.

Patrick Massot. 2023. Lean Verbose. https://github.com/PatrickMassot/lean-verbose.

Mermaid contributors. [n. d.]. Mermaid | Diagramming and Charting Tool. https://mermaid.js.org/.

Wojciech Nawrocki, Edward W Ayers, and Gabriel Ebner. 2023. An Extensible User Interface for Lean 4. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs))*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:20. https://doi.org/10.4230/LIPIcs.ITP.2023.24

Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Virtual USA, 155–174. https://doi.org/10.1145/3426425.3426940 https://dl.acm.org/doi/10.1145/3426425.3426940.

Clément Pit-Claudel and Thomas Bourgeat. 2021. An Experience Report on Writing Usable DSLs in Coq. In *CoqPL'21: The Seventh International Workshop on Coq for PL*, Assia Mahboubi and Amin Timany (Eds.). https://pit-claudel.fr/clement/papers/koika-dsl-CoqPL21.pdf.

Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. 2003. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research* 13, 11 (Nov. 2003), 2498–2504. https://doi.org/10.1101/gr.1239303 https://genome.cshlp.org/content/13/11/2498.

Till Tantau. 2023. The TikZ and PGF Packages. https://www.ctan.org/pkg/pgf.

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Transactions on Graphics* 39, 4 (Aug. 2020). https://doi.org/10.1145/3386569.3392375 https://dl.acm.org/doi/10.1145/3386569.3392375.

Sašo Živanović. 2017. The Forest Package for LaTeX/TikZ/PGF. https://www.ctan.org/pkg/forest.