# Introducing Proof Tree Automata and Proof Tree Graphs

Valentin D. Richard
LORIA, UMR 7503
Université de Lorraine, CNRS, Inria
54000 Nancy, France
valentin.richard@loria.fr

**Introduction** Research in structural proof theory [7] may lead to considering large calculi, containing several dozens of rules (e.g. 68 rules found in [5]). Keeping track of all possible combinations of theses rules is an issue. This problem is particularly critical at the design phase, when trying to come up with a calculus which meets some desiderata.

When trying to design a calculus, researchers often do not just want to test *whether* it has the expected specification, but to know *why* and *how* it does or does not. Their requirements often revolve around intuitions about connectives and rules, e.g. "What happens if we add or remove this rule?".

The combinatorics of rules also brings a challenge at proof phase, when trying to demonstrate properties about a calculus. Many theorems on calculi still make use of case disjunction. Such a strategy becomes difficult and fastidious as the size of the system increases. There is a desire to get a larger picture of calculi, to get new insights about them.

Approaches based on graphical languages, like proof nets or string diagrams, turned out to be of great use to give visual intuitions. Nevertheless, they often focus on a single derivation and divert from the very structure of derivation trees.

**Proposal** The contribution of this article is twofold:

1. Introducing a novel graphical representation of a calculus aiming at bringing better intuition about the interconnection of rules and sequents
2. Providing a new perspective of proof search through tree automata theory

**Proof tree graphs** The graphical representation we introduce is called Proof Tree Graph (PTG). A PTG can represent a calculus, or more generally any term deduction system.[1]

- Vertices are sets of terms (e.g. sequents, if we work with a sequent calculus)
- Edges are rules

In Fig. 1, we display a PTG for implicational sequent calculus ImpL ((1) in appendix), and in Fig. 2 a PTG for the sequent calculus of the multiplicative fragment of linear logic (MLL, see [3]).[2] Edge $\Delta, \varphi \vdash \psi \xrightarrow{\to \text{I.}} \Delta \vdash \varphi \to \psi$ represents rule ($\to$ I.), from the hypothesis to the conclusion. The axiom is

[1] Technically, a PTG is a directed hypergraph [1, chap. 6] with additional dashed edges.

[2] In both calculi considered, we take commas to be multiset separators.

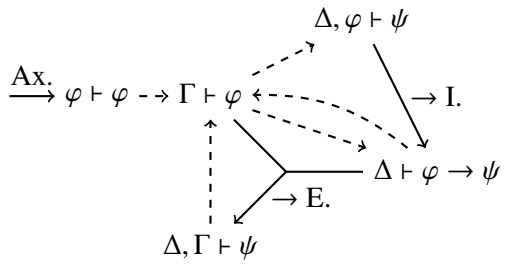


Figure 1: Proof Tree Graph for implicational sequent calculus.

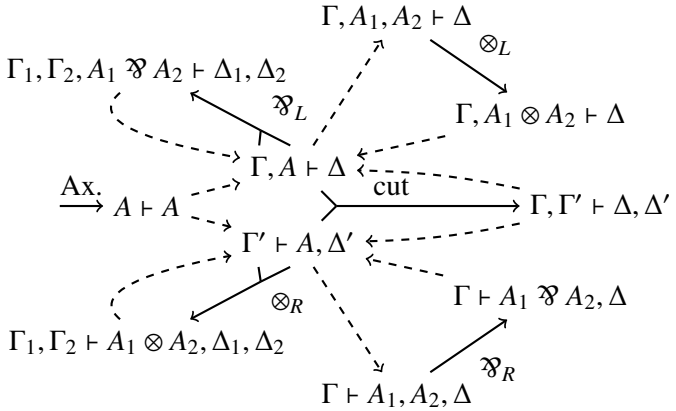


Figure 2: Proof Tree Graph for MLL.

represented by an edge with no source as it has no hypothesis. By the same idea, rule $\to$ E. has two source vertices as it has two hypotheses.

A dashed edge $u \dashrightarrow v$ mean that we can pass from vertex $u$ to $v$ without applying a rule. For example, $\varphi \vdash \varphi \dashrightarrow \Gamma \vdash \varphi$ means that both vertices share a common instance sequent, e.g. $p \vdash p$ if $p$ is an atomic formula. Note here that a sequent actually stands for a set of instances. They are thus taken up to meta-variable renaming.

The goal of a PTG is to give visual intuitions about the relationships between rules by linking the hypotheses and the conclusions of these rules. This way, it appears clearly how certain rules can follow other rules. Thus, a PTG illustrates the whole system, and not a particular derivation. For example, the antecedent – succedent symmetry of linear logic is visible through the horizontal axis symmetry if Fig. 2.

One recipe to create a PTG out of any calculus $\mathcal{K}$ is the following:

1. As vertex, take any hypothesis or conclusion of a rule of $\mathcal{K}$
2. Create an $n$-ary edge for every $n$-ary rule on the corresponding vertices
3. Add a dashed edge $u \dashrightarrow v$ for every vertices $u$ and $v$ which share an instance (i.e. $u \cap v \neq \emptyset$)

Proof nets [4] and string diagrams [8], widespread graphical languages, differ from PTGs on the kind of object represented. They can only represent (sets of equivalent) derivations, whereas PTGs allows us to represent the whole calculus. Therefore, PTGs give an overarching image of the rules. Some first idea a such a diagram of rules can be found in [5, Fig. 2].

**Proof Tree Automata** If all rules of a calculus are unary, a PTG on that calculus looks like the graphical representation of a non-deterministic finite automaton: vertices are states and edges are transitions. In this setting, axiom targets make initial states, dashed edges are $\varepsilon$-transitions and all states are accepting.

We build on that analogy to retro-engineer a new kind of tree automata called Proof Tree Automata (PTA), which graphical representations are PTGs. A PTA $\mathcal{A}$ on a calculus $\mathcal{K}$ is a tree automaton ([2]) with additional material. Its language is the derivation language of $\mathcal{K}$. A forward proof-search in $\mathcal{K}$ corresponds to a bottom-up run in $\mathcal{A}$.

The additional material is a pair of relations called control relations. Their goal is to ensure that, while parsing a proof tree, hypothesis terms and conclusion terms are correctly related.

Using automata and graphs is an open door to topological methods for term deduction. One goal of PTA and PTGs is to provide a tool with which we can translate properties expressed on sets of derivation trees into properties expressed on automaton runs or graph walks.

**Additional results and open questions** An interesting point is the comparison of a PTA $\mathcal{A}$ on a calculus $\mathcal{K}$ and its tree automaton counterpart $F(\mathcal{A})$, i.e. with control relations removed. The language of $F(A)$ is wider than the language of $\mathcal{A}$ because it contains derivations which are not correct wrt. $\mathcal{K}$.

One can build a function[3] $U$ from $\mathcal{K}$ to $F(A)$, mapping sets of terms to states and derivations to runs. Function $U$ has the following property: a derivations of $F(A)$ is correct iff it belong to the image of $U$. Thus, a PTA appears as tree automaton parameterized by a calculus.

As a novel tool, many questions arise about PTA and PTGs. Particularly, we deem investigations about relations on PTA to be relevant. When can we say that a PTA is finer than another one? Could we design a criterion for PTA equivalence (i.e. having the same language). It would also be useful to find graph rewriting techniques to compute these problems on PTGs.

# References


[1] Alain Bretto. *Hypergraph Theory: An Introduction*. Springer Science & Business Media, April 2013.

[2] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008.

[3] Roberto Di Cosmo and Dale Miller. Linear logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] Giuseppe Greco, Valentin D. Richard, Michael Moortgat, and Apostolos Tzimoulis. Lambek-Grishin calculus: Focusing, display and full polarization. *Logic and Structure in Computer Science and Beyond*, 2021.

[6] Paul-André Melliès and Noam Zeilberger. Functors are Type Refinement Systems. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, January 2015.

[7] Sara Negri and Jan von Plato. *Structural Proof Theory*. Cambridge University Press, Cambridge, 2001.

[8] Peter Selinger. A Survey of Graphical Languages for Monoidal Categories. In Bob Coecke, editor, *New Structures for Physics*, Lecture Notes in Physics, pages 289–355. Springer, Berlin, Heidelberg, 2011.


**Sequent calculus for implicational logic**

$$\frac{}{\varphi \vdash \varphi}\ \text{Ax.} \qquad \frac{\Delta, \varphi \vdash \psi}{\Delta \vdash \varphi \to \psi} \to \text{I.} \qquad \frac{\Delta \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Delta, \Gamma \vdash \psi} \to \text{E.} \tag{1}$$

[3] Actually, $U$ is a monoidal functor between monoidal categories. This way, $U$ is a refinement system [6].