Automatic Heap-Memory Diagrams for Separation-Logic Proofs

Yawen Guan, Shardul Chiplunkar(✉), and Clément Pit-Claudel

School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
{yawen.guan,shardul.chiplunkar,clement.pit-claudel}@epfl.ch

Abstract. Separation-logic proofs of heap-manipulating programs require careful accounting of objects and pointers in memory. On paper, these proofs are often accompanied by heap-memory diagrams that help authors and readers track the evolution of the program's abstract state. However, users of interactive theorem provers must instead work with plain-text notations that obscure object relationships. This paper presents the first automatic visualization library for separation-logic heap predicates, mimicking hand-drawn diagrams found in published materials. Four key features make the library practical. First, it supports animating across proof steps. Second, it offers users a DSL to specify how custom predicates should be visualized. Third, it is straightforward to port to new separation logic frameworks. And fourth, it can be used in browsers and IDEs, during and after proof development. We demonstrate these features by implementing support for CFML and Iris and integrating with Alectryon and VsRocq.

Keywords: Proof assistants · Visualization · Rocq

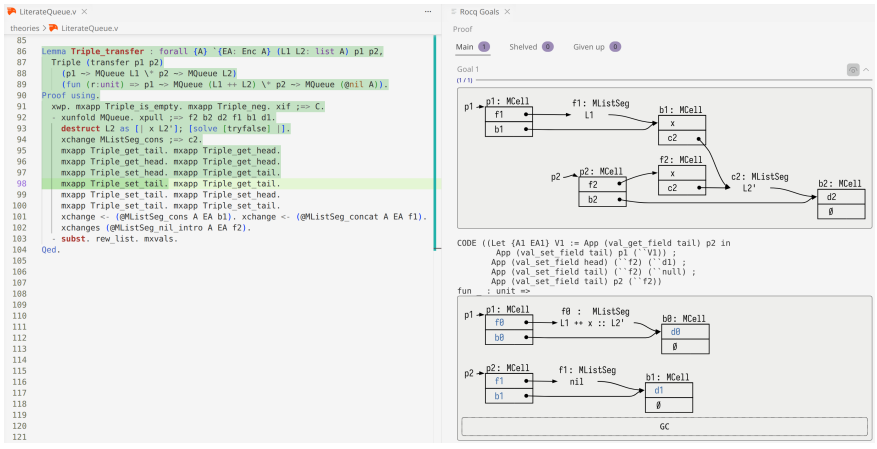

Fig. 1. One step in a proof about queues in CFML [8], presented with the VsRocq plugin [19] in VS Code extended with our visualization library. On the left is the proof script, and on the right, the proof state. (The goal context is collapsed for readability.)

1 Introduction

Since its introduction over two decades ago, separation logic has become a keystone of program verification [27,28,37]. At its core, the logic is an extension of Hoare logic designed for reasoning about programs that use shared mutable data structures, or in other words, that manipulate heap objects and pointers; in addition to *pure* predicates about values, pre- and post-conditions assert that the heap has a certain structure of objects and pointers among them.

Such *heap predicates* in separation logic have a natural diagrammatic representation as pointer graphs (e.g. Fig. 1). From early papers [37] to modern teaching materials [9], pointer graphs are widely (but informally) used to develop and explain proofs. Just like program state visualizations help a programmer keep track of the concrete state at a given point in a program's execution [41], these heap diagrams help a proof user keep track of the symbolic execution state beyond what conventional algebraic syntax affords.

Yet, when separation logic proofs are performed in an interactive theorem prover, visualizations are absent. This gap likely has a more general cause: proof tools do not yet have convenient facilities to specify diagrammatic notations or to control how they evolve over proof steps [11]. Instead, common mechanizations of separation logic like Iris [22] and CFML [8] extensively use textual notations to mimic on-paper syntax, but do not visualize heap predicates.

Automatic visualizations on a computer may even be easier to follow than diagrams on paper and whiteboards, if transitions between proof steps are animated. Animation avoids having to partially erase and redraw diagrams, which is tedious for large or repeated changes and impossible in print or other static media.

In this paper, we present the first software library for automatically visualizing and animating symbolic program states when reasoning about memory in separation logic. Its key features are:

- It supports common predicate combinators ($\exists, \forall, \ulcorner P \urcorner, \mathrel{-\!\!*}$, etc.).
- It visualizes data-structure representation predicates as records, with pointers depicted as arrows, configurable by the user with a convenient data description language (§2.2). E.g., in Fig. 1, the predicate `p1 ⇝ MCell f1 b1` that describes a pair at location `p1` containing the values `f1` and `b1` is visualized as a record with two fields, which are themselves pointers.
- It can be used with any mechanization of separation logic, because it takes conventional textual syntax as input, which is easy to remap with notations. For instance, we support the logics of Iris, CFML, and the Separation Logic Foundations textbook [10].
- It can be used in any HTML- and JavaScript-based user interface. We already integrate with two popular proof tools, the VsRocq plugin [19] for VS Code (e.g. Fig. 1) and the literate proof tool Alectryon [32] (e.g. Fig. 2).

Our library is open source and available online. Although we only have the space to show a handful of distinct visualizations in this paper, we have validated the usefulness of our library on additional examples, including a verified list API in CFML that is available for browsing online at systemf.epfl.ch/etc/sepviz/.

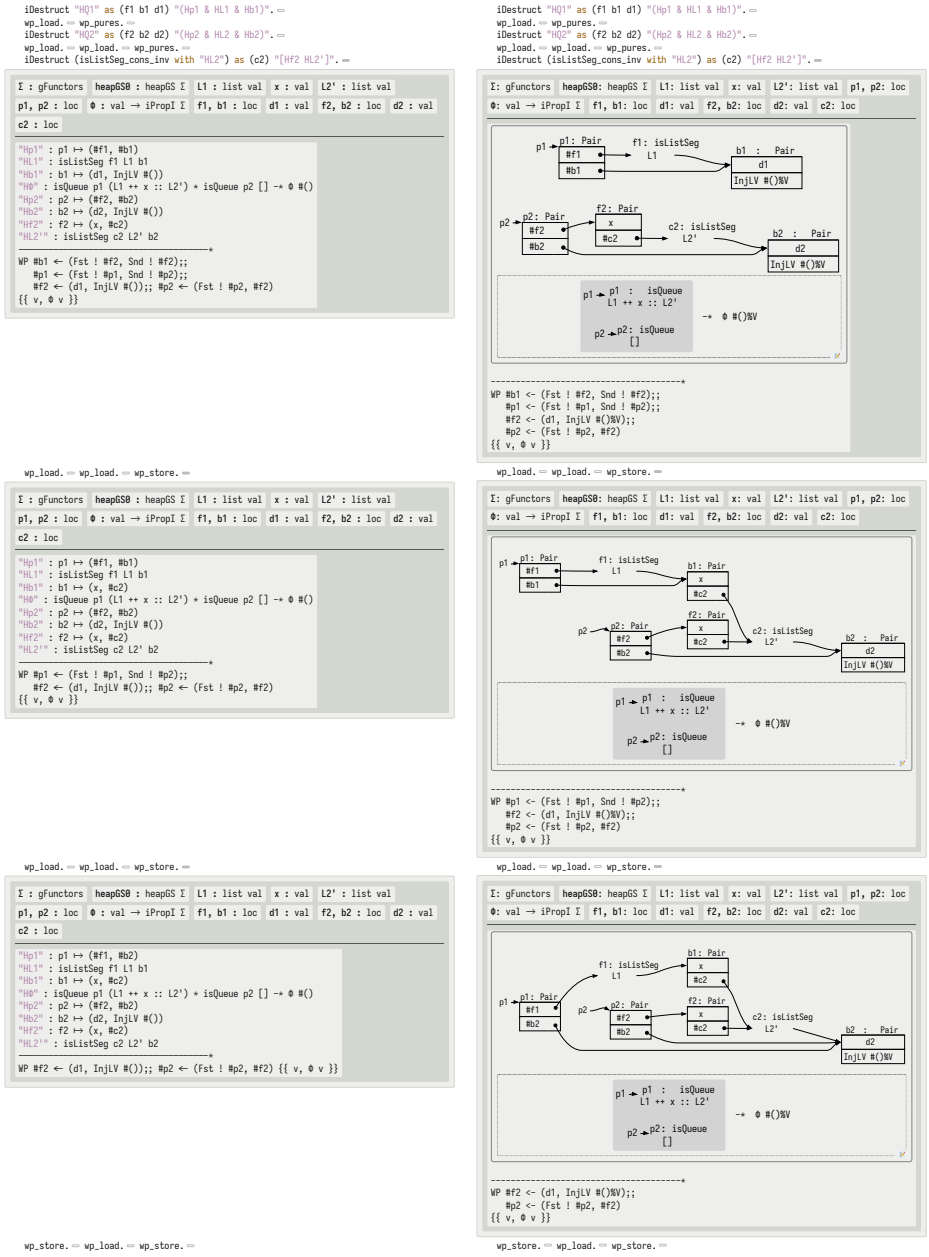

Fig. 2. An excerpt from a proof about queues using Iris presented in Alectryon, with (right) and without (left) our visualization library. (The screenshots on the right are static but the actual interface animates each step from the one before.) For example, the visualization renders the points-to predicate `p1` ↦ `(#f1, #b2)` as a record at location `p1` with two fields, `f1` and `b2`, with edges to the objects they point to. The displayed proof steps symbolically execute the code on the heap described in the pre-condition, which is hard to follow without heap diagrams. Explore interactively online.

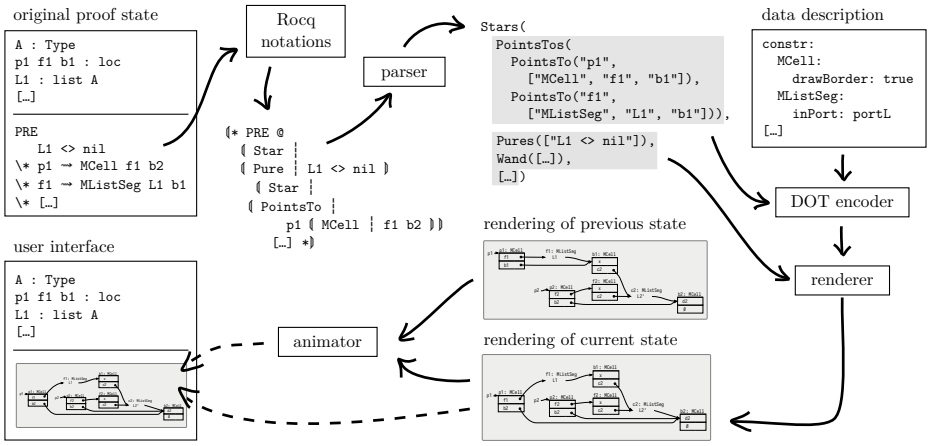

Fig. 3. Outline of the architecture of our tool. The proof state, translated with appropriate Rocq notations specific to the separation logic mechanization, is parsed to extract internal representations of the heap predicates (§2.1), aggregating points-to (i.e., representing data structures) and pure (i.e., relating only to logical models) predicates at each level of nesting. The points-to predicates are encoded into GraphViz's DOT format [15] (§2.2), controlled by the user-configured data description, specifying properties like whether to draw record borders and where to connect incoming edges. This DOT encoding is rendered to SVG and embedded in an HTML rendering of the rest of the proof state (§2.3). The rendering replaces the displayed proof state and is optionally shown as an animated transition from the previous step's output.

2 Architecture and Usage

Fig. 3 outlines the architecture of our library. At a high level, our library interacts with Rocq output at the level of surface syntax and produces its own output as HTML and SVG. To use our library, a user must provide a data description (described in §2.2) and write a set of notations to adapt the surface syntax of their separation logic mechanization to the generic, explicit syntax recognized by the parser (described in §2.1). (We already provide these for CFML and Iris.) Then, the user can immediately use the library with VsRocq or Alectryon, or integrate it with a tool of their choice using its JavaScript API.

2.1 Parsing the Proof State

Fig. 4 presents the parsing expression grammar (PEG) for our lightweight encoding of separation-logic proof states as Rocq notations, lightly simplified for clarity. Each *Goal* state is parsed as a collection of heap predicates (*HProp*s, or *RichHProps* if annotated with pre- or post-condition tags or a name) and heap *Value*s, ignoring everything not within tortoise shell brackets and double brackets (*Skip*), which we reserve for our exclusive use. The ignored parts are recorded and reproduced unchanged in the output. A heap predicate consists of a head

```
Goal ← (RichHProp / HProp / Value / Skip)*
RichHProp ← "(|*" (("PRE" / "POST") "@")? ("" Ident "" ":")?
    Skip* HProp? Skip* "*|)"
HProp ← "(|" Ident ("¦" (HProp / Value* / Skip*))* "|)"
Value ← "[|" Ident ("¦" (Value* / Skip*))* "|]"
Reserved ← "¦" / "(|*" / "*|)" / "(|" / "|)" / "[|" / "|]"
Skip ← !Reserved regex .
Ident ← !Reserved regex [a-zA-Z_\u0080-\uFFFF$]
    [a-zA-Z0-9_'\u0080-\uFFFF]*
```

Fig. 4. Parsing expression grammar (PEG) for our lightweight encoding of separation-logic proof states as Rocq notations, lightly simplified for clarity. *HProp* stands for "heap proposition". Symbols in a sans-serif italic font, like *HProp*, are non-terminals; quoted symbols, like "PRE", are terminals; and other symbols are part of the grammar notation. "/" is an *ordered* choice, "*" is a greedy quantifier, and "!" indicates a negative lookahead.

term followed by zero or more arguments, which can be other heap predicates, values, or arbitrary Gallina terms. *Value*s are parsed similarly.

We use the Peggy [29] parser generator to build a parser that constructs the top-level parse result as a JS array of either (i) nested JS objects, each corresponding to a *RichHProp*, *HProp*, or *Value*, with names and arguments stored as properties; or (ii) strings corresponding to *Skip*ped input.

Then, some head terms of *HProp*s have special interpretations:

- "Star", "Conj", and "Disj" indicate (binary) separating conjunction ($\star$), conjunction, and disjunction, respectively. As these operators are associative and most tactics are agnostic to their order of association, nested *HProp*s with these head terms are flattened into n-ary constructs.
- "PointsTo" and "Pure" indicate points-to and pure predicates, i.e. representing data structures or purely relating to logical models, respectively. Within each $\star$-conjunction, each kind is aggregated into a list.
- "Exists" and "Forall" indicate existentially and universally quantified variables, respectively. These are collapsed into a single scope, as if they were all at the start of the formula. We found that omitting the often-irrelevant scoping information makes the visualization more convenient. Bound names are resolved, and name conflicts are avoided with numerical suffixes. These names are highlighted in blue in the rendered diagrams (e.g. d0 in Fig. 1).
- "BigOp" indicates a "big" or iterated operator, such as iterated $\star$.

The grammar and the post-processing steps thus define a generic separation logic syntax with explicit nesting and associativity. We found that translating the surface syntax of CFML and Iris to this syntax using Rocq notations was straightforward, and we expect the same for other mechanizations.

2.2 Encoding into DOT

Before describing how our library encodes the parse result into DOT, we explain some relevant DOT concepts. A (directed) "graph" can contain "nodes" and directed "edges". Each of these objects has a string ID for internal use, an optional label for display, and other attributes controlling its layout and style. In particular, nodes can have "HTML-like labels" that can specify, among other possibilities, that the node should be drawn as a table, with some additional parameters like cell borders or spacing. Each table cell can serve as a "port" (with a specified "port name") that an edge can connect to or from. All nodes in our visualization except root pointers have HTML-like labels specifying tables.

Then, to encode a set of parsed, $\star$-conjoined points-to predicates into DOT, in broad terms, each points-to predicate is encoded as a node with a table label and an edge from the location to the node. The first row of the table (outside the border) indicates the location and name of the representation predicate, and the next rows are for its arguments. For example (Fig. 5), the predicate `p1` $\rightsquigarrow$ `MCell f1 b1` states that `p1` points to a record (an `MCell`) with two fields, with values `f1` and `b1`. `p1` does not exist elsewhere in the proof state so is shown as a label to the left. In contrast, when encoding `b1` $\rightsquigarrow$ `MCell x` $\emptyset$, `b1` does already exist as a field in the first record, which becomes the source of the edge.

The details of this encoding are controlled by the user-configured data description. Fig. 5 shows the data description (in YAML format) for our running example of Fig. 1. A data description is a map from representation predicate names to `ConstrConfig`s, where a `ConstrConfig` consists of:

- `drawBorder`, a Boolean, defaults to `false`: Should table cells for this predicate's arguments have borders?
- `args`, a map from argument indices (integers) to `ArgConfig`s, where an `ArgConfig` consists of:
  - `inTable`, a Boolean, defaults to `drawBorder`: Should this argument be shown in the table, or hidden?
  - `forceEdge`, a Boolean, defaults to `false`: Should this argument always produce an outgoing edge? If not, the argument will only produce one if it is known to be the location of an object elsewhere in the formula.
  - `inPort` and `outPort`, strings, default to `in$X` and `out$X`, where `X` is the argument index: Port names for incoming and outgoing edges.
- `inPort`, a string, defaults to `in$0`: The port incoming edges connect to.
- `label`, a string, defaults to predicate name: The predicate name to display.

The configuration file containing the data description can also contain DOT layout parameters that apply to all nodes, all edges, or the graph as a whole, trivially translated between YAML and DOT. However, modifying our default settings beyond simple stylistic changes can adversely affect the visualization, and should not be required for most ordinary uses. Finally, the file can also include configuration for value constructors: a `label` controlling how a *Value* is displayed, and a `uid` controlling its encoding-internal identifier.

```
MCell:
  drawBorder: true
MListSeg:
  inPort: 'portL'
  args:
    0: # e.g. b1
      forceEdge: true
      outPort: 'portL'
    1: # e.g. L1
      inTable: true
      inPort: 'portL'
```

```
  p1 ⇝ MCell f1 b1
⋆ f1 ⇝ MListSeg b1 L1
⋆ b1 ⇝ MCell x ∅
```

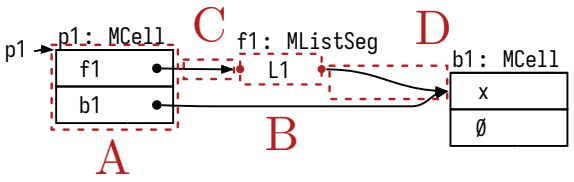

Fig. 5. (Explained in §2.2.) On the left, data description (in YAML format) for our running example of Fig. 1. On the right, an excerpt from a proof state and its illustration, annotated with red capital letters and dotted boxes. For `p1 ⇝ MCell f1 b1`, `drawBorder:true` means that record "A" has a border and both arguments of `MCell` are shown as record fields by default. In contrast, record "B" for `MListSeg` has no border, and only its argument 1 is shown due to `inTable:true`. Each field in a record can define its port name (e.g. `L1` defines `portL`); incoming edges to that port connect on the left, outgoing ones on the right. Then, other properties can refer to that port name, such as `inPort:'portL'` for `MListSeg` (resulting in edge "C") and `outPort:'portL'` for argument 0 of `MListSeg` (resulting in edge "D" to the object at location `b1`).

With the appropriate data description, the DOT encoding thus produced is *correct*, but needs further refinement to achieve more *canonicity* and *continuity* [4,11,33]: the visualization should be unaffected by changes to the separation logic formula that are irrelevant to the proof, and its evolution should be easy to follow over relevant changes. For canonicity, we use the following algorithm to determine the order in which nodes and edges are specified:

1. Sort edges alphabetically by their source node ID, source port, destination node ID, and destination port, in that order.
2. Partition the graph into weakly connected components via union-find.
3. Sort the components alphabetically by the node ID of their root. (Step #1 ensures that permuting a cycle in a component does not change which node is considered the root.)
4. Within each component,
   (a) Convert it to a DAG by condensation, i.e. replacing strongly connected components (SCCs) with single nodes.
   (b) Topologically sort the nodes of the DAG.
   (c) Decondense the DAG, i.e. restitute SCC nodes in alphabetical order.
5. Output first all nodes, then all edges.

Note that the above is specific to the graph layout algorithm we use, GraphViz's "dot", and does not ensure perfect canonicity for reasons discussed in §4. We empirically found the algorithm to produce more canonical results than the alternatives we tried, such as not sorting nodes at all, sorting them only alphabetically, or adding invisible edges to influence dot's crossing minimization decisions.

We experimentally also found that the procedure above improves continuity by avoiding some unnecessary layout changes, though some remain in the current version of the library.

Our main strategy for better continuity is to make layout changes easier to follow by animating them. To transition between "before" and "after" DOT graphs, the animator must identify which nodes and edges disappear from the "before" graph, which newly appear in the "after" graph, and which correspond between the two. We configure the animation library we use (d3-graphviz, described in the next subsection) to determine this correspondence based on the `id` attribute of DOT nodes and edges, which our DOT encoder controls: nodes are identified by the location name of the object they represent, and edges by the location name and port (if any) of their source. This means, for instance, that an edge changing its target node will be animated, but an edge changing its source node will not, which we have found useful in practice. For technical reasons discussed in §4, smooth animation with d3-graphviz relies on nodes with table labels keeping the same shape (number of rows and columns) in the "before" and "after" graphs, which our encoder also ensures.

2.3 Rendering, Animation, and Integration

Finally, our library renders the DOT encoding of the points-to predicates into SVG, embedded in a rendering of the rest of the proof state as an HTML fragment. The fragment is composed of nested HTML elements corresponding to predicate combinators (like $-\!\!*$) and textual labels (such as for pure predicates). It also serves as a target for user-driven events, like clicking to switch between the original textual proof state and the visualization or to copy the DOT encoding for debugging purposes. The DOT is converted to an SVG graph visualization using the "dot" layout algorithm of GraphViz [15], using a WebAssembly port of GraphViz from the hpcc-js-wasm project [18], wrapped as a D3 module [6] by the d3-graphviz library [21].

d3-graphviz also powers our animations. Our library animates the transition between "before" and "after" proof states by first rendering the "before" visualizations instantly, and then performing a D3 transition to the "after" visualizations, handled by d3-graphviz. The pre- and post-condition formulas in a proof state are visualized and animated independently. In our Alectryon integration, the "before" state is always the one just prior to the "after", and there are no animations for initial proof states, e.g. upon starting a new subgoal. In our VsRocq integration, the "before" state is whichever one the user was previously inspecting, even if not consecutive or in chronological order. We discuss this difference further in §4.

As our library compiles to JavaScript (from TypeScript) and produces HTML and SVG output, integration with JavaScript-based user interfaces for proof tools is straightforward. For our existing integrations with Alectryon and VsRocq, the compiled library is loaded into the HTML document that constitutes the interface, and run on the text of interface elements containing proof states to replace each with an interactive visualization.

3 Related Work

In this section, we discuss five categories of prior work and how they relate to ours: diagrammatic notation for separation logic, program state visualization tools, other formal methods visualizations, frameworks for building visualizations, and frameworks for integrating visualizations with proofs.

The diagrammatic notation used throughout this paper is far from original and rather mimics a notation ubiquitous in separation logic work, in research papers as well as teaching materials, early as well as modern. Examples can be found in [2,5,9,10,12,31,37,38,43]. [39] even proposed to formalize diagrammatic reasoning in separation logic as the basis for an automated theorem prover, but we did not find any further work in this direction.

Program state visualizations (PSVs) produced by educational, exploratory, and debugging tools like Aquascope [14], PythonTutor [16], and DDD [47] are similar to ours, but differ in three key aspects: the states they visualize are concrete, typically inaccessible to the user, and typically not animated. PSVs assume knowledge of concrete values in the program's execution, and hence do not directly apply to symbolic executions where some information may only be known as abstract heap predicates. Moreover, they are built from data a user does not typically access, collected by debuggers or instrumented runtimes, while our visualizations take the default user-facing output of a separation logic mechanization as input. Lastly, few PSVs to our knowledge animate transitions between states, and often only those targeted towards novices, like Jeliot [4], whereas continuity is a concern even for expert proof users. Theia [33] identifies continuity and animation as relevant goals for future PSV work, and Aquascope takes the alternative approach of visualizing multiple static states in the same display, as is also possible with our library's Alectryon integration. We point the reader towards [41] for a broader survey of PSVs.

Visualization for formal methods is a young but growing research area. Several works have focused on visualizations for the Alloy software modeling tool: [24] reports on interviews with expert users of a range of formal methods tools that informed the development of the Penlloy visualizer; [34] presents Cope-and-Drag, a diagramming language based on cognitive principles of diagramming and an analysis of an existing corpus of Alloy illustrations; and [13] describes advances in the built-in Alloy visualizer. In line with the findings of [11], all three highlight the need for low-cost, customizable diagramming systems that strive for canonicity and continuity of diagrammatic representations. VisB [44], a visualizer for the B modeling system as well as Event-B, Z, TLA+, and Alloy, is less automatic and requires a template SVG diagram from the user. The Symbolic Execution Debugger (SED) [17] visualizes symbolic execution trees along with object diagrams similar to ours describing the possible memory layouts of a symbolic state, and can be used to inspect correctness proofs created with KeY [1]. Lizard [3,42] supports similar exploration for programs written in the Viper intermediate verification language [25], building on Viper's symbolic-execution backend Silicon [40] and the Viper IDE infrastructure [23]. Finally, [36] is the

only prior work we found on animating proof states of a proof assistant (Lean), albeit only for plain text and noninteractively.

Beyond domain-specific visualization tools, among the long and rich history of frameworks for building general-purpose visualizations, the ones directly related to our work are GraphViz [15] and D3 [6]. GraphViz is a collection of graph layout algorithms that work with a common input format called DOT, which encodes the vertex-edge structure of a graph as well as parameters for its layout. D3 is a library for building interactive data visualizations, including many kinds of statistical charts as well as vertex-edge graph layouts. Our tool is built on GraphViz and D3 as described in §2. It could conceivably have used other libraries instead, such as graph-tool [30] or yFiles [46], because it does not rely on GraphViz- or D3-specific features, but we chose the latter out of convenience. An alternative approach to building visualizations is that taken by Penrose [45], focusing on diagrams for more mathematical domains like geometry, and for creative, iterative exploration rather than automatic proof illustration. As such, canonicity and continuity are not relevant objectives for Penrose diagrams.

Lastly, two projects are of note for enabling the use of existing visualizations in the context of computerized proofs: ProofWidgets [26] for Lean, and Alectryon [32] for literate proofs in Rocq and Lean. So far, we have focused on Rocq mechanizations of separation logic, and our tool integrates with Alectryon as described in §2.3. ProofWidgets integration could make for useful future work and would let the library work directly with Lean's internal representations of proof states instead of translating and reparsing surface syntax. Other possible integration targets include proof systems with native support for rich output (LaTeX, HTML) such as Isabelle [20] and PVS [35].

4 Discussion and Future Work

We conclude by outlining four limitations of our work and corresponding avenues for further research.

Our integration with VsRocq is more experimental than that with Alectryon, for two reasons. The first is that VsRocq exposes the proof state only at the user's current location, unlike Alectryon, which records the proof state at every location. When the state changes, it is animated from whichever state the user was previously inspecting, which may not always be desirable, instead of from the previous state in the proof as in Alectryon. The second shortcoming is that we had to fork VsRocq to integrate our library as we were unable to programmatically interface with it. VsRocq is a relatively young tool, and we hope that our experiences can inform ongoing discussions and development for making extensions like ours easier in the future.

While many separation logic frameworks have syntaxes conceptually similar to the generic one recognized by our parser, it may be difficult in exceptional cases to massage portions of their concrete syntax with Rocq notations into the expected format. It may be simpler to extend the grammar instead. We expect such cases to be rare.

Extending the scope of our visualizations is an open-ended direction for future work. Our treatment of magic-wand predicates could be improved by mimicking other hand-drawn diagrammatic notations, such as in [7], and by animating introduction and elimination steps. We believe it would also be valuable to support hierarchical rendering, such as for visualizing context and quantifier scoping for nested predicates.

Lastly, our visualizations do not achieve perfect canonicity and continuity: semantically irrelevant changes sometimes still affect the layout, and relevant ones sometimes produce larger layout changes than we suspect are necessary. This limitation is partly inherent to our approach. Like many graph layout algorithms, GraphViz's "dot" is designed for general-purpose use at scale, and optimizes properties like edge crossings and the aspect ratio with internal heuristics that are difficult to control (and potentially unnecessary) for small, structured applications like ours. It also does not support *incremental* layout, i.e. changing a previously laid-out graph while keeping layout changes minimal, or rigid specifications of alignment or ordering, which would benefit our visualizations. Meanwhile, d3-graphviz, the library we use for animation, detects and animates changes only in the SVG output of GraphViz, and cannot take into account higher-level, structural information about the graph or the heap objects it represents. For example, we might want parts of a diagram that were unchanged between two steps to stay in the same position. Future work could explore layout and animation algorithms purpose-built for separation logic proofs.

Acknowledgments. The authors would like to thank Alexander Müller for contributions to an early prototype of diagram animations. The authors are members of the EPFL-Inria REMPAR associate team. This work was funded in part by the Swiss National Science Foundation (SNSF) under grant no. 10003649. Parts of this material are based upon work of the second and third authors supported by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the AFRL and DARPA.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement. The tool described in this paper is available online in two formats: as a source code repository on GitHub at github.com/epfl-systemf/sepviz, and as a Docker image on Zenodo at doi:10.5281/zenodo.19844623. The repository reflects the current state of development of the tool, which may have diverged from the claims made in this paper; the `cav2026-artifact` tag on the repository marks the version included in the Docker image. The Docker image was submitted for artifact evaluation alongside this paper and additionally contains a compiled version of the tool, all software dependencies required to run and recompile it, and a detailed description of how the tool supports our claims in this paper.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – the KeY Book. No. 10001 in Lecture Notes in Computer Science, Springer, Cham, Switzerland (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Appel, A.W., Beringer, L., Cao, Q.: Verifiable C (Mar 2023), https://softwarefoundations.cis.upenn.edu/vc-current/index.html
3. Aurecchia, A.: Visual Debugging for Symbolic Execution. Master's thesis, ETH Zürich, Zürich, Switzerland (Sep 2018), https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/AlessioAurecchia_MA_report.pdf
4. Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P.A.: The Jeliot 2000 program animation system. Computers & Education **40**(1), 1–15 (Jan 2003). https://doi.org/10.1016/S0360-1315(02)00076-3
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11804192_6
6. Bostock, M., Ogievetsky, V., Heer, J.: D3: Data-driven documents. IEEE Transactions on Visualization & Computer Graphics **17**(12), 2301–2309 (Nov 2011). https://doi.org/10.1109/TVCG.2011.185, http://vis.stanford.edu/papers/d3
7. Cao, Q., Wang, S., Hobor, A., Appel, A.W.: Proof pearl: Magic wand as frame (Sep 2019). https://doi.org/10.48550/arXiv.1909.08789, http://arxiv.org/abs/1909.08789
8. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 418–430. ICFP 2011, Association for Computing Machinery, New York, NY, USA (Sep 2011). https://doi.org/10.1145/2034773.2034828
9. Charguéraud, A.: Separation logic (Jan 2016), https://www.charglueraud.org/teach/verif/seplogic.pdf
10. Charguéraud, A.: Separation logic foundations (Jan 2026), https://softwarefoundations.cis.upenn.edu/slf-current/index.html
11. Chiplunkar, S., Pit-Claudel, C.: Diagrammatic notations for interactive theorem proving. In: 4th International Workshop on Human Aspects of Types and Reasoning Assistants (HATRA '23). EPFL (2023). https://doi.org/10.5075/epfl-SYSTEMF-305144, https://infoscience.epfl.ch/record/305144
12. Costea, A., Zhu, A., Polikarpova, N., Sergey, I.: Concise read-only specifications for better synthesis of programs with pointers. In: Müller, P. (ed.) Programming Languages and Systems. Lecture Notes in Computer Science, vol. 12075, pp. 141–168. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-44914-8_6
13. Couto, R., Campos, J.C., Macedo, N., Cunha, A.: Improving the Visualization of Alloy Instances. In: Proceedings of the 4th Workshop on Formal Integrated Development Environment (F-IDE 2018). EPTCS 284, arXiv (Nov 2018). https://doi.org/10.4204/EPTCS.284.4, https://arxiv.org/abs/1811.10817v1
14. Crichton, W., Gray, G., Krishnamurthi, S.: A grounded conceptual model for ownership types in Rust. Proceedings of the ACM on Programming Languages **7**(OOPSLA2), 265:1224–265:1252 (Oct 2023). https://doi.org/10.1145/3622841

15. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software: Practice and Experience **30**(11), 1203–1233 (Aug 2000). https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N, https://graphviz.org/documentation/GN99.pdf
16. Guo, P.J.: Ten million users and ten years later: Python Tutor's design guidelines for building scalable and sustainable research software in academia. In: The 34th Annual ACM Symposium on User Interface Software and Technology. pp. 1235–1251. UIST 2021, Association for Computing Machinery, New York, NY, USA (Oct 2021). https://doi.org/10.1145/3472749.3474819
17. Hentschel, M., Bubel, R., Hähnle, R.: The Symbolic Execution Debugger (SED): A platform for interactive symbolic execution, debugging, verification and more. International Journal on Software Tools for Technology Transfer **21**, 485–513 (Mar 2018). https://doi.org/10.1007/s10009-018-0490-9
18. HPCC Systems: @hpcc-js/wasm. HPCC Systems (Aug 2024), https://github.com/hpcc-systems/hpcc-js-wasm
19. INRIA and contributors: VsRocq (Nov 2025), https://github.com/rocq-prover/vsrocq
20. Isabelle contributors: Isabelle (Jan 2026), https://isabelle.in.tum.de/index.html
21. Jacobsson, M.: d3-graphviz (Aug 2024), https://github.com/magjac/d3-graphviz
22. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 637–650. POPL 2015, Association for Computing Machinery, New York, NY, USA (Jan 2015). https://doi.org/10.1145/2676726.2676980
23. Kälin, R.: Advanced Features for an Integrated Verification Environment. Master's thesis, ETH Zürich, Zürich, Switzerland (2016). https://doi.org/10.3929/ETHZ-A-010807288, http://hdl.handle.net/20.500.11850/156118
24. Liang, Y., Palliyil, A., Kang, E., Sunshine, J.: Towards Better Formal Methods Visualizations. In: Proceedings of the 15th Workshop on Programming Languages and Human-Computer Interaction. PLATEAU 2025, Carnegie Mellon University, Pittsburgh, PA, USA (Feb 2025). https://doi.org/10.1184/R1/29086949.v1
25. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI 2016). pp. 41–62. No. 9583 in Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Dec 2015). https://doi.org/10.1007/978-3-662-49122-5_2
26. Nawrocki, W., Ayers, E.W., Ebner, G.: An extensible user interface for Lean 4. In: 14th International Conference on Interactive Theorem Proving (ITP 2023). pp. 24:1–24:20. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ITP.2023.24
27. O'Hearn, P.W.: Separation logic. Communications of the ACM **62**(2), 86–95 (Jan 2019). https://doi.org/10.1145/3211968
28. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
29. Peggy contributors: Peggy – parser generator for JavaScript (Aug 2025), https://peggyjs.org/

30. Peixoto, T.P.: Graph-tool (Aug 2025), https://graph-tool.skewed.de/
31. Piskac, R., Wies, T., Zufferey, D.: GRASShopper: Complete heap verification with mixed specifications. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_9, http://link.springer.com/10.1007/978-3-642-54862-8_9
32. Pit-Claudel, C.: Untangling mechanized proofs. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. pp. 155–174. SLE 2020, ACM, Virtual (Nov 2020). https://doi.org/10.1145/3426425.3426940
33. Pollock, J., Roesch, J., Woos, D., Tatlock, Z.: Theia: Automatically generating correct program state visualizations. In: Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E. pp. 46–56. SPLASH-E 2019, Association for Computing Machinery, New York, NY, USA (Oct 2019). https://doi.org/10.1145/3358711.3361625
34. Prasad, S., Greenman, B., Nelson, T., Krishnamurthi, S.: Lightweight Diagramming for Lightweight Formal Methods: A Grounded Language Design. In: Aldrich, J., Silva, A. (eds.) 39th European Conference on Object-Oriented Programming (ECOOP 2025). Leibniz International Proceedings in Informatics (LIPIcs), vol. 333, pp. 26:1–26:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2025). https://doi.org/10.4230/LIPIcs.ECOOP.2025.26
35. PVS contributors: PVS (Nov 2020), https://pvs.csl.sri.com/
36. Renshaw, D.: animate-lean-proofs (Dec 2025), https://github.com/dwrensha/animate-lean-proofs
37. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. IEEE (Jul 2002). https://doi.org/10.1109/LICS.2002.1029817, https://ieeexplore.ieee.org/document/1029817
38. Reynolds, J.C.: A short course in separation logic (Mar 2006), https://cs.ioc.ee/yik/schools/win2006/reynolds/estslides.pdf
39. Ridsdale, M., Jamnik, M., Benton, N., Berdine, J.: Diagrammatic reasoning in separation logic. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrammatic Representation and Inference. Lecture Notes in Computer Science, vol. 5223, pp. 408–411. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87730-1_50
40. Schwerhoff, M.H.: Advancing Automated, Permission-Based Program Verification Using Symbolic Execution. Ph.D. thesis, ETH Zürich, Zürich, Switzerland (2016). https://doi.org/10.3929/ETHZ-A-010835519, http://hdl.handle.net/20.500.11850/127711
41. Sorva, J., Karavirta, V., Malmi, L.: A review of generic program visualization systems for introductory programming education. ACM Transactions on Computing Education **13**(4), 15:1–15:64 (Nov 2013). https://doi.org/10.1145/2490822
42. Ter-Gabrielyan, A.: Lizard: The visual debugger for Viper (Aug 2021), https://github.com/viperproject/lizard
43. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. ACM Transactions on Programming Languages and Systems **30**(4), 24:1–24:58 (Aug 2008). https://doi.org/10.1145/1377492.1377499
44. Werth, M., Leuschel, M.: VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-

Based Methods. pp. 260–265. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_21
45. Ye, K., Ni, W., Krieger, M., Ma'ayan, D., Wise, J., Aldrich, J., Sunshine, J., Crane, K.: Penrose: From mathematical notation to beautiful diagrams. ACM Transactions on Graphics **39**(4) (Aug 2020). https://doi.org/10.1145/3386569.3392375
46. yWorks: yFiles for HTML. yWorks (Feb 2025), https://www.yworks.com/yfiles-overview
47. Zeller, A., Lütkehaus, D.: DDD—a free graphical front-end for UNIX debuggers. ACM SIGPLAN Notices **31**(1), 22–27 (Jan 1996). https://doi.org/10.1145/249094.249108