The proof equivalence problem for multiplicative linear logic is PSPACE-complete v0.2

Willem Heijltjes and Robin Houston · July 9, 2013

MLL proof equivalence is the problem of deciding whether one proof may be turned into another by a series of commuting conversions. Thanks to many years of work on proof nets (Girard, 1987; Danos and Regnier, 1989; Trimble, 1994; Blute et al., 1996; Hughes, 2012) we know this is equivalent to a reasonably simple rewiring problem on graphs.

On the other side of the fence, thanks to many years of work on combinatorial games and complexity theory (Flake and Baum, 2002; Hearn and Demaine, 2005, 2009; Gopalan et al., 2006; Ito et al., 2011) we know many examples of rewiring problems on graphs that are PSPACE-complete. So in a sense it is not a great surprise to find that MLL proof equivalence is PSPACE-complete as well. On the other hand, I think this is the first time these different ideas have been brought together. We will give a reduction from the configuration-to-configuration problem for Nondeterministic Constraint Logic (Hearn and Demaine, 2005, 2009).

Nondeterministic Constraint Logic

A *constraint graph* is an undirected graph with an assignment of a natural number *weight* to each edge and a natural number *minimum in-flow constraint* to each vertex. A *configuration* of the constraint graph is an orientation (direction) of the edges such that the sum of incoming edge weights at each vertex is at least the minimum in-flow constraint of that vertex. A *move* from one configuration to another configuration consists of reversing the direction of a single edge such that the minimum in-flow constraints remain satisfied. The *configuration-to-configuration problem* is: given a constraint graph and two configurations s and t, is there a sequence of moves from s that results in t (or vice versa, since moves are reversible). This problem is PSPACE-complete. (Hearn and Demaine, 2005, 2009)

For present purposes, we need to introduce a slightly different notion of configuration. A *progressive configuration* assigns to each edge not only a direction but also an *orientation value*, which is a natural number at most the weight of the edge. The configuration is valid when the sum of the incoming orientation values at each vertex is precisely the minimum in-flow constraint of that vertex. A move consists of:

- either reversing the direction of an edge that has zero orientation value
- or taking a pair of edges oriented towards the same node, and incrementing the orientation value of one while decrementing the orientation value of the other.

Each progressive configuration determines a configuration in the Hearn-Demaine sense, and an edge of a progressive configuration may be reversed just when the edge of the corresponding Hearn-Demaine configuration may be. In this sense the notions are equivalent. We'll sometimes write PNCL to mean NCL with progressive configurations.

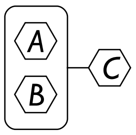
Hughes-style proof nets

We shall work in the unit-only fragment of MLL, where our formulas are built from the constants $\bot$ and 1 with the operations $\otimes$ and $⅋$, and we'll work with one-sided sequents and modulo associativity of $\otimes$ and $⅋$.

In this setting, a Hughes-style proof net for a given sequent is simply a function from the occurences of $\bot$ to the occurences of 1 such that the usual proof net criterion is satisfied. A rewiring step consists of moving a single $\bot$-link to a different occurence of 1.

Notation

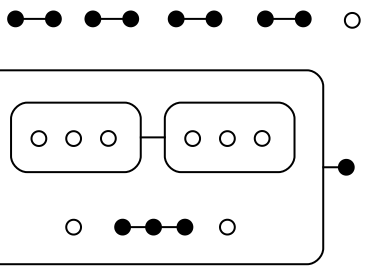
We'll use a diagrammatic notation for sequents and proof nets. The atoms 1 and $\bot$ are represented by a circle ∘ and a disc • respectively. Then we build up larger expressions in the following way. Suppose A is represented by $\langle A \rangle$ and B is represented by $\langle B \rangle$; then $A \parr B$ is $\langle A \rangle \ \langle B \rangle$ and $A \otimes B$ is $\langle A \rangle\!-\!\langle B \rangle$. Tensor of multiple elements is denoted by stringing them together in a line, so $A \otimes B \otimes C$ is $\langle A \rangle\!-\!\langle B \rangle\!-\!\langle C \rangle$. Boxes play the role of parentheses, so $(A \parr B) \otimes C$ is drawn as



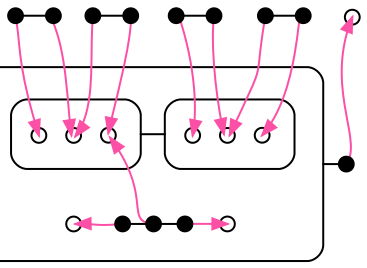
So, for example, this sequent

$$\vdash \bot \otimes \bot, \bot \otimes \bot, \bot \otimes \bot, \bot \otimes \bot, 1, \{[(1 \parr 1 \parr 1) \otimes (1 \parr 1 \parr 1)] \parr 1 \parr (\bot \otimes \bot \otimes \bot) \parr 1\} \otimes \bot$$

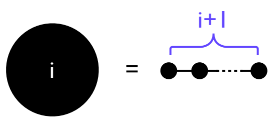
could be drawn like this:

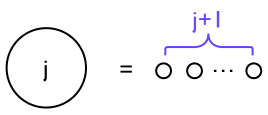


We represent a proof net by drawing an arrow from each • to some ∘. For example, one proof net on the above sequent is

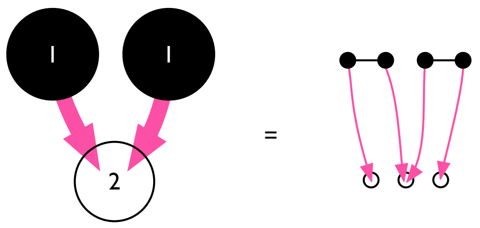


It will be useful to have an abbreviated notation for expressions of the form $\bot \otimes \cdots \otimes \bot$ and $1 \parr \cdots \parr 1$. We shall write a larger disc with the number i inside for the tensor of $i + 1$ occurences of $\bot$, and a larger circle with j inside for the par of $j + 1$ occurences of 1, like so:

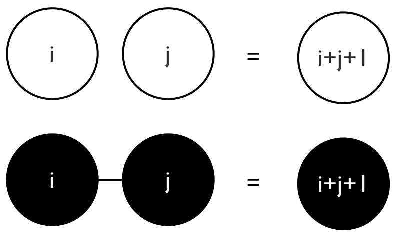




We also use fat arrows to represent a series of arrows in a proof net, like so:

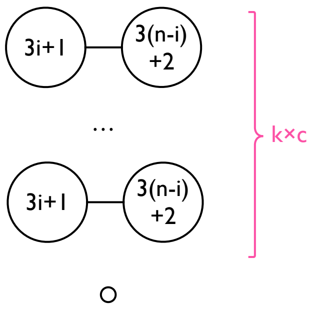


This is the reason for the '+1' in the definition: it means that a valid proof net is formed by a collection of fat arrows with a common target just when the numbers written on the sources add up to the number written on the target. The price we pay for this convenience is a less obvious arithmetic of decomposition:

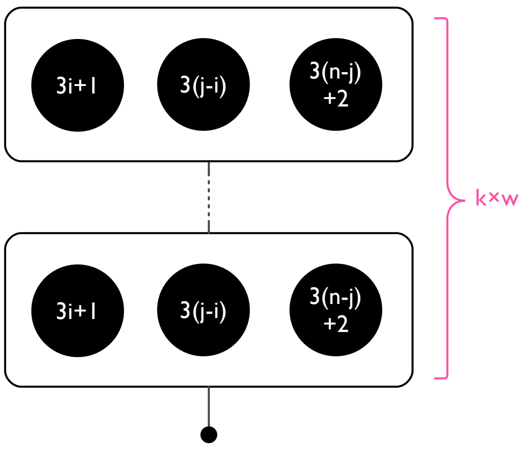


Reduction

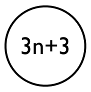
The idea is to encode a constraint graph as a sequent, in such a way that proof nets over the sequent correspond to progressive configurations of the constraint graph, and each PNCL move corresponds to a sequence of rewiring steps in the proof net. First we number the nodes from 0 to n, and let k be the number of edges. Then we represent node i with minimum inflow constraint c by the following gadget:



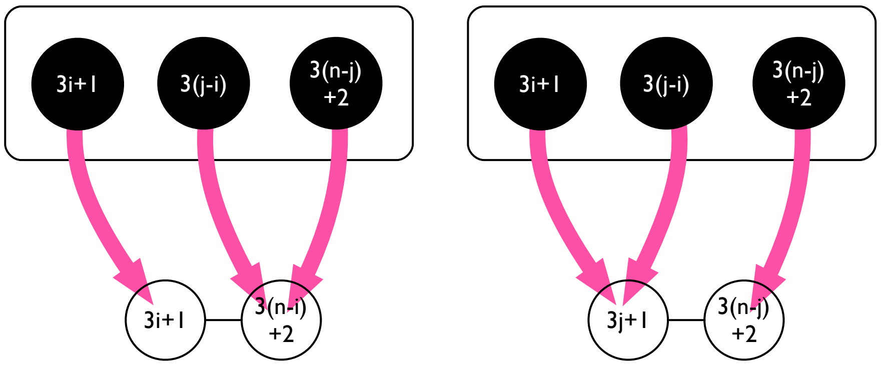
and an edge of weight w connecting nodes $i < j$ by the following:



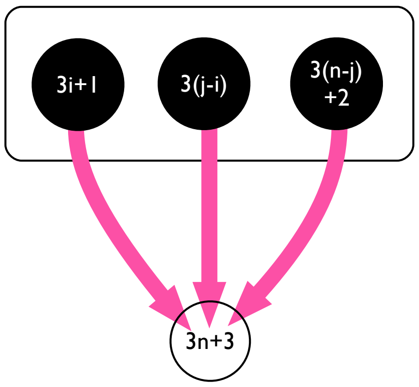
The other gadget we need is the "edge absorber":



The key property of this encoding is that a row of an edge gadget may be attached completely to a row of a node gadget just when the edge is attached to that node in the constraint graph, as follows:



Similarly, a row of any edge gadget may be attached completely to an edge absorber:



Let d be the difference between the sum of the edge weights and the sum of the minimum inflow constraints. Our sequent consists of:

- All the edge gadgets (as separate entries),
- The tensor of all the node gadgets (a single entry),
- $k \times d$ edge absorbers (as separate entries).

Now, every progressive configuration of the constraint graph may be represented by a proof net over this sequent, in the following way:

- Each edge is attached to the the node it is oriented towards, by attaching the small $\bullet$ at the bottom of the edge gadget to the small $\circ$ at the bottom of the node gadget. We'll call this the *index link*.
- kv rows of the edge gadget are attached to kv rows of the node gadget as described above, where v is the edge's orientation value.
- The remaining edge weight is assigned to edge absorbers, one row of the edge gadget to a single edge absorber. In total an edge of weight w and orientation value v uses $k(w - v)$ edge absorbers.

Of course there are many proof nets over our sequent that are not precisely of this form. We shall try to establish xxxx things:

- Every proof net over this sequent corresponds (in a sense we shall define precisely) to a slightly generalised form of progressive configuration;
- xxxx
- Every move in the PNCL game may be simulated by some sequence of proof net rewirings.
- If we start with a proof net that represents a configuration x, then perform a sequence of rewirings and end up with another proof net that represents a different configuration y, then there is a sequence of PNCL moves from x to y.

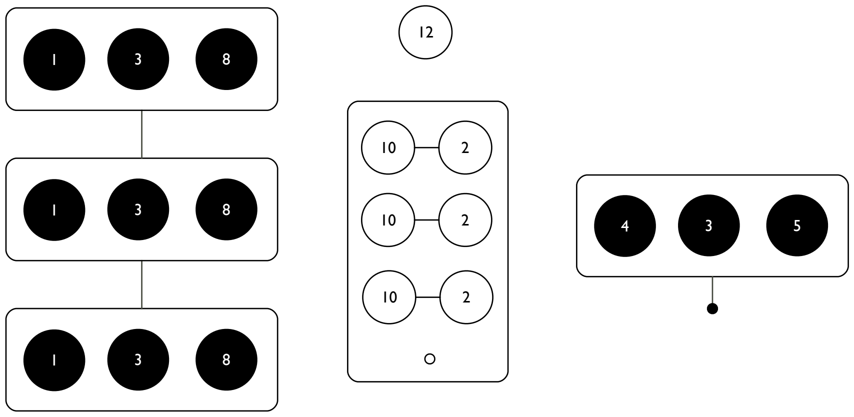
However the converse is false: in general it is possible to simulate certain invalid moves as well. The next section describes how this can happen, and how to fix it.

Naughty proof nets

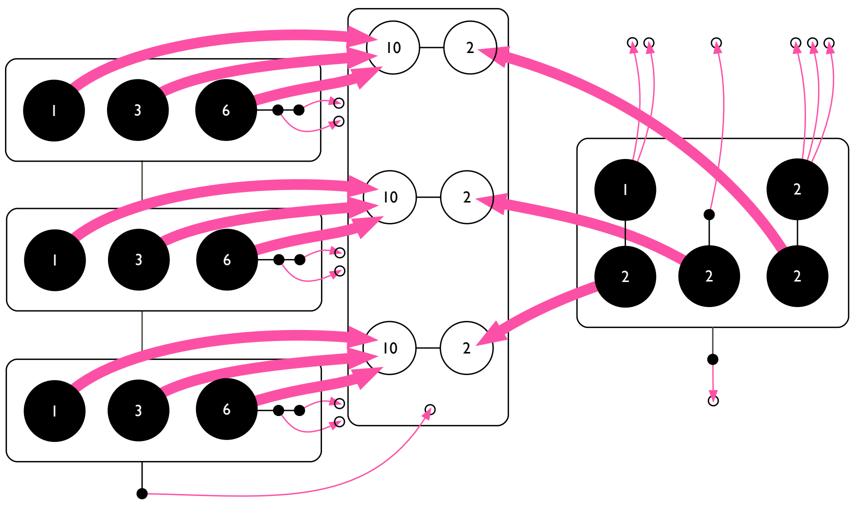
Although it is clear that a row of the node gadget may be filled by an edge gadget only when the edge is attached to that node, it is possible for several edge gadgets to join forces and fill part of a node gadget illegitimately. For example, suppose we have the following constraint graph with four nodes

$$\begin{tikzcd}
0 \arrow[r, dash, "w=3"] & 1 \arrow[r, dash, "w=1"] & 2 \\
& 3_{c=3} &
\end{tikzcd}$$

Then it is possible for the two edges to fill the gadget representing node 3 and one edge absorber. The relevant gadgets are these:



and one illegitimate wiring looks like this:



We have decomposed the edge absorber into 13 individual ∘ nodes that are scattered around the diagram, and decomposed some of the other nodes into parts, to make the diagram more legible.

The trick we use to overcome this problem is to multiply all the edge weights and minimum inflow constraints by a constant factor that is large enough to make it impossible for an edge to be freed illegitimately using a naughty wiring. The point here is that as w and c become larger, these naughty wirings use the edges more and more inefficiently. To make this precise, we'll use a little combinatorial lemma:

Lemma 1. *A non-empty bipartite forest whose leaves are all in the same part has more vertices in the part that contains the leaves than in the part that doesn't.*

Proof. Here a forest is an acyclic graph, a leaf is a vertex that has fewer than two neighbours, and bipartite means the nodes are coloured red and blue in such a way that every arc connects a red and a blue node. Without loss of generality, by symmetry, we assume the leaves are red; we want to show there are more red vertices than blue ones.

It suffices to show this for trees, i.e. connected acyclic graphs, since a forest is a union of trees. So take a tree: either all its nodes are red, in which case we are done, or it has some blue non-leaf nodes, in which case pick one to be the root and direct all the edges towards it. Every vertex has a unique successor (or parent) under this orientation, and since the leaves are red and the colours alternate along each path this successor relation determines a surjection from the red nodes onto the blue ones. Since the root has more than one predecessor by hypothesis, this surjection cannot be a bijection, so there must be strictly more red vertices than blue ones. □

Now, a wiring of our gadgets determines a bipartite graph of this sort, in the following way. The blue vertices are going to be the rows of node gadgets, specifically the rows that are linked to more than one edge gadget. (Our aim is to find an upper bound on the number of such rows.) The red vertices are going to be the *columns* of the edge gadgets.

When a row of a node gadget is filled by more than one edge gadget, there must in any case be some pair of edge gadgets that touch opposite sides of the tensor in the node gadget. For each such row, choose some such pair, and add a pair of corresponding arcs to the graph.

This graph must be acyclic by the proof net criterion. So the number of naughtily-filled rows is less than the total number of edge gadget columns, which is three times the number of edges in the constraint graph. It follows that, when constructing a sequent from a constraint graph, we may multiply all our w and c values by a factor m equal to thrice the number of edges. Since we know that fewer than m rows of any node gadget may be filled in a naughty way, and since each edge gadget has a number of rows that is a multiple of m, it can never now happen that an edge is released from a node when it ought not to have been.

[TODO: Explain how edge absorbers can be moved about when an edge gadget moves to a different node, so that it doesn't matter how the edge absorbers are attached to the edges in the two proof nets.]

Remarks

It is PSPACE-complete to decide, for a proof net, whether a particular link may be moved. This follows from the PSPACE-completeness of the configuration-to-edge problem for NCL.

I believe proof equivalence is decidable in linear time for the intuitionistic fragment of MLL, which is an interesting (and surprising?) contrast to the classical case. Dominic Hughes and I have an old unfinished draft showing this – it is basically all Dominic's work, to be honest – and I think the argument there is sound. There's an attempt at it in Audrey Tan's thesis, which unfortunately contains serious errors. Our draft is an attempt at a correct proof based on Tan's ideas. Does it follow from any correct existing work?

There is an unresolved question about depth-limited formulas, where by depth I mean how deeply nested the boxes are in the diagram. At depth zero – where no boxes are used, and so every entry in the sequent is a tensor of atoms – there is a straightforward (but not entirely trivial) linear-time algorithm. The reduction above shows that depth two is PSPACE-complete. So what happens at depth one? This is an open question.

References

Richard Blute, Robin Cockett, Robert Seely, and Todd Trimble. Natural deduction and coherence for weakly distributive categories. *Journal of Pure and Applied Algebra*, 113: 220–296, 1996.

Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

Gary William Flake and Eric B. Baum. Rush hour is PSPACE-complete, orfh "why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270 (1–2):895–911, 2002. ISSN 0304-3975. doi: 10.1016/S0304-3975(01)00173-6. URL `http://www.sciencedirect.com/science/article/pii/S0304397501001736`.

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

Parikshit Gopalan, Phokion G. Kolaitis, Elitza N. Maneva, and Christos H. Papadimitriou. The connectivity of boolean satisfiability: Computational and structural dichotomies. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 346–357. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35904-3. doi: 10.1007/11786986_31. URL `http://research.microsoft.com/pubs/77674/jsiam.pdf`.

Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1–2):72–96, 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.05.008.

Robert A Hearn and Erik D Demaine. *Games, puzzles, and computation.* AK Peters, Ltd., 2009.

Dominic J.D. Hughes. Simple free star-autonomous categories and full coherence. 2012. arXiv:math.CT/0506521.

Takehiro Ito, Erik D. Demaine, Nicholas J.A. Harvey, Christos H. Papadimitriou, Martha Sideri, Ryuhei Uehara, and Yushi Uno. On the complexity of reconfiguration problems. *Theoretical Computer Science*, 412(12–14): 1054–1065, 2011. ISSN 0304-3975. doi: 10.1016/j.tcs.2010.12.005. URL `http://erikdemaine.org/papers/NPReconfiguration_TCS/`.

Todd Trimble. *Linear logic, bimodules, and full coherence for autonomous categories*. PhD thesis, Rutgers University, 1994.