A Formalization of Multiplicative Proof-Nets in Rocq

Rémi Di Guardia* Olivier Laurent†

TLLA 2025

Abstract

We present an implementation of proof-nets for unit-free multiplicative linear logic in the proof assistant Rocq. It contains a definition of proof-structures and proof-nets, a proof of the sequentialization theorem as well as a definition of cut-elimination.

1 Introduction

Proof assistants are used to formalize mathematical reasonings and thus to certify formal proofs. A line of work in formalization consists in implementing a logic then proving some of its properties – *e.g.* admissibility of the *cut*-rule. This gives more confidence in those proofs where usually many cases are to consider, with sometimes tedious details and technicalities. There are several implementations of (sub-systems of) *linear logic* [Gir87] in different proof assistants: *e.g.* [1; 2; 3; 4; 5; 6] in Rocq, [7] in Abella and [8; 9] in Isabelle. However, all these libraries consider only the sequent calculus syntax and never the *proof-net* syntax, despite the later being a major contribution from linear logic [Gir96]. Contrary to the usual representation of proofs as derivation trees in sequent calculus, proof-nets represent proofs as general graphs respecting some *correctness criterion* [DR89]. They quotient derivations of sequent calculus up to rule commutations, thus results like cut elimination are easier to prove in this formalism. Even if there is not much doubt that the key results (sequentialization, cut-elimination, etc.) about multiplicative proof-nets hold, for they have been proved in many different ways, this is not the case for all proof-nets. For instance, various definitions of unit-free multiplicative-additive proof-nets [HG05; HH16] have not only complex sequentialization proofs, but also rarely more than one such proof. Still, before formalizing these complex objects, starting with the simpler multiplicative proof-nets is a step where most difficulties arising from the formalization are present.

The current absence of implementations can be explained by the very nature of proof-nets: while a proof of sequent calculus is an inductive object, easily defined in most proof assistants, a proof-net is a graph, which is harder to define and manipulate formally. Indeed, many reasonings involve "geometric" or "graphical" arguments, with drawings to explain the proof, as well as silent "evident" arguments – for instance that some concatenation of simple paths is simple without giving explicitly all hypotheses leading to this conclusion. Another explanation is that transformations of graphs – *e.g.* adding or removing vertices and edges – are done very frequently but are complex formally: vertices after the transformation are not those before and, whereas on paper one usually and implicitly considers graphs up to isomorphisms, on computer one has to explicitly provide these isomorphisms (removing a vertex then adding it back is not exactly the original graph).

We formalized proof-nets for unit-free multiplicative linear logic in the *Rocq* prover [Roc], so as to expand the *Yalla* [Lau17] library developed by the second author. It uses *Graph Theory* [DP18], a recent graph library built upon the *Mathematical Components* library [MC] with its *SSReflect* proof language. This library defines graphs and many needed operations on them: adding and removing vertices and edges, isomorphisms of graphs, sub-graphs, etc. More precisely, we implemented:

- a definition of proof-nets;

*Université Paris Cité, CNRS, IRIF, Paris, France, diguardia@irif.fr
†CNRS, ENS de Lyon, UCBL1, LIP, UMR 5668, 69342, Lyon cedex 07, France, olivier.laurent@ens-lyon.fr

- a definition of the desequentialization function from derivations of sequent calculus to graphs (with a proof the obtained graph is a proof-net);
- a proof of the sequentialization theorem: given a proof-net, there exists a proof desequentializing to it (more precisely to a graph isomorphic to it);
- a definition of cut-elimination in a proof-net (with a proof it preserves being a proof-net and that it terminates).

In this paper are given details about some of these results, how they have been formalized and why – as a large part of the work was to discover which formalization choices were best suited to the problem. The source code is public and can be found on Software Heritage[1] (the current version is also available on a git[2]). More details on this formalization are given in the first author's thesis [Di24, Chapter 5].

Outline We start by giving the chosen formal definition of proof-net, with some discussion about *rejected* solutions (Section 2). Then are explained some difficulties encountered during the formalization (Section 3).

2 Formalization of proof-nets

We present here the cornerstone of our formalization, where choices mattered the most: the formal definition of proof-nets, and all depending strata (proof-structures, etc.). Choosing these definitions was the most complex part of the formalization because they directly influence all other choices. Many different, but equivalent, definitions of unit-free multiplicative proof-nets exist in the literature, notably as multigraphs [FR94] and as hypergraphs [GM01]. We choose here one of the most usual presentation, seemingly simple enough to be formalized: a proof-net is a directed multigraph made of *ax*-, $\parr$-, $\otimes$- and *cut*-vertices, with edges between them. Even once this definition chosen, there are still subtle variations possible, among which:

- does one put conclusion vertices, or have pending edges, and thus a partial graph?
- does one put *ax*-vertices or just axiom links?
- which correctness criterion to distinguish proof-nets among proof-structures?
- $\parr$- and $\otimes$-vertices have ordered premises (*i.e.* syntactically $A \parr B \neq B \parr A$), how to take it into account?

Our goal was to choose a formalization making the definitions and proofs we are interested in easier to write (sequentialization and cut-elimination). Furthermore, for taking advantage of the definition of derivations already implemented in the Yalla library, some notions of formulas and sequents were already fixed. In particular, the choices made in the current version of Yalla is that a sequent is a *list* Γ of formulas and there is an *explicit* exchange rule $\dfrac{\vdash \Gamma}{\vdash \sigma(\Gamma)}\ (ex)$ (with σ a permutation). We choose to have conclusion vertices, so as to consider total graphs, as well as *ax*-vertices to keep only directed edges. The chosen correctness criterion is the Danos-Regnier one [DR89], which is one of the most commonly used criterion and for which there is a simple proof of sequentialization [Di+25].

Concretely, two main difficulties arise when defining proof-nets. On one hand, we have graphs with some supplementary data: identification of left and right premises of $\parr$- and $\otimes$-vertices, ordering to recover a conclusion sequent. On the other hand, there are all the constraints on proof-structures: in- and out-degrees of vertices, relations respected by the formulas labeling edges. It is natural to separate these two parts, with first a definition of graphs with some more data, then asking properties on these objects, yielding proof-structures and then proof-nets. Since the properties depend on the concrete definitions of the supplementary data, it is no surprise the first part was the most difficult to formalize. Here, we give a definition of proof-nets along a description of some non-trivial implementation choices.

[1] https://archive.softwareheritage.org/browse/directory/53def4330cb28b417c83405f68a67a9735a95d1d/
[2] https://github.com/RemiDiG/proofnet_mll

2.1 Graph with additional data

The library Graph Theory defines (finite) graphs as follows:

```
Record graph (Lv Le : Type) : Type :=
  Graph {
      vertex:> finType;
      edge: finType;
      endpoint: bool → edge → vertex;
      vlabel: vertex → Lv;
      elabel: edge → Le }.
Notation source := (endpoint false).
Notation target := (endpoint true).
```

In plain words, a graph is the data of a finite set *vertex* of vertices, a finite set *edge* of edges, a function *endpoint* giving the source and target of edges, and labeling functions *vlabel* and *elabel* on vertices and edges respectively. Hence our definition of the underlying graphs for proof-nets:

Definition 1. A MLL **graph** is a finite total directed multigraph [JU76] equipped with:

- a labeling function $\mathcal{R}$ from vertices to $\{ax; cut; \otimes; \parr; c\}$ (with c for conclusion vertices);
- a labeling function $\mathcal{F}$ from edges to formulas;
- a labeling function $\mathcal{L}$ from edges to Booleans to identify left premises.

Vertices are named according to their label: *ax*-**vertices**, *cut*-**vertices**, $\otimes$-**vertices**, $\parr$-**vertices** and *c*-**vertices**. Given a vertex v, edges with target v are the **premises** of v. To identify which edges are left and right premises of $\otimes$- and $\parr$-vertices, the standard trick on paper is to always write the left premise on the left of the vertex and the right premise on its right. Obviously, this is not formal: here the $\mathcal{L}$ function is used instead. Our implementation of MLL graphs follows:[3]

```
Notation base_graph := graph (flat rule) (flat (formula × bool)).
Definition flabel {G : base_graph} (e : edge G) : formula := fst (elabel e).
Definition llabel {G : base_graph} (e : edge G) : bool := snd (elabel e).
```

This means a *base_graph* is a graph with vertices labeled by *rule* (a type whose elements are $\{ax; cut; \parr; \otimes; c\}$), and edges by *formula* (a type taken from Yalla) together with a *bool* to identify left and right edges. This definition is easy to write and manipulate, as it corresponds to an instance of the graphs from Graph Theory.

Remark we still lack data at this point: how to recover the conclusion sequent associated to such a graph? As it is a list, we have for instance that $\vdash A, B$ and $\vdash B, A$ are different. Thus, we order *c*-vertices.

Definition 2. A **pre-proof-structure** is a MLL graph along with a list of some of its edges. The **sequent** associated to a pre-proof-structure is the list of formulas obtained by applying $\mathcal{F}$ on this list.

In Rocq this corresponds to the following:

```
Record graph_data : Type :=
  Graph_data {
    graph_of :> base_graph;
    order : seq (edge graph_of)}.
```

Later, for a proof-structure, the list of edges *order* will be asked to contain exactly once each edge of target a *c*-vertex. This list indeed allows to define the sequent associated to such a graph:

```
Definition sequent (G : graph_data) : seq formula := [seq flabel e | e ← order G].
```

We choose to separate the list *order* from the other data on the graph, because these are quite disjoint concepts, and we rarely need the sequent of a graph – *e.g.* it is useless for the correctness criterion.

Remark these definitions are quite far from the usual ones in the literature. In particular, we have useless information: the $\mathcal{L}$ function is defined for all edges and not just for premises of $\parr$- and $\otimes$-vertices. The reason is that it is usually easier to have one's data on a simple object, rather than a more accurate but partial

[3] The *flat* function is the identity, used by the graph library for technicalities about isomorphisms. It can be safely ignored.

data on complex structures. Typically, when modifying a graph one also has to adapt proofs that it is a proof-structure, on which will depend the definition of a left edge in this case. This leads to convoluted proofs and definitions of left edges, whereas having left from the very beginning, with possibly some nonsensical value for, say, the premise of a c-vertex, makes defining objects easier, and leads to simpler proofs. For instance, formalizing near what happens on paper would yield the following:

```
Record graph_data : Type :=
  Graph_data {
      graph_of :> graph rule formula;
      order : [finType of { v : graph_of | vlabel v == c}] →
        'I_#|[finType of { v : graph_of | vlabel v == c}]|;
      order_inj : injective order;
      direction : bool → [finType of { v : graph_of | vlabel v == ⊗ || vlabel v == ⅋ }] → edge graph_of }.
Notation left := (direction false).
Notation right := (direction true).
```

We would have functions *left* and *right*, generalized as a unique function *direction*, defined on ⅋- and ⊗-vertices, and an injective (thus bijective) *order* function associating to each c-vertex its order in the sequent ('*I_n* is the set of natural numbers lesser than *n*). Alas, this definition is terrible to manipulate. Using cardinals makes it mandatory to compute these cardinals as soon as one defines such a graph, or worse when defining a graph from another. Furthermore, dependent types complexify the uses of this definition, which is a shame as it is our basic block and therefore is present everywhere! As an example, to do a basic operation such as the disjoint union of two graphs, one needs to use proofs that a vertex labeled ⊗ before keeps this label after, and so on, just to define this union. Hence, we choose another implementation to keep the following definitions and proofs easy to handle. The chosen solution is nonetheless not without fault, for we need to fix a value for *llabel* on irrelevant edges, that we choose arbitrary to be *true*.

2.2 Proof-nets

From the previous graphs, proof-structures can be easily defined by asking for geometrical constraints on in- and out-degrees along (local) considerations on labels of edges. We also have to relate the list of edges to the c-vertices. There is nothing surprising nor difficult here, leading to implementations such as:

```
Definition proper_tens_parr (G : base_graph) :=
  ∀ (b : bool) (v : G), vlabel v = (if b then ⅋ else ⊗) → ∃ el er ec,
  el \in edges_at_in v ∧ llabel el ∧ er \in edges_at_in v ∧ ¬llabel er ∧
  ec \in edges_at_out v ∧ flabel ec = (if b then parr else tens) (flabel el) (flabel er).
```

This property means that any ⅋-vertex (resp. ⊗-vertex) has two in-coming edges el and er and one out-going edge ec such that $\mathcal{L}(el)$ is true, $\mathcal{L}(er)$ is false and $\mathcal{F}(ec) = \mathcal{F}(el) \parr \mathcal{F}(er)$ (resp. $\mathcal{F}(ec) = \mathcal{F}(el) \otimes \mathcal{F}(er)$).

We can finally define proof-nets. A path is **simple** if it is without repetition of edges.

Definition 3.

- A **switching path** (resp. **cycle**) is a simple undirected path (resp. cycle) not containing both premises of a ⅋-vertex. A **left-switching path** is a simple undirected path not containing a premise of a ⅋-vertex whose image by $\mathcal{L}$ is false.
- A **proof-net** is a proof-structure which is **correct**, meaning we have both:

  acyclicity: there is no non-empty switching cycle;

  connectedness: the graph is not empty, and between any two vertices there is a left-switching path.

One can check this definition is equivalent to the standard one using correctness graphs [DR89], because all acyclic correctness graphs have the same number of connected components. Considering these switching and left-switching paths allows to stay in the underlying graph of the proof-net, preventing from proving results such as "if a correctness graph is acyclic, and a vertex in the proof-net is removed, then the resulting correctness graph is acyclic". These results, not proved on paper as trivial, are tedious formally because one needs to transport paths from one graph to another, which is not hard but is a long and boring chore.

Considering only one graph prevents the need for such lemmas, at the lighter cost of considering paths with restrictions on their edges – restriction we need in some shape anyway, for we have to forbid repeating edges.

About the implementation in Rocq, undirected paths were not in Graph Theory, there were only directed paths. We defined them as follows, by mimicking the definition of directed paths, and prove some of their usual properties (concatenation, sub-paths, etc.).

```
Definition upath {Lv Le : Type} {G : graph Lv Le} := seq ((edge G) × bool).
Notation usource e := (endpoint (~~e.2) e.1).
Notation utarget e := (endpoint e.2 e.1).
Fixpoint uwalk (x y : G) (p : upath) :=
  if p is e :: p' then (usource e == x) && uwalk (utarget e) y p' else x == y.
```

Having a notion generalizing both switching and left-switching paths allows to factorize most of the work about correctness. To this end, we need to associate edges (premises of the same ⅋-vertex for switching paths) and to forbid edges (right premises of ⅋-vertices for left-switching paths). Thus are defined f-paths, given as parameter a function f from edges to some option type, which can be seen as an edge-coloring with a special color *None*. A path is an **f-path** if its image by f has no duplicate (*i.e.* for two edges a and b inside, $f(a) \neq f(b)$) and does not contain *None*. Switching paths are f-paths for f the identity, except for premises of ⅋-vertices which are sent to the same element – *e.g.* the ⅋-vertex they point to. Left-switching paths are f-paths for f the identity, except for non-left premises of ⅋-vertices which are sent to *None*. Remark in particular that with f the identity, an f-path is exactly a simple path. This yields in Rocq:

```
Definition supath {Lv Le : Type} {I : eqType} {G : graph Lv Le} (f : edge G → option I) (s t : G)
  (p : upath) := (uwalk s t p) && uniq [seq f e.1 | e ← p] && (None \notin [seq f e.1 | e ← p]).
Definition switching {G : base_graph} : edge G → option ((edge G) + G) :=
  fun e ⇒ Some (if vlabel (target e) == ⅋ then inr (target e) else inl e).
Definition switching_left {G : base_graph} : edge G → option (edge G) :=
  fun e ⇒ if (vlabel (target e) == ⅋) && (~~llabel e) then None else Some e.
```

One then defines acyclicity and connectedness for these paths, finally yielding a definition of proof-nets.

3 Limits of the implementation

Once proof-nets are defined, other definitions and proofs follow more or less straightly. We explain here one of the main difficulties faced in this process: isomorphisms in sequentialization (see [Di24, Chapter 5] for two other difficulties: manipulating graphs explicitly and the computation time of Rocq itself). When proving the sequentialization theorem, the hard part on paper is finding a splitting vertex. Then, proving that such a vertex can be removed to get one or two proof-nets is trivial enough to be left implicit, and is often not even mentioned. On computer, it is the converse: proving there is a splitting vertex is not much harder than on paper, whereas the implicit part translates on computer into a most tedious task!

As an example, consider the search for a splitting ⊗-vertex v. It is easy to obtain a terminal ⊗-vertex not belonging to any cycle. This is enough for sequentialization, as removing it gives two connected components, whence two proof-nets. However, formally one has to define those connected components, for instance by looking at all vertices having a path from them to the source of the left premise of v. Then one uses this complex definition to prove that adding a c-vertex in place of v in this graph indeed yields a proof-net. Furthermore, once this tedious work done on both graphs, one still has to give an isomorphism between the original graph and the two proof-nets to which one adds v back!

4 Conclusion & Perspectives

We implemented in Rocq a definition of unit-free multiplicative proof-nets with two main results of this theory: the sequentialization theorem and the normalization of cut-elimination. Many concepts and results could be added to this formalization: axiom-expansion, confluence, links with sequent calculus regarding cut-elimination and rule commutations, tacking into account the *mix*-rules, etc. Nonetheless, this project shows that formalizing proof-nets can be done in a not so tedious fashion. Extensions to exponential or additive proof-nets seem doable, as the main difficulties should be about manipulating graphs, meaning the same as for multiplicative proof-nets.

References

[Di+25] Rémi Di Guardia, Olivier Laurent, Lorenzo Tortora de Falco, and Lionel Vaux Auclair. "Yeo's Theorem for Locally Colored Graphs: the Path to Sequentialization in Linear Logic". In: *International Conference on Formal Structures for Computation and Deduction (FSCD)*. Ed. by Maribel Fernandez. Leibniz International Proceedings in Informatics (LIPIcs). To appear. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, July 2025.

[Di24] Rémi Di Guardia. "Identity of Proofs and Formulas using Proof-Nets in Multiplicative-Additive Linear Logic". Thèse de Doctorat. Ecole normale supérieure de lyon - ENS LYON, Sept. 2024. URL: https://theses.hal.science/tel-04830060.

[DP18] Christian Doczkal and Damien Pous. *Graph Theory*. Coq library. 2018. URL: https://github.com/coq-community/graph-theory.

[DR89] Vincent Danos and Laurent Regnier. "The structure of multiplicatives". In: *Archive for Mathematical Logic* 28 (1989), pp. 181–203. DOI: 10.1007/BF01622878.

[FR94] Arnaud Fleury and Christian Retoré. "The Mix Rule". In: *Mathematical Structures in Computer Science* 4.2 (1994), pp. 273–285. DOI: 10.1017/S0960129500000451.

[Gir87] Jean-Yves Girard. "Linear logic". In: *Theoretical Computer Science* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4.

[Gir96] Jean-Yves Girard. "Proof-nets: the parallel syntax for proof-theory". In: *Logic and Algebra*. Ed. by Aldo Ursini and Paolo Agliano. Vol. 180. Lecture Notes In Pure and Applied Mathematics. New York: Marcel Dekker, 1996, pp. 97–124. DOI: 10.1201/9780203748671-4.

[GM01] Stefano Guerrini and Andrea Masini. "Parsing MELL proof nets". In: *Theoretical Computer Science* 254.1–2 (2001), pp. 317–335. DOI: 10.1016/S0304-3975(99)00299-6.

[HG05] Dominic Hughes and Rob van Glabbeek. "Proof Nets for Unit-free Multiplicative-Additive Linear Logic". In: *ACM Transactions on Computational Logic* 6.4 (2005), pp. 784–842. DOI: 10.1145/1094622.1094629.

[HH16] Willem Heijltjes and Dominic Hughes. "Conflict nets: Efficient locally canonical MALL proof nets". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM Press, July 2016, pp. 437–446. DOI: 10.1145/2933575.2934559.

[JU76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. 1976.

[Lau17] Olivier Laurent. *Yalla: Yet Another deep embedding of Linear Logic in Coq*. Coq library. July 2017. URL: https://perso.ens-lyon.fr/olivier.laurent/yalla/.

[MC] The MC Team. *Mathematical Components*. Coq library. URL: https://math-comp.github.io/.

[Roc] The Rocq Development Team. *The Rocq Prover*. DOI: 10.5281/zenodo.15149629. URL: https://rocq-prover.org/.

Linear Logic Formalizations in Proof Assistants

[1] https://github.com/olaure01/yalla

[2] https://github.com/ComputerAidedLL/PowerWebster_ILL

[3] https://github.com/meta-logic/coq-ll

[4] https://github.com/Matafou/ill_narratives

[5] https://github.com/ppedrot/ll-coq

[6] https://eprints.soton.ac.uk/261814/

[7] https://github.com/meta-logic/abella-reasoning

[8] https://www.cl.cam.ac.uk/~lp15/papers/Workshop/papers/kalvala-linear.pdf

[9] https://link.springer.com/chapter/10.1007/3-540-59338-1_41