Fast Verification of MLL Proof Nets via IMLL

ANDRZEJ S. MURAWSKI and C.-H. LUKE ONG
Oxford University Computing Laboratory

We consider the following decision problems:

ProofNet : Is a given multiplicative linear logic (MLL) proof structure a proof net?
EssNet : Is a given essential net (of an intuitionistic MLL sequent) correct?

In this paper we show how to obtain linear-time algorithms for EssNet. As a corollary, by showing that ProofNet is linear-time reducible to EssNet (by the Trip Translation), we obtain a linear-time algorithm for ProofNet.

We show further that it is possible to optimize the verification so that each node of the input structure is visited at most once. Finally, we present linear-time algorithms for *sequentializing* proof nets and essential nets, i.e. finding derivations of the underlying sequents.

Categories and Subject Descriptors: F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Complexity of proof procedures*; F.4 [**Theory of Computation**]: Mathematical Logic

General Terms: Algorithms,Verification

Additional Key Words and Phrases: Multiplicative linear logic, proof nets, essential nets, dominator trees, disjoint union find

Contents

1 Introduction **2**

2 Proof nets, essential nets and linear-time reduction **4**
2.1 MLL and Proof Nets 4
2.2 IMLL and Essential Nets 4
2.3 Linear-time reduction of ProofNet to EssNet 7

3 Verification by backtracking **9**
3.1 Algorithm A 10
3.2 Correctness of Algorithm A 11

4 Verification using dominator trees **13**
4.1 Structure of dominator trees of essential nets 14
4.2 Fast algorithms for computing dominator trees 14

Authors' address: OUCL, Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom.
E-mail: {Andrzej.Murawski,Luke.Ong}@comlab.ox.ac.uk

4.3 Constructing dominator trees of essential nets 15

5 One-pass verification algorithms 16
5.1 A one-pass algorithm for EssNet 16
5.2 A one-pass algorithm for ProofNet 17

6 Linear-time sequentialization algorithms 19
6.1 Sequentializing essential nets 19
6.2 Sequentializing essential nets in linear time 20
6.3 Sequentializing proof nets in linear time 25

1. INTRODUCTION

By MLL we mean the multiplicative $(\otimes, \parr, \text{-}^{\perp})$-fragment of linear logic due to Girard [1987]. We are concerned with algorithms for the decision problem:

ProofNet: Is a given MLL proof structure a proof net?

Guerrini [1999] gave the first linear-time algorithm for ProofNet at LICS 1999, which was an important and surprising discovery. His approach was based on a reformulation of Danos' contractibility criterion [1990], but expressed as a kind of unification problem; the algorithm uses a special linear-time case of the Disjoint Set Union Problem [Gabow and Tarjan 1985]. In this paper, we take a different and, in our view, conceptually simpler approach by first considering the corresponding problem for the intuitionistic fragment of MLL, i.e. the $(\otimes, \multimap)$-fragment of linear logic, denoted IMLL. For this sublogic, Lamarche [1994] has proposed a version of proof nets, called *essential nets*. The essential net of an IMLL sequent is a rooted directed graph of a certain kind constructed from the respective abstract syntax trees of the formulas in the sequent; nodes of the graph are labelled by polarized symbols (namely, $\parr^+, \parr^-, \otimes^+, \otimes^-, a^+, b^+, \cdots, a^-, b^-, \cdots$, where $a, b, \cdots$ range over a fixed set of atoms), and to each $\parr^+$-labelled node (or simply $\parr^+$-node) p, the construction associates a negatively-labelled node called the *sink* of p. Lamarche has shown that the essential net of an IMLL sequent is correct (in the sense that it denotes a proof) if and only if it is acyclic and every $\parr^+$-node p satisfies *Condition L*: every path from the root that reaches the sink of p passes through p. Our first result is a linear-time algorithm for the decision problem:

EssNet: Is a given essential net (of an IMLL sequent) correct?

Condition L is a property of a certain tree (called dominator tree) that is derived from the essential net in question. We can verify it though without constructing the tree by backtracking over the input net, i.e. traversing the net against the orientation of the edges. Naïvely checking Condition L for each $\parr^+$-node in turn would give a quadratic-time algorithm. Instead our algorithm visits the $\parr^+$-nodes in a topological order and uses a special linear-time case of the Disjoint Set Union Problem [Gabow and Tarjan 1985] for efficient bookkeeping[1], reducing its asymptotic

[1] Gabow and Tarjan's algorithm [1985] is used here for a different purpose from the application in [Guerrini 1999].

complexity to linear time. This is the first result of our paper, which is set out in Section 3.

However, it is also possible to compute the dominator tree in linear time which, as it turns out, contains sufficient information for sequentialization of essential nets. The problem of computing dominator trees is solvable in linear time even in the general case of an arbitrary flowgraph; several such algorithms have recently been obtained by various groups [Gabow 1990; Alstrup et al. 1999; Buchsbaum et al. 1998]. A *flowgraph* is just a directed graph such that every node of the graph is reachable from a fixed node called the root. Nodes of such graphs can be ordered by a notion of *domination*: we say that node v *dominates* w just in case every path from the root that reaches w passes through v. The ordering has a tree structure; we call the tree (which has the same underlying node-set as the graph, and whose edges induce the ordering) the *dominator tree* of the graph. As acyclic essential nets are flowgraphs, they are correct if, in addition, every $\parr^+$-node dominates its sink (this is just a paraphrase of Condition L).

A pleasing consequence is that we can also derive a linear-time algorithm for ProofNet. We say that a decision problem P is *linear-time reducible* to another Q just in case there is a linear-time computable function f from inputs of P to inputs of Q such that for any P-input x, x is a yes-instance of P if and only if $f(x)$ is a yes-instance of Q. It turns out that the trip translation of Bellin and van de Wiele [Bellin and Scott 1994], which takes a proof structure, a chosen conclusion and a DR-switching as input and returns an essential net (under certain conditions), gives a linear-time reduction of ProofNet to EssNet.

The ProofNet algorithms thus obtained, though linear time, make several passes over the input proof structure. In Section 5, we show that the linear-time algorithms for EssNet and ProofNet which compute the dominator tree can be optimized so that just one pass over the input structure suffices – each node of the respective input structure is visited at most once. We call them Algorithms C and D respectively. A notable feature is that both algorithms construct a dominator tree as a useful side effect, whenever the input is a YES-instance.

The domination ordering plays a key rôle in the substructure analysis of an essential net – it tells us how to decompose correct essential nets. For instance, the (node-set of the) *empire*[2] of a positive node of an essential net consists exactly of all nodes dominated by that node. In Section 6, by a method of systematic decomposition (of essential nets), we obtain linear-time algorithms, Algorithms E and F respectively, for *sequentializing* essential nets and proof nets.

Chronology. The results contained in Section 3 of this paper have appeared in [Murawski and Ong 2000a], and the rest have been announced in the Proceedings of LICS 2000 [Murawski and Ong 2000b].

[2]By the *empire* of a node of an essential net, we mean the “largest correct subnet” containing that node.

2. PROOF NETS, ESSENTIAL NETS AND LINEAR-TIME REDUCTION

2.1 MLL and Proof Nets

MLL formulas, ranged over by $A, B, C, \cdots$, are generated from atoms $(a, b, \cdots)$ and negative atoms $(a^\perp, b^\perp, \cdots)$ by the connectives $\otimes$ (tensor) and $\parr$ (par). Valid MLL sequents are defined[3] as follows: we let $\Gamma, \Delta, \cdots$ range over finite sequences of formulas:

$$\textbf{(id)} \quad \frac{}{\vdash a^\perp, a} \qquad\qquad \textbf{(tensor)} \quad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}$$

$$\textbf{(par)} \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B} \qquad\qquad \textbf{(exch)} \quad \frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}$$

A ***proof structure*** is a pair (Γ, ϕ) where Γ is an MLL sequent and ϕ, called an *involution*, is a bijection between occurrences of atoms and occurrences of negated atoms in Γ such that ϕ maps an occurrence of an atom to an occurrence of the negation of the same atom. A ***DR-switching*** σ [Danos and Regnier 1989] for (Γ, ϕ) is an assignment of "left" or "right" to each occurrence of $\parr$ in Γ. The associated ***DR-graph*** $\mathrm{Gr}(\Gamma, \phi, \sigma)$ is an undirected graph that is obtained from the disjoint union of the respective abstract syntax trees of the formulas in Γ by performing the following edge operations:

— for each occurrence e of an atom, add an edge (i.e. the axiom link) joining e and $\phi(e)$

— for each occurrence p of $\parr$, if $\sigma(p) =$ "left" (respectively "right"), delete the edge that joins p to its right (respectively left) child.

Intuitively, a proof net is a proof structure that is correct, i.e. one that denotes a proof. (As we are mainly concerned with provability and not proofs, we shall only consider *cut-free* proof nets.) Girard's original correctness criterion in [Girard 1987] has been simplified by Danos and Regnier [1989] and we use their version here: we define a ***proof net*** to be a proof structure (Γ, ϕ) such that for every DR-switching σ, the associated DR-graph $\mathrm{Gr}(\Gamma, \phi, \sigma)$ is connected and acyclic. Girard has shown that an MLL sequent $\vdash \Gamma$ is provable if and only if there is some ϕ such that (Γ, ϕ) is a proof net. From now on we shall only consider proof structures (Γ, ϕ) such that Γ is ***linearly balanced*** (i.e. each atom that appears in Γ has exactly one positive and one negative occurrence), so that ϕ is redundant. Thus there is no harm in identifying proof structures Γ with linearly balanced MLL sequents $\vdash \Gamma$.

2.2 IMLL and Essential Nets

IMLL formulas are generated from the atoms $a, b, c, \cdots$ by the connectives $\otimes$ and $\multimap$ (linear implication). Valid IMLL sequents are defined by the following rules:

[3] As we are only concerned with *cut-free* proofs, the Cut Rule is omitted from both the MLL and IMLL Sequent Calculus.

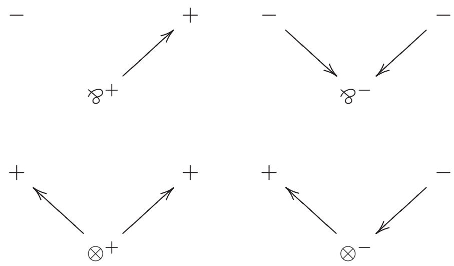


Fig. 1. The directional rules

$$
\begin{array}{llll}
\textbf{(var)} & a \vdash a & \textbf{(exch)} & \dfrac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \\[2ex]
(\otimes\textbf{-l}) & \dfrac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} & (\otimes\textbf{-r}) & \dfrac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \\[2ex]
(\multimap\textbf{-l}) & \dfrac{\Gamma \vdash A \qquad B, \Delta \vdash C}{\Gamma, A \multimap B, \Delta \vdash C} & (\multimap\textbf{-r}) & \dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}
\end{array}
$$

Given an IMLL formula A, we can define its classical counterpart $\overline{A}$ as follows:

$$
\begin{array}{rclrcl}
\overline{a} & = & a & \underline{a} & = & a^{\perp} \\
\overline{A \multimap B} & = & \underline{A} \parr \overline{B} & \underline{A \multimap B} & = & \overline{A} \otimes \underline{B} \\
\overline{A \otimes B} & = & \overline{A} \otimes \overline{B} & \underline{A \otimes B} & = & \underline{A} \parr \underline{B}
\end{array}
$$

By a simple induction over the rules of the IMLL Sequent Calculus, we have the following:

PROPOSITION 2.1. *For any IMLL formulas $C_1, \cdots, C_n$ and A, we have*

$$C_1, \cdots, C_n \vdash A$$

is IMLL-provable if and only if $\vdash \underline{C_1}, \cdots, \underline{C_n}, \overline{A}$ is MLL-provable; further there is a linear-time algorithm that transforms every derivation of the former (in IMLL) to a derivation of the latter (in MLL). □

Any linearly balanced IMLL formula A can be transformed to a polarized MLL formula $\ulcorner A \urcorner$, which is constructed from *polarized* atoms, namely,

$$a^+, b^+, c^+, \cdots, a^-, b^-, c^-, \cdots$$

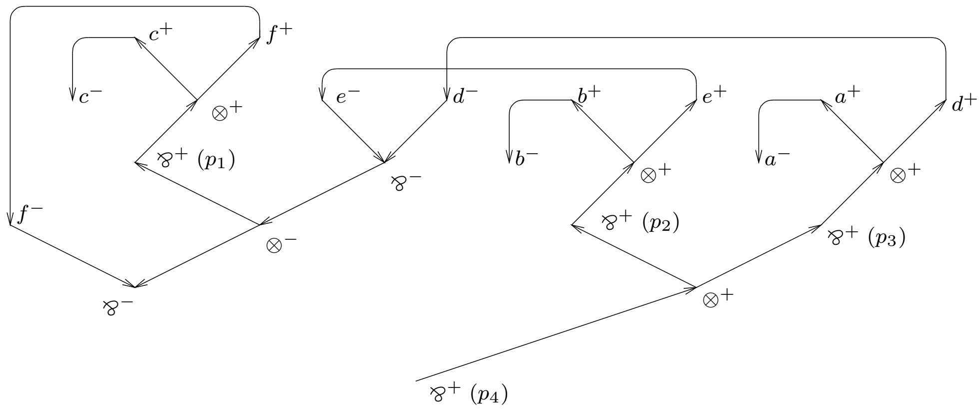


Fig. 2. The essential net of $\vdash\ f \otimes ((c \multimap c \otimes f) \multimap e \otimes d) \quad \multimap \quad (b \multimap b \otimes e) \otimes (a \multimap a \otimes d)$.

and polarized connectives (namely $\otimes^+, \otimes^-, \parr^+$ and $\parr^-$) by the following rules: $\ulcorner a \urcorner = a^+$, $\llcorner a \lrcorner = a^-$ and

$$\begin{aligned}
\ulcorner A \multimap B \urcorner &= \llcorner A \lrcorner \parr^+ \ulcorner B \urcorner \\
\llcorner A \multimap B \lrcorner &= \ulcorner A \urcorner \otimes^- \llcorner B \lrcorner \\
\ulcorner A \otimes B \urcorner &= \ulcorner A \urcorner \otimes^+ \ulcorner B \urcorner \\
\llcorner A \otimes B \lrcorner &= \llcorner A \lrcorner \parr^- \llcorner B \lrcorner
\end{aligned}$$

The ***essential net*** [Lamarche 1994] of a linearly balanced IMLL sequent

$$C_1, \cdots, C_n \vdash A,$$

denoted $\mathcal{E}(C_1, \cdots, C_n \vdash A)$, is a directed graph that is obtained from the disjoint union of the respective abstract syntax trees of $\llcorner C_1 \lrcorner, \cdots, \llcorner C_n \lrcorner$ and $\ulcorner A \urcorner$ by augmenting it with oriented axiom links (the direction is from a positive atom a^+ to its negative counterpart a^-), and by orienting edges incident to the connective-nodes (namely $\parr^+, \parr^-, \otimes^+$ and $\otimes^-$) according to the ***directional rules*** in Figure 1. Note that there is no edge between a $\parr^+$-node and its negative left "child", which we call its ***sink***.

Example 2.2. The essential net of

$$\vdash\ f \otimes ((c \multimap c \otimes f) \multimap e \otimes d) \quad \multimap \quad (b \multimap b \otimes e) \otimes (a \multimap a \otimes d)$$

is shown in Figure 2. (By convention $\otimes$ binds more strongly than $\multimap$.)

An essential net is said to be ***correct*** if it is acyclic and every $\parr^+$-node p satisfies ***Condition L***: every path from the root that reaches p's sink passes through p. For example, the reader may wish to check that $\mathcal{E}(a \multimap a \otimes b \vdash b)$ is cyclic, each $\parr^+$-node of $\mathcal{E}(\vdash\ (a \multimap b) \otimes (b \multimap a))$ violates Condition L, but the essential net in Figure 2 is correct. An important result in [Lamarche 1994] is a characterization of IMLL provability:

THEOREM 2.3 LAMARCHE. *A linearly balanced IMLL sequent* $\Gamma \vdash A$ *is provable if and only if its essential net* $\mathcal{E}(\Gamma \vdash A)$ *is correct.*

PROOF. For a proof using empires see [Lamarche 1994]. A different proof based on reduction to a canonical form can be found in [Murawski and Ong 2003]. The sequentialization results presented in this paper also constitute a new proof of the result. ☐

Remark 2.4. It is straightforward to prove that the *reachability condition*, i.e. every node is reachable from the root, is a consequence of acyclicity.

2.3 Linear-time reduction of PROOFNET to ESSNET

(*Henceforth we shall identify proof structures* Γ *with linearly balanced MLL sequents* $\Vdash \Gamma$.) Suppose we have a proof structure Γ (with a chosen formula B in Γ) and a DR-switching σ for it such that the DR-graph $\mathrm{Gr}(\Gamma, \sigma)$ is connected and acyclic – this assumption is important. We give a procedure, called *Trip Translation*, that returns an essential net $\mathcal{E}(\Gamma, B, \sigma)$. (For any edge (u, v) of a directed graph, we say that u is a ***predecessor*** of v, or equivalently v is a ***successor*** of u.)

Notation. When describing algorithms, we use bold semicolon **;** to denote sequential composition (of commands).

The Algorithm could be thought of simply as a traversal of the DR-graph $\mathrm{Gr}(\Gamma, \sigma)$ starting from the chosen conclusion B, which assigns orientation to edges (consistent with the direction of traversal) and polarity to nodes along the way: it assigns "+" to nodes that are reached from below, and "−" to nodes reached from above, with the two ends of an axiom link given opposite polarities. The direction of traversal follows the orientation of edges defined in the directional rules in Figure 1. Note that the Algorithm ensures that a $\invamp$-node p which is assigned "−" is reached from above through the child-node indicated by the switching σ at p. The polarity assigned to an atom in the output net may be different from its polarity in the input proof structure, i.e. $a^{\perp}$ could be assigned + and hence appear as a^{+} in the output net. For instance, consider the case where the input sequent is $\Vdash a \invamp a^{\perp}$ with σ set to the right switch.

THEOREM 2.5 BELLIN-VAN DE WIELE. *The trip translation terminates and is correct: for each* σ *such that* $\mathrm{Gr}(\Gamma, \sigma)$ *is acyclic and connected, and for each choice* B *of conclusion, the Algorithm returns* $\mathcal{E}(\Gamma, B, \sigma) = \mathcal{E}(C_1, \cdots, C_n \vdash A)$ *for some IMLL sequent* $C_1, \cdots, C_n \vdash A$*. Moreover,* $\Vdash \Gamma$ *is MLL-provable iff* $\Vdash \underline{C_1}, \cdots, \underline{C_n}, \overline{A}$ *is.*

PROOF. Since $\mathrm{Gr}(\Gamma, \sigma)$ is acyclic and connected, it is easy to see that the Trip Translation Algorithm will traverse it, assigning polarities to all nodes and YES will be returned; in addition, the polarities and orientation must observe the directional rules (see Figure 1).

To show that $\Vdash \underline{C_1}, \cdots, \underline{C_n}, \overline{A}$ is MLL-provable iff $\Vdash \Gamma$ is, consider the syntactic forests of both sequents and observe that there exists an isomorphism between them which preserves the labels of nodes except that for some atoms a it maps a to $a^{\perp}$ and vice versa. Therefore for any DR-graph associated with the proof structure of one of the sequents, there exists an isomorphic DR-graph associated with the proof

Algorithm *Trip Translation*

Input: an MLL proof structure Γ, a chosen conclusion B (in Γ), a DR-switching σ
Output: YES (with $N = \mathcal{E}(\Gamma, B, \sigma)$) if $\mathrm{Gr}(\Gamma, \sigma)$ is acyclic and connected, NO otherwise

1. Initialize N to the singleton tree consisting of the node v_0 corresponding to B. (N.B. $\boldsymbol{assign}(p, v)$ assigns polarity p to node v of N, and $\boldsymbol{link}(w, v)$ makes w a successor node of v in N.)
2. Run $\boldsymbol{polarize}(+, v_0)$, where:

   $\boldsymbol{polarize}(p : \mathit{Polarity}, v : \mathit{Node}) \quad =$
   if (v already visited) then return NO **;**

   $\boldsymbol{assign}(p, v)$ **;**

   Cases of (p, v) :
   – $(+$, end of axiom link ending in $v')$ $\quad\rightarrow\quad$ $\boldsymbol{link}(v', v)$ **;** $\boldsymbol{polarize}(-, v')$
   – $(+$, $\otimes$-node$)$ where v_1 and v_2 are the nodes above v $\quad\rightarrow$
   $\qquad\boldsymbol{link}(v_1, v)$ **;** $\boldsymbol{polarize}(+, v_1)$ **;** $\boldsymbol{link}(v_2, v)$ **;** $\boldsymbol{polarize}(+, v_2)$
   – $(+$, $⅋$-node$)$ $\quad\rightarrow\quad$ $\boldsymbol{link}(v', v)$ **;** $\boldsymbol{polarize}(+, v')$
   where v' is the node above v indicated by $\sigma(v)$
   – $(-, v)$ $\quad\rightarrow$
   if v is a $\otimes$-node then $\{\boldsymbol{link}(v', v)$ **;** $\boldsymbol{polarize}(+, v')\}$
   where v' is the node above v other than the one just visited

   if v has a node w below then
   Cases of w :
   – $\otimes$-node $\quad\rightarrow\quad$ $\boldsymbol{link}(w, v)$ **;** $\boldsymbol{polarize}(-, w)$
   – $⅋$-node $\quad\rightarrow$
   if $((w, v)$ same side as $\sigma(w))$ then
   $\{\boldsymbol{link}(w, v)$ **;** $\boldsymbol{link}(w, v')$ **;** $\boldsymbol{polarize}(-, w)\}$
   where v' is the other node above w.
3. If every node has been visited by $\boldsymbol{polarize}(+, v_0)$ then return YES else return NO.

structure of the other. This proves that the two sequents are MLL-equivalent as required. $\square$

Example 2.6. (i) Take the input sequent to be $\Vdash (a^\perp ⅋ b) \otimes c^\perp, (a \otimes b^\perp) ⅋ c$; and let σ assign "left" to the two $⅋$-occurrences. Taking the first formula of Γ as the conclusion, the trip translation returns the essential net of $(b \multimap a) \otimes c \vdash (b \multimap a) \otimes c$; taking the second formula as the conclusion, it returns the essential net of

$$c \multimap a \otimes b \vdash c \multimap a \otimes b.$$

(ii) The Algorithm returns NO if the DR-graph $\mathrm{Gr}(\Gamma, \sigma)$ is disconnected or cyclic; e.g. take the input sequent to be

$$\Vdash (a^\perp ⅋ b^\perp) ⅋ (a ⅋ b) \text{ and } \Vdash (a^\perp \otimes b^\perp) ⅋ (a \otimes b)$$

respectively, and any DR-switching.

(iii) Note that the essential net $\mathcal{E}(\Gamma, B, \sigma)$ returned in a YES-instance may be cyclic even though $\mathrm{Gr}(\Gamma, \sigma)$ is acyclic (because a $⅋^-$-node must have two incoming edges). E.g. take the input sequent to be $\Vdash b^\perp \otimes (b ⅋ a^\perp), a$, and B to be the second formula, with σ mapping the $⅋$-node to "right"; the algorithm returns $\mathcal{E}(b \multimap b \otimes a \vdash a)$.

THEOREM 2.7 LINEAR-TIME REDUCIBILITY. *For any proof structure* Γ

(i) if Γ *is a proof net then for every DR-switching* σ *and every* B, $\mathcal{E}(\Gamma, B, \sigma)$ *is a correct essential net,*

(ii) if for some σ *and* B*, the trip translation returns YES and if* $\mathcal{E}(\Gamma, B, \sigma)$ *is correct, then* Γ *is a proof net.*

Thus the decision problem PROOFNET *is linear-time reducible to* ESSNET.

PROOF. (i) follows by successive appeals to Theorem 2.5, Proposition 2.1 and Theorem 2.3. For (ii) one should use them in reverse order noting that $\Vdash \Gamma$ is MLL-provable if and only if Γ is a proof net.

As each node is visited by ***polarize*** at most once, the Trip Translation Algorithm is linear-time. □

3. VERIFICATION BY BACKTRACKING

Recall that an essential net is correct just in case it is acyclic and every $\wp^+$-node satisfies Condition L. As the acyclicity of a directed graph can be checked (by a depth-first search) in linear time, to verify essential nets efficiently, one should try to optimize the checking of Condition L. We present two solutions. The first will be based on backtracking with efficient bookkeeping provided by the special case of the Disjoint Union Problem. The second computes the whole dominator tree instead, but can be optimized to a one-pass algorithm and accumulates enough information for a linear-time sequentialization algorithm.

First a quick introduction to the Disjoint Set Union Problem (see e.g. [Cormen et al. 1990, Chap. 22]). The problem concerns three standard operations: we let v and v' range over a finite set of (input) elements

makeset(v) : construct a new singleton set $\{ v \}$ whose *name* (or representative) is v

find(v) : return the name of the set containing v

union(v, v') : construct[4] a new set that is the union of the respective sets containing v and v'; the new set is named by the name of the old set containing v.

Gabow and Tarjan [1985] have shown that there is a linear-time algorithm for the special case where the structure of the unions (defined by a "union tree") is known in advance. They call the special case *Static Tree Set Union Problem*, which can be stated as follows. We are given a rooted tree T of n nodes. Initially every node v of the tree is in a singleton set $\{ v \}$ named v, so that **find**$(v) = v$. Let $parent(v)$ be the parent of v in T. We wish to perform on-line (i.e. each operation must be completed before the next one is known) an intermixed sequence of m **find** and **link** operations where **link**(v) is equivalent to **union**$(parent(v), v)$. The Gabow-Tarjan algorithm has time complexity $O(m + n)$. In the following, we shall make use of their algorithms for **find** and **link**.

[4]This operation assumes that v and v' are initially in different sets, and destroys the old sets containing v and v'.

Algorithm A

Input: an acyclic essential net N with $\wp^+$-nodes $p_1, \cdots, p_k$ and root r
Output: YES if the essential net is correct, NO otherwise

1. Apply *Backtrack* to each $\wp^+$-node in any order such that the procedure is performed on a $\wp^+$-node only if it has been performed on all of its children. We can derive one such order from a depth-first search of N.

   Backtrack(p_i): *Visit*(s_i, i) where s_i is the sink of p_i.

   Visit(v, i) = *Cases of* v:
   1. v unlabelled $\wedge\ v = p_i \ \rightarrow$ label v by i
   2. v unlabelled $\wedge\ v = p_j \wedge j \neq i \ \rightarrow$ return NO
   3. v unlabelled $\wedge\ v$ not a $\wp^+$-node $\ \rightarrow$ if $v = r$ then return NO else { **do** *Visit*(u, i) **for** each predecessor u of v **;** label v by i}
   4. v labelled by $i \ \rightarrow$ **stop**
   5. v labelled by some $j \neq i \wedge \mathbf{find}(p_j) = p_i \ \rightarrow$ **stop**
   6. v labelled by some $j \neq i \wedge \mathbf{find}(p_j) = p_l \wedge l \neq i \ \rightarrow$
      Visit(u, i) **;** $\mathbf{link}(p_l)$
      where u is the predecessor of p_l
2. return YES

Fig. 3. Algorithm A for EssNet

3.1 Algorithm A

Fix an acyclic essential net N. Recall that given any finite directed graph, it is linear-time decidable if it is cyclic.

Let $p_1, \cdots, p_k$ be the $\wp^+$-nodes of N and r its root (which may happen to be a $\wp^+$-node). We define the relation $v > v'$ for $v, v' \in \{p_1, \cdots, p_k, r\}$ to mean that there is a non-trivial path in N from v to v' without passing through any $\wp^+$-nodes. The relation defines an acyclic connected graph, though it is not in general a tree (see Example 3.1). Let T_N be *any* spanning tree of this graph. We can obtain one such directly from N in linear time by a depth-first search starting from r. T_N will play the role of the union tree in our algorithm. Henceforth, whenever we mention the ancestor, descendant, parent or child node of some $\wp^+$-node, it is with reference to a chosen T_N. However, predecessors and successors will be calculated with respect to the directed graph N. Observe that each node different from the root has at least one predecessor.

We present the main Algorithm in Figure 3. Note that Algorithm A returns "NO" as soon as a "return NO" is executed in any run of *Visit*. We refer to Case 6 as a *jump* (from the j-labelled v) to p_l. The invariants that we are going to prove later guarantee that p_l is never a root and hence a predecessor u will always exist.

Example 3.1. Take the provable IMLL formula

$$f \otimes ((c \multimap c \otimes f) \multimap e \otimes d) \quad \multimap \quad (b \multimap b \otimes e) \otimes (a \multimap a \otimes d)$$

whose essential net we have given in Figure 2. There are four $\wp^+$-nodes, namely, p_1, p_2, p_3 and $p_4 = r$, as indicated in parentheses in the Figure. The $>$-relation has the shape of a diamond, with p_4 at the top and p_1 at the bottom. Note that during *Backtrack*(p_2), no node labelled 1 was visited (for otherwise, Condition L would have to fail for p_2 as there would be a backtracking path from p_2's sink to the root via p_3), so that $\mathbf{find}(p_1)$ is still p_1 at the end of *Backtrack*(p_2); similarly for *Backtrack*(p_3). However $\mathbf{find}(p_i) = p_4$ for all i at the end of *Backtrack*(p_4).

3.2 Correctness of Algorithm A

For any $\wp^+$-node p, Condition L is equivalent to: every backtracking path from p's sink reaches back to p. Note that at the start of each *Backtrack*(p_i), no node has label i. The effect of *Backtrack*(p_i) is to label certain nodes i; throughout Algorithm A no node is ever labelled more than once. The intuition is that the label i annotated to a node is a certificate that all backtracking paths from it reach back to p_i. Though the label, once given, is never changed, the name of each p_i (as given by $\mathbf{find}(p_i)$) can and does change from time to time during the course of Algorithm A, subject to the invariant that if v is labelled j and $\mathbf{find}(p_j) = p_l$ then every backtracking path from v reaches back to p_l (see (I1) below). The efficiency of the Algorithm is achieved by ensuring that the real work of checking each node is never duplicated. We find it useful to identify several invariants for the correctness proof of Algorithm A. If v has a label j and $\mathbf{find}(p_j) = p_l$, we refer to l as the *name* of v.

***Invariant for* Algorithm A**

(I1) If v is labelled j and $\mathbf{find}(p_j) = p_l$ then every backtracking path from v reaches back to p_l, passing only through nodes named l.

Invariants for *Backtrack*(p_i): During the execution of *Backtrack*(p_i)

(I2) whenever $\mathbf{find}(p_j)$ changes, the new name is p_i

(I3) if the label of v is j at the completion of *Visit*(v, i) then $\mathbf{find}(p_j) = p_i$.

Before we show that the invariants hold let us state a lemma.

LEMMA 3.2. *Suppose that the invariants hold and Backtrack(p_i) is to be called next. Then:*

(i) $\mathbf{find}(p_j) = p_i$ *if and only if* $j = i$.

(ii) The first node for which Visit(v, i) is going to succeed (provided it succeeds for some node) is p_i. Therefore, p_i is the first node to be labelled i.

(iii) When p_i is labelled, $\mathbf{find}(p_i) = p_i$ *holds.*

PROOF. At the very beginning we have $\mathbf{find}(p_j) = p_j$ for each j. By (I2) before the start of *Backtrack*(p_i) we have $\mathbf{find}(p_j) \neq p_i$ for each label $j \neq i$. Therefore $\mathbf{link}(p_i)$ has never been invoked and so $\mathbf{find}(p_i) = p_i$ at all times up to the start of *Backtrack*(p_i).

Visit(v, i) can only succeed for the first time in Cases 1 or 5, but Case 5 is ruled out by (i), so $v = p_i$. Then we still have $\mathbf{find}(p_i) = p_i$, because no **find** values could change since *Backtrack*(p_i) was called. □

It is a consequence of (I2) that there is always a predecessor available in Case 6. If $r = p_l$, then *Backtrack*(p_l) is invoked at the very end. Thus, by Lemma 3.2 (i), $\mathbf{find}(p_j) \neq p_l$ for $j \neq l$ before *Backtrack*(p_l) is called and we have $l = i$ during *Backtrack*(p_l). Hence, Case 6 cannot take place if $r = p_l$.

Now we verify the invariants immediately after their parameters have changed. Thus we need to check the following cases:

Invariant	Cases to check
(I1)	1, 3, 6
(I2)	6
(I3)	1, 3, 4, 5, 6

Case 1 : As argued in the proof of Lemma 3.2 no **find** value has changed since the start of *Backtrack*(p_i) so (I1) holds for all nodes that are distinct from $v = p_i$. p_i is the only node with label i and we have **find**$(p_i) = p_i$ by Lemma 3.2, so (I3) holds. (I1) is thus trivially satisfied for $v = p_i$.
Case 3 : To verify (I3) for v we have to check that **find**$(p_i) = p_i$, which follows from Lemma 3.2 (the equality held at the start of *Backtrack*(p_i)) and (I2) (it has not changed since). By (I3) and (I1) for the predecessors of v (if *Visit*(v, i) succeeds then $v \neq r$), we have (I1) for v, because each backtracking path from v must go through one of its predecessors.
Case 4 : By (I3) at the time v was labelled and (I2) we have **find**$(p_i) = p_i$; hence (I3) holds.
Case 5 : (I3) is satisfied trivially, because the assumption is **find**$(p_j) = p_i$.
Case 6 : By (I3) for *Visit*(u, i), if u has label h, **find**$(p_h) = p_i$, and by (I1) at that moment, each backtracking path reaches back to p_i passing only through nodes with name i. Therefore **find**$(\mathit{parent}(p_l)) = p_i$ (as $\mathit{parent}(p_l)$ lies on one such path) and **link**(p_l) changes **find**(p_l) to p_i as required in (I2). Hence, **find**$(p_j) = p_i$ thus satisfying (I3).

The operation **link**(p_l) changes the name of every node previously named l to i (the rest remain unchanged). By (I1) immediately before **link**(p_l), each backtracking path from a node named l reaches back to p_l, passing only through nodes named l; and, by (I1) applied at the moment *Visit*(u, i) is completed, any backtracking path from u reaches back to p_i passing only through nodes with name i. Thus each backtracking path from any node named l before **link**(p_l) is performed must reach back to p_i; and, after **link**(p_l) is performed, it can only pass through nodes named i. Thus, (I1) holds for the nodes whose names have changed.

In conclusion, for each i, when *Backtrack*(p_i) succeeds, p_i's sink must be named i by (I3), and by (I1) it satisfies Condition L. Therefore, if Algorithm A returns YES, Condition L holds.

Lemma 3.3. *Suppose Backtrack applied to all of p_i's descendants has succeeded. If Backtrack(p_i) fails, then p_i does not satisfy Condition L.*

Proof. The precondition for executing *Backtrack*(p_i) is that *Backtrack* has succeeded for every $\otimes^+$-node which is a child of p_i. This implies that every $\otimes^+$-node that can only be reached from the root via p_i has been labelled. Thus, if an unlabelled $\otimes^+$-node p_j such that $j \neq i$ is visited during *Backtrack*(p_i) (Case 2), there will be a path from the root to p_j which does not pass through p_i. That path combined with the backtracking path from the sink of p_i to p_j (determined by

Backtrack(p_i)) will yield a path that violates Condition L. To show this, we need to verify that each jump from v to u (Case 6) can be replaced with a path from u to v which does not pass through p_i. By (I2) each jump can be replaced with a path from u to v passing only through nodes labelled with $j \neq i$. Since p_i cannot have been labelled with $j \neq i$ (some preceding *Backtrack* would fail then because of Case 2), p_i fails Condition L.

If the failure results from Case 3, i.e. the root is reached, we can produce a path violating Condition L in an analogous way. □

Lemma 3.4. *Algorithm A terminates.*

Proof. Since N is acyclic, non-termination can only arise because of jumps. So suppose during *Backtrack*(p_i), jumping from a j-labelled node v to $\mathbf{find}(p_j) = p_l$ where $l \neq i$ and backtracking from it, possibly involving several more jumps, eventually reaches back to v. Because v is reachable from the root and because (I1) tells us that every backtracking path from v passes through $\mathbf{find}(p_j) = p_l$, we can trace a backtracking path from v to p_l. Thus, applying the same argument to each jump, we obtain a (path with a) cycle in N, which is a contradiction. □

Lemma 3.5. *Every node is visited at most twice during the Algorithm.*

Proof. Suppose v is a sink node. It is visited once at the start of the corresponding *Backtrack*. If v is a negative atom or a $\parr^-$-node, then it will not be visited again because it has no successor. If v is a $\otimes^-$-node, it may be revisited as a result of Case 3 or Case 6. A visit of v by Case 3 can happen only once as v has only one successor; further the successor is not a $\parr^+$-node, and so, v is never visited as a result of Case 6. Similarly, a visit of v in Case 6 can only happen once and Case 3 cannot happen then.

Next suppose v is not a sink node. Any run of *Visit*(v, i) is called by some *Visit*(u, i) either as a result of a jump from a j-labelled u to some p_l which is a successor of v (Case 6), or where u is a successor of v (Case 3). After that call of *Visit*(u, i) is completed, u has name i by Invariant (I3), and so, v will never be called by *Visit*(u, l) again for any l (even though u itself may be revisited). Since v has at most two successors, it can be visited at most twice during the Algorithm. □

Thus to summarise

Theorem 3.6 Correctness. *Algorithm A is a linear-time algorithm for* EssNet.

4. VERIFICATION USING DOMINATOR TREES

The nodes of an essential net N can be related by the *domination ordering*: we say that node v ***dominates***[5] w if and only if every path from the root to w passes through v. It is not difficult to see that the ordering has a tree structure. The ***dominator tree*** of N is induced by the edges $(idom(v), v)$, where v ranges over the non-root nodes of N and where $idom(v)$ is the *immediate dominator* of v, i.e. $idom(v)$ dominates v, and every other node that dominates v also dominates

[5]Lamarche [1994] calls it the *dependency* ordering, but we prefer to use what seems to us standard terminology in Algorithmics.

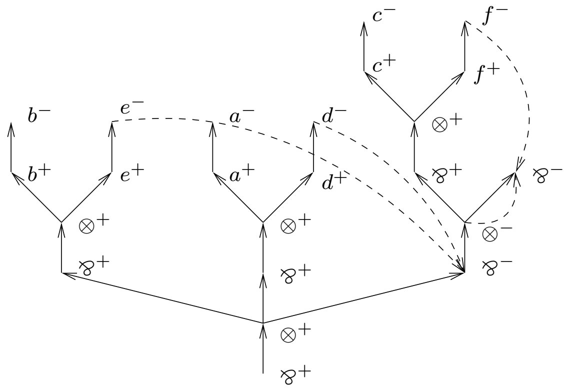


Fig. 4. The dominator tree of the essential net in Figure 2.

idom(v). Plainly the root of the tree is the root of N, and the parent of every other node in the tree is its immediate dominator. The domination ordering and the induced dominator tree can be defined for any directed graph such that every node is reachable from some fixed node. Such graphs are called *flowgraphs* in the literature. (Acyclic essential nets are flowgraphs – see Remark 2.4, though flowgraphs are not assumed to be acyclic.)

Example 4.1. The dominator tree of the essential net in Example 2.2 is given in Figure 4. (The reader should ignore the dashed arrows in the Figure for the time being; they are edges of the essential net, *not* part of the dominator tree.)

4.1 Structure of dominator trees of essential nets

In the dominator tree of an essential net, a $\otimes^+$-node has exactly two positive successors, and a finite number of negative successors, each of which being a $\parr^-$-node. A $\otimes^-$-node has exactly one positive successor and a finite number of negative successors, at most one of which is a $\otimes^-$-node. It is a property of such dominator trees that the parent of a $\parr^-$-node is either a $\otimes^+$-node or a $\otimes^-$-node.

It is easy to see that (the dominator tree of) the *empire* of a positive node of an essential net is exactly the subtree of the dominator tree rooted at that node; this is false in case the node is negative.

We can restate Condition L (for a $\parr^+$-node p) simply as: p dominates its sink. Given the dominator tree of an essential net N, Condition L for all $\parr^+$-nodes can be verified by a depth-first search of its dominator tree. Several linear-time algorithms for constructing dominator trees have very recently been proposed; we give a survey shortly. Using any of these algorithms, we can easily assemble a linear-time algorithm for EssNet, and so, thanks to Theorem 2.7, also a linear-time algorithm for ProofNet.

4.2 Fast algorithms for computing dominator trees

The problem of computing dominator trees (of flowgraphs) has a long and interesting history in control flow analysis. The standard and widely used efficient algorithm is slightly superlinear; it was discovered by Lengauer and Tarjan [1979] after a series of successive improvements by many authors. By using ingenious data structures, Harel [1985] was able to reduce its complexity to linear time; his result was later refined by Alstrup et al. [1999]. Although theoretically important, the later algorithms are not really practical because they depend on highly complex data structures (e.g. Q-heaps due to Fredman and Willard [1994]), which are costly to use.

Recently Buchsbaum et al. [1998] have introduced another linear-time algorithm, which modifies the Lengauer and Tarjan method. Rather than complicated data structures, they combine the use of microtrees and memoization with a linear-time case of the Disjoint Set Union Problem [Gabow and Tarjan 1985] and new observations on a restricted class of path compressions. Tests conducted by Buchsbaum *et al* show that the algorithm[6] is only 10 to 20% slower than the Lengauer-Tarjan algorithm on real flowgraphs of reasonable size and only a few percent slower on very large graphs.

4.3 Constructing dominator trees of essential nets

Here we present a method for constructing dominator trees of essential nets (which we will use later in the sequentialization algorithms in Section 6). First some observations. Fix an essential net N. If w is the only predecessor of v in N, then w must be an immediate dominator of v (i.e. w is the predecessor of v in the dominator tree). If v has two predecessors w_1 and w_2 in N, then in the dominator tree, v is a successor of the infimum (i.e. the *least common ancestor*) of w_1 and w_2, denoted $\mathsf{LCA}(w_1, w_2)$. In the latter case, v is necessarily a $\parr^-$-node. (For example, in the dominator tree of Example 4.1, each of the $\parr^-$-nodes is a successor node of the least common ancestor of the two nodes which are the respective sources of the dashed arrows pointing to the $\parr^-$-node.) These are in fact all the cases that can arise in an essential net. The preceding analysis suggests a linear-time algorithm, which we call Algorithm B, for constructing dominator trees of essential nets. The algorithm uses (linear-time) topological sort (see e.g. [Cormen et al. 1990]).

Each iteration of the for-loop in Algorithm B adds a node to the tree T. Note that the node j in 3.1 (or nodes j_1 and j_2 in 3.2) must already be present in T at the start of each iteration because the nodes are processed in topological order.

The time complexity of Algorithm B depends on how fast one can compute the LCA (least common ancestors) queries. By using a multi-level algorithm due to Gabow [1990], which reduces the problem of computing LCAs to compressed trees, and packs look-up tables into words of memory, one can implement the LCA queries and the additions of nodes as constant-time operations. (The compressed tree of a tree has the same number of nodes as the initial tree, but its depth is not more than log(number of nodes). Some paths from the initial tree (called *heavy*) are preserved, some are not.) We can use Algorithm B as part of a procedure to decide if an essential net is correct. If we do so naïvely, each node will be visited four

[6] Buchsbaum [2000] told us that it had taken his team about 6 months to reduce the constant factor (by studying cache effects) sufficiently to achieve the experimental results; in contrast the Lengauer-Tarjan algorithm can be implemented in a day, and still runs faster on any reasonable flowgraph.

Algorithm B

Input: an acyclic essential net N
Output: the dominator tree T of N

1. Sort the input net topologically, i.e. if there is a path from v_i to v_j, then $i < j$. (Suppose the nodes are enumerated from 1 (the root) to n as a result of the sorting. Henceforth we refer to the nodes by number.)
2. Initialize T to the singleton tree consisting of 1. ($\boldsymbol{link}(w, v)$ makes w a successor node of v in T.)
3. For each $i = 2, \cdots, n$, perform the following operation on T:
   3.1 if j is the only predecessor of i in N then $\boldsymbol{link}(i, j)$
   3.2 else $\boldsymbol{link}(i, \mathsf{LCA}(j_1, j_2))$, where j_1 and j_2 are the two predecessors of i in N.

times: acyclicity check and topological sort of the input net, construction of the dominator tree, and verification of Condition L for each $⅋^+$-node by a depth-first search of the dominator tree. This number can be reduced to one and we show how it can be done in the next section.

5. ONE-PASS VERIFICATION ALGORITHMS

We first give Algorithm C, a linear-time algorithm for EssNet that makes only one pass over the input essential net. By combining it judiciously with the Trip Translation Algorithm, we obtain a linear-time one-pass algorithm for ProofNet, which we call Algorithm D.

5.1 A one-pass algorithm for EssNet

In Algorithm C, $\boldsymbol{link}(w, v)$ and $\mathsf{LCA}(v, w)$ are the constant-time operations due to Gabow [1990]. Note that the Algorithm returns the dominator tree of the input essential net N in case N is correct.

We prove that Algorithm C always terminates and is correct.

Proposition 5.1. *(i) Algorithm C always terminates; if it returns YES, the input essential net is correct.*

(ii) If Algorithm C returns NO during stage 2, some $⅋^+$-node in the input net does not dominate its sink.

(iii) If Algorithm C returns NO during stage 3, the input net is cyclic.

Proof. (i) We first note that the procedure ***visit*** never visits a cycle, if one exists in the input net. By analysing the number of incoming and outgoing edges incident to each of the four types of connective-nodes, one can see that the only way to enter a cycle is via some $⅋^-$-node (which is a node of the cycle). Since the algorithm only visits a $⅋^-$-node if both its predecessors have already been visited, no cycle will ever be entered by ***visit***. Since the next node visited is always a successor of some previously visited node and no node is visited more than once, the Algorithm always terminates. Another consequence is that if all nodes are visited, then the input net is acyclic. Checks made in the last clause of stage 2 ensure that every $⅋^+$-node satisfies Condition L. Thus if the Algorithm returns YES, the input net must be correct.

Algorithm C

Input: an essential net N (whose root is v_0)
Output: YES (T = dominator tree of N) if N is correct, NO otherwise

1. Initialize T to a singleton tree of node v_0.
   (N.B. the operation $\boldsymbol{link}(w, v)$ makes w a successor node of v in T.)
2. Run $\boldsymbol{visit}(v_0)$, where:

   $\boldsymbol{visit}(v) \quad = \quad$ *Cases of* v :
   – positive end of an axiom link ending in v^- $\quad\rightarrow\quad$ $\boldsymbol{link}(v^-, v)$; $\boldsymbol{visit}(v^-)$
   – $\otimes^+$-node with successors v_1 and v_2 $\quad\rightarrow$
   $\qquad\boldsymbol{link}(v_1, v)$; $\boldsymbol{visit}(v_1)$; $\boldsymbol{link}(v_2, v)$; $\boldsymbol{visit}(v_2)$
   – $⅋^+$-node with successor v' $\quad\rightarrow\quad$ $\boldsymbol{link}(v', v)$; $\boldsymbol{visit}(v')$
   – a negative node $\quad\rightarrow$
   $\qquad$if (v is the sink of a $⅋^+$-node w) and (w is absent from T or $\mathsf{LCA}(w, v) \neq w$)
   $\qquad$then return NO
   $\qquad$if v is a $\otimes^-$-node with positive successor v^+ then $\{\boldsymbol{link}(v^+, v)$; $\boldsymbol{visit}(v^+)\ \}$
   $\qquad$if v has a negative successor w then *Cases of* w :
   $\qquad$– $\otimes^-$-node $\quad\rightarrow\quad$ $\boldsymbol{link}(w, v)$; $\boldsymbol{visit}(w)$
   $\qquad$– $⅋^-$-node $\quad\rightarrow$
   $\qquad\quad$if v' (the other predecessor of w) has already been visited then
   $\qquad\qquad\{\boldsymbol{link}(w, \mathsf{LCA}(v, v'))$; $\boldsymbol{visit}(w)\}$
3. If $\boldsymbol{visit}(v_0)$ has visited every node then return YES else return NO.

(*ii*) A node is added to T just before it is visited. If a $⅋^+$-node p dominates its sink $sink(p)$ in the input net, then when the latter is visited, the former must have already been visited and is thus present in T; in addition, we must have $\mathsf{LCA}(p, sink(p)) = p$.
(*iii*) Suppose, for a contradiction, the input net is acyclic. By Remark 2.4, all nodes are reachable from the root. Therefore there is some $⅋^-$-node which has not been visited by ***visit***. Take the one that has the greatest number in some topological ordering (the root is least in the ordering) and call it v. This means that all $⅋^-$-nodes through which v can be reached must have been visited. Hence v must have been visited as well – contradiction. □

5.2 A one-pass algorithm for ProofNet

Our next algorithm, Algorithm D, is a significant refinement of Algorithm C: it takes a proof structure as input, and performs *at once* the trip translation[7] (assigning polarities to nodes) and the requisite correctness checks of an essential net. Algorithm C performs the Condition L check whenever a sink is visited. Here we do so by asking whether the node below the alleged sink has already been visited (if it has, it must have been visited with positive polarity). If not, the $⅋$-node below must then be given negative polarity. However, because polarities are not stored anywhere, when a $⅋$-node is visited with positive polarity, we need to ascertain

[7]In contrast to the Trip Translation Algorithm, the DR-switching corresponding to the actual traversal by ***visit2*** is determined during runtime. However, if Algorithm D returns YES, we know that a switching σ such that $\mathrm{Gr}(\Gamma, \sigma)$ is acyclic and connected does exist, namely, the one actually "used" by the traversal.

Algorithm D

Input: an MLL proof structure Γ
Output: (YES, T = dominator tree of $\mathcal{E}(\Gamma, B, \sigma)$ for some B and σ) if Γ is a proof net, NO otherwise

1. Initialize T to a singleton tree of node v_0 corresponding to some B in Γ (the choice is irrelevant). (N.B. the operation $\boldsymbol{link}(w, v)$ makes w a successor node of v in T.)
2. Run $\boldsymbol{visit2}(+, v_0)$, where:
   $\boldsymbol{visit2}(p : \mathit{Polarity}, v : \mathit{Node}) \quad =$
   if (v already visited) then return NO **;**
   Cases of (p, v) :
   – ($+$, end of axiom link ending in v') $\quad\rightarrow\quad$ $\boldsymbol{link}(v', v)$ **;** $\boldsymbol{visit2}(-, v')$
   – ($+$, $\otimes$-node), where v_1 and v_2 are the nodes above v $\quad\rightarrow$
     $\boldsymbol{link}(v_1, v)$ **;** $\boldsymbol{visit2}(+, v_1)$ **;** $\boldsymbol{link}(v_2, v)$ **;** $\boldsymbol{visit2}(+, v_2)$
   – ($+$, $⅋$-node) $\quad\rightarrow$
     if either of the two nodes above v has been visited then return NO
     else $\{\boldsymbol{link}(v', v)$ **;** $\boldsymbol{visit2}(+, v')\}$
     where v' is either of the two
   – $(-, v)$ $\quad\rightarrow$
     if v is a $\otimes$-node then $\{\boldsymbol{link}(v', v)$ **;** $\boldsymbol{visit2}(+, v')\ \}$
     where v' is the node above v other than the one just visited

     if v has a node w below then *Cases of* w :
     – $\otimes$-node $\quad\rightarrow\quad$ $\boldsymbol{link}(w, v)$ **;** $\boldsymbol{visit2}(-, w)$
     – $⅋$-node $\quad\rightarrow$
       if (w already visited) then if $\mathsf{LCA}(w, v) \neq w$ then return NO **;**

       if (w has not been visited and the other node v' above w has been visited) then $\{\boldsymbol{link}(w, \mathsf{LCA}(v, v'))$ **;** $\boldsymbol{visit2}(-, w)\ \}$
3. If every node has been visited by $\boldsymbol{visit2}(+, v_0)$ then return YES else return NO

that its sink has not been visited before. As before, the operations $\boldsymbol{link}(w, v)$ and $\mathsf{LCA}(v, w)$ are constant time *a la* Gabow. As the Algorithm makes only one pass over the input proof structure (visiting each node at most once), it is linear time.

Proposition 5.2. *Algorithm D always terminates and is correct.*

Proof. The first command of $\boldsymbol{visit2}(p, v)$, which checks if v has already been visited, guarantees termination of the Algorithm. If the Algorithm returns YES, each node has been visited. By referring to the Trip Translation Algorithm, note that the polarities with which the nodes have been visited are in accord with the essential net $\mathcal{E}(\Gamma, B, \sigma)$; also the actions of Algorithm D corresponding to each type of polarized nodes are exactly those in Algorithm C (as explained in the para. before the Algorithm). By the correctness of Algorithm C, the essential net $\mathcal{E}(\Gamma, B, \sigma)$ is correct, and so, by Theorem 2.7 (*ii*), Γ is a proof net.

There are four different reasons why the Algorithm may return NO. We go through the four "return NO" clauses in the textual order. The first two are caused by a cycle in the input DR-graph $\mathrm{Gr}(\Gamma, \sigma)$; the third corresponds to the $⅋^+$-node (of the essential net $\mathcal{E}(\Gamma, B, \sigma)$) in question violating Condition L. The last is caused either by a cycle in $\mathrm{Gr}(\Gamma, \sigma)$ or its disconnectedness. In all these cases, the input proof structure is not a proof net. □

Remark 5.3. Algorithm D does not keep a record of the polarities assigned to nodes. Alternatively, when a node is visited, we could record the polarity and assign polarities to adjacent nodes in the syntactic tree (as in the Trip Translation Algorithm). When visiting a node with polarity p, we would then have to check that it has not been assigned $\neg p$ before. We reject this approach because each node would be examined several times and the code would be longer. In contrast, Algorithm D is very concise, though it is possibly slightly less clear.

6. LINEAR-TIME SEQUENTIALIZATION ALGORITHMS

In this Section, we first give two algorithms for sequentializing essential nets. The first, called Decomposition Algorithm, returns a sequence of nodes of the net which encodes a derivation of the sequent in question; the second, Algorithm E, returns a derivation of the sequent. We then show that a linear-time algorithm for sequentializing proof nets can be obtained as a corollary (essentially by running the Trip Translation Algorithm on the input, followed by Algorithm E, and ending with a modicum of "bureaucratic" operations on derivations).

6.1 Sequentializing essential nets

Take a correct essential net N and fix a linear ordering $>$ of its nodes by a topological sort. By an *ordered* dominator tree, we mean a dominator tree such that for each $\otimes^+$-node and each $\otimes^-$-node v, the negative successors of v, if any, are arranged in ascending order. For example, the dominator trees returned by Algorithm B are ordered. (In Algorithm B, since the nodes are already added to the tree in topological order, we can "conserve" the order by e.g. adding negative nodes to a queue associated with each $\otimes$-node.)

Fix a correct essential net N, and let T_N be an ordered dominator tree of N. First we give an informal account of how our algorithm will work. Sequentializing N amounts to decomposing it by successive removals of nodes (together with the associated edges) subject to the following conditions:

(i) when a node v is to be deleted, all nodes 'below' it in N must have already been eliminated;

(ii) a^+ can only be removed in conjunction with a^-, i.e. an axiom link becomes eliminable only after all nodes below a^+ and a^- have been eliminated;

(iii) when a $\otimes^+$- or $\otimes^-$-node is removed, its removal must split the net to which the node belongs.

During the process of decomposition the original net is being replaced by a collection of smaller ones that comprise all the nodes that have not been eliminated yet. Therefore, if a node has not been removed yet, there will be a unique net of which it is a node (cf. (iii) above). A sequentialization of a net can thus be specified by an order on the nodes. Our sequentialization algorithm implies that one such order can be computed in linear time by a depth-first search of T_N. The traversal is guided by the order on the children of nodes: first all negative children of a node are visited in reverse topological order, then the node itself can be eliminated and, finally, the procedure visits the positive children in any order. The procedure is outlined in Decomposition Algorithm. To start off, we call $\boldsymbol{visit}(r^+)$ for the root

Decomposition Algorithm

Input: an ordered dominator tree T_N of a correct essential net N
Output: a 'sequentialization' of N given by a sequence of nodes

- Run $\boldsymbol{visit}(r^+)$, where r^+ is the root of N:

$\boldsymbol{visit}(v : Vertex)$

1. call $\boldsymbol{visit}(n)$ for each negative T_N-child n of v in *reverse* topological order
2. *ELIMINATE* v
3. call $\boldsymbol{visit}(p)$ for each positive T_N-child p of v

r^+ of the net. The order in which *ELIMINATE* 'signals' are given will be the order in which nodes can be removed from the net. Observe that *ELIMINATE* a^- will always be followed by *ELIMINATE* a^+. This is to be treated as a cue to remove the axiom link. Pictorially, the algorithm sweeps the tree from right to left. If we apply the algorithm to the net from Figure 2, (whose dominator tree is shown in Figure 4) we obtain the following sequence of nodes:

$$\parr^+ \parr^- \otimes^- \parr^+ \otimes^+ f^- f^+ c^- c^+ \parr^- \otimes^+ \parr^+ \otimes^+ d^- d^+ a^- a^+ \parr^+ \otimes^+ e^- e^+ b^- b^+$$

The reader should have no difficulty associating the nodes with their occurrences in Figure 2. Figure 5 contains another example. The correctness of the Decomposition Algorithm will follow from the correctness of the algorithm presented in the next section.

6.2 Sequentializing essential nets in linear time

Next we give a linear-time algorithm, called Algorithm E, for sequentializing essential nets. It takes a correct essential net $N = \mathcal{E}(\Gamma \vdash A)$ as input and returns a derivation of the sequent $\Gamma \vdash A$, and is defined in terms of the procedure ***deriv***. Note that it will process negative nodes in topological order, as the objective here is to construct the corresponding derivation.

Algorithm E

Input: a correct essential net $N = \mathcal{E}(\Gamma \vdash A)$
Output: a derivation of $\Gamma \vdash A$

1. Run Algorithm B with input N to produce an ordered dominator tree T_N
2. Run ***deriv*** v_0 where v_0 is the root of T_N.

We write *PosNode* (respectively *NegNode*) for the type of positive nodes (respectively negative nodes) of T_N, *Derivation* for the type of IMLL derivations, and [*NegNode*] for the type of lists of *NegNode*. We define functions

$$\boldsymbol{deriv} : PosNode \to Derivation$$
$$\boldsymbol{derivNeg} : Derivation \to [NegNode] \to Derivation$$

by recursion, using a standard functional programming notation. By $succ_{T_N}(v)$ we mean the successor of v in T_N.

$$\boldsymbol{deriv}\ v \quad = \quad \textit{Cases of } v :$$

1. positive atom a^+ such that $a^- = succ_{T_N}(v)$ is a leaf of T_N $\quad\rightarrow\quad$ "$a \vdash a$"
2. positive atom a^+ such that $a^- = succ_{T_N}(v)$ is not terminal in T_N $\quad\rightarrow$
   $\boldsymbol{derivNeg}$ "$a \vdash a$" $[\,w\,]$
   where $w = succ_{T_N}(succ_{T_N}(v))$
3. $\otimes^+$-node $\quad\rightarrow\quad$ $\boldsymbol{derivNeg}\ d\ [\,w_1, \cdots, w_n\,]$
   where: d is the derivation formed from $\boldsymbol{deriv}\ v_1$ and $\boldsymbol{deriv}\ v_2$ by the rule **($\otimes$-r)** where v_1 and v_2 are the positive successors, and $w_1, \cdots, w_n$ are the negative successors (such that $w_1 < w_2 < \cdots < w_n$ with respect to the given topological order) of v in T_N
4. $⅋^+$-node $\quad\rightarrow\quad$ d
   where d is formed from $\boldsymbol{deriv}\ succ_{T_N}(v)$ by the rule **($\multimap$-r)** and the formula eliminated is given by the sink of v.

The definition of $\boldsymbol{derivNeg}$ has two clauses.

$$\boldsymbol{derivNeg}\ d\ [\,]\ =\ d$$

$$\boldsymbol{derivNeg}\ d\ (w : ws)\ =\ \textit{Cases of } w :$$

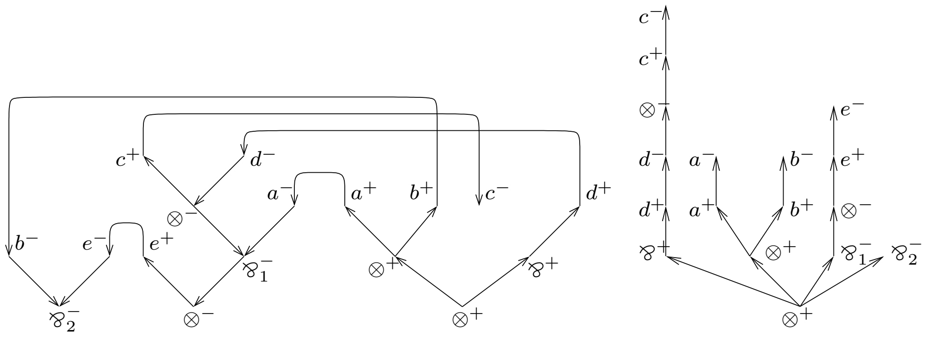
1. $⅋^-$-node and terminal in T_N $\quad\rightarrow\quad$ $\boldsymbol{derivNeg}\ d'\ ws$
   where: the derivation d' is constructed from d by the rule **($\otimes$-l)** where the two formulas, which form the tensor formula introduced on the left, are given by the predecessors of w in N
2. $⅋^-$-node which is not terminal in T_N $\quad\rightarrow\quad$ $\boldsymbol{derivNeg}\ d'\ (w' : ws)$
   where d' is constructed as in the previous case and $w' = succ_{T_N}(w)$
3. $\otimes^-$-node $\quad\rightarrow\quad$ $\boldsymbol{derivNeg}\ d'\ ([w_1, \cdots, w_n] +\!\!+ ws)$
   where: d' is the derivation formed from $\boldsymbol{deriv}\ v$ and d by the rule **($\multimap$-l)** (so that the respective sequents proved by d' and d have the same conclusion and where the B of the formula $A \multimap B$ introduced on the left is given by the predecessor of w in T_N), v is the positive successor and $w_1, \cdots, w_n$ are the negative successors (in ascending topological order) of w in T_N.

The rest of this subsection is devoted to a proof of correctness of Algorithm E. We begin with some technical preliminaries on essential nets and their dominator trees.

Definition 6.1. An ***abstract essential net*** is a vertex-labelled finite directed graph given by the five-tuple $\langle V, E, v_0, \textit{label}, \textit{sink}\rangle$ where $\langle V, E\rangle$ is an unlabelled graph (V is the set of nodes and E the set of edges) such that:

(i) The map *label* gives each node a label: in case the label is a positive atom or $⅋^+$ or $\otimes^+$, we call the node *positive*; in case the label is a negative atom or $⅋^-$ or $\otimes^-$, we call the node *negative*. If a node has label l, we call it an l-node.

(ii) The number and polarity of nodes incident to each type of nodes are governed by the *incidence rules*:



$$\parr_2^- \otimes^- e^- e^+ \parr_1^- \otimes^+ \otimes^+ a^- a^+ b^- b^+ \parr^+ \otimes^- c^- c^+ d^- d^+$$

Fig. 5. The essential net of $b \otimes e, e \multimap (c \multimap d) \otimes a \vdash (a \otimes b) \otimes (c \multimap d)$, its dominator tree and an order in which the net could be decomposed.

Types of nodes	*Number of successors*	*Number of predecessors*
atom a^+	1, must be a^-	0 or 1 positive
$\parr^+$	1 positive	0 or 1 positive
$\otimes^+$	2 positive	0 or 1 positive
atom a^- (say)	0 or 1 negative	1, must be a^+
$\parr^-$	0 or 1 negative	2 negative
$\otimes^-$	1 positive, and 0 or 1 negative	1 negative

(iii) There is an injective map *sink* from $\parr^+$-nodes to negative nodes that have no negative successors. We call *sink*(p) the *sink* of the $\parr^+$-node p.

(iv) v_0 is a distinguished positive node called *root*. Each positive node, except the root, has exactly one predecessor which must be positive.

The following proposition clarifies the sense in which essential nets and abstract essential nets are the same. Henceforth we shall not bother to distinguish them.

Proposition 6.2. *(i) Every essential net is an abstract essential net.*

(ii) Every abstract essential net is the essential net of a linearly balanced IMLL sequent.

Proof. It is straightforward to verify (i). (ii) can be shown by defining a procedure that recursively assigns formulas to nodes of an abstract essential net. □

Remark 6.3. Suppose $C_1, \cdots, C_n \vdash A$ is the sequent corresponding to an essential net N, there is a one-one correspondence between the premises $C_1, \cdots, C_n$ and the negative nodes of N that have no negative successors *and* are not sinks.

Given an essential net $N = \langle V, E, v_0, \mathit{label}, \mathit{sink} \rangle$, we define the associated ***dominator structure*** D_N to be the six-tuple $\langle V, E_d, E_a, v_0, \mathit{label}, \mathit{sink} \rangle$ where $\langle V, E_d \rangle$

is the dominator tree of the directed graph $\langle V, E \rangle$, and E_a, the set of *auxiliary edges* of D_N, is defined to be $\{ (u,v) \in E : v \text{ is a } \wp^-\text{-node} \}$. We shall refer to $\langle V, E_d \rangle$ as the *underlying tree* of the dominator structure. For example, in the tree in Example 4.1, the dashed arrows are precisely the auxiliary edges. (In general we need the auxiliary edges in order to reconstruct N from D_N: indeed the edge-set E of N is $E_a \cup \{ (u,v) \in E_d : v \text{ is not a } \wp^-\text{-node} \}$. We can always reconstruct the essential net from its associated dominator structure by performing the following operation on each $\wp^-$-node p in turn: cut off the subtree rooted at p, and graft it back by the auxiliary edges (u,p) and (v,p).)

Fix an essential net N and let $D = D_N$ be the associated dominator structure $\langle V, E_d, E_a, v_0, \mathit{label}, \mathit{sink} \rangle$. In the following we shall consider subtrees $T_0 = \langle V_0, E_0 \rangle$ of the underlying tree $\langle V, E_d \rangle$ of D. We are concerned with whether T_0 is the underlying tree of a dominator structure associated with an essential net. Precisely, we ask whether the five-tuple derived from T_0

$$N(T_0) \;=\; \langle V_0, \{ (u,v) \in E_0 : v \text{ is not a } \wp^-\text{-node} \} \cup (E_0)_a, v, \mathit{label} \restriction V_0, \mathit{sink} \restriction V_0 \rangle$$

where $(E_0)_a$ is the *inherited* auxiliary edge-set i.e. $E_a \cap (V_0 \times V_0)$, is an essential net, for an appropriate $v \in V_0$. If it is, we shall refer to T_0 simply as a *dominator substructure* of D_N.

LEMMA 6.4. *Using the same notation as before, for any positive node v of V, the subtree of the tree $\langle V, E_d \rangle$ rooted at v, denoted* $\mathsf{subtree}(v)$*, is a dominator substructure of D (in fact it is the dominator structure associated with the empire of v). Note however that* $\mathsf{subtree}(w)$ *is not a dominator substructure if w is a negative node.*

PROOF. To show that $N(\mathsf{subtree}(v))$ satisfies the incidence rules of Definition 6.1, we need only check that each $\wp^-$-node q in it has two negative predecessors. For any such q, its D-predecessor is in $\mathsf{subtree}(v)$; hence its N-predecessors q_1, q_2 (say) are also in $\mathsf{subtree}(v)$, and so, $(q_1,q), (q_2,q)$ are included among the inherited auxiliary edges of $N(\mathsf{subtree}(v))$. □

Let T_0 be a dominator substructure of D. Take a negative non-atomic node w of D which is not in T_0. If the N-predecessor(s) of w are in T_0 (then the D-predecessor of w, $\mathit{pred}_D(w)$, is also in T_0), we define $\mathsf{graft}(T_0, w)$ to be the subtree obtained from T_0 by "grafting" $\mathsf{subtree}(w)$ onto it at $\mathit{pred}_D(w)$. It follows from the definition that $\mathsf{graft}(T_0, w)$ is a subtree of $\langle V, E_d \rangle$. In fact

LEMMA 6.5. $\mathsf{graft}(T_0, w)$ *is a dominator substructure of D.*

PROOF. w is either a $\otimes^-$-node or a $\wp^-$-node. In either case, we observe that in $N(\mathsf{graft}(T_0, w))$, the requisite negative predecessor(s) are present. The incidence rules are easily seen to be satisfied by the other nodes. □

Take a dominator substructure T_0 of D. We say that T_0 is *extensible* over a non-empty list $[\, w_1, \cdots, w_n \,]$ of negative non-atomic nodes just in case for each $1 \leqslant i \leqslant n-1$,

$$\mathsf{graft}(\cdots \mathsf{graft}(\mathsf{graft}(T_0, w_1), w_2) \cdots, w_i)$$

which by abuse of notation we simply write as $\mathsf{graft}(T_0, [w_1, \cdots, w_i])$, is a dominator substructure of D, and the N-predecessor(s) of w_{i+1} are in it. It follows that $\mathsf{graft}(T_0, [w_1, \cdots, w_n])$ is a dominator substructure of D.

For any IMLL derivation d, we write D_d for the dominator structure of the essential net of the sequent proved by d, and we shall say, by abuse of language, that d is a derivation of the dominator substructure D_d.

THEOREM 6.6. *We use the same notation as before.*

(P) For any positive node v, ($\boldsymbol{deriv}$ v) is a derivation of (the dominator structure defined by) $\mathsf{subtree}(v)$.

(N) For any derivation d whose dominator structure D_d is a substructure of D, for any non-empty list ws of negative non-atomic nodes, if D_d is extensible over ws, then ($\boldsymbol{derivNeg}$ d ws) is a derivation of the dominator substructure $\mathsf{graft}(D_d, ws)$.

PROOF. We use induction on the respective measures of v and of (d, ws). The measure of a node v (whether positive or negative) of D is defined to be the number of nodes of $\mathsf{subtree}(v)$. The measure of $(d, [w_1, \cdots, w_n])$ is the sum of the respective measures of $w_1, \cdots, w_n$.

For (P), we shall just consider the harder cases of 3 and 4 in the definition of $\boldsymbol{deriv}$ v. In case 3 where v is a $\otimes^+$-node, by the induction hypothesis (P), ($\boldsymbol{deriv}$ v_1) and ($\boldsymbol{deriv}$ v_2) are derivations of $\mathsf{subtree}(v_1)$ and $\mathsf{subtree}(v_2)$ respectively, and so d as defined is a derivation of the subtree of D rooted at v but *less* $w_1, \cdots, w_n$ and all their respective descendants. It follows from the topological ordering $w_1 < \cdots < w_n$ that D_d is extensible over $ws = [w_1, \cdots, w_n]$ (For if not, suppose an N-predecessor of w_{i+1} (say) is not in $\mathsf{graft}(D_d, [w_1, \cdots, w_i])$, but in (say) $\mathsf{subtree}(w_j)$ for some $j > i$. Then we have an N-path from w_j to w_i and so $w_j < w_i$, contradicting the ordering.) By the induction hypothesis (N), ($\boldsymbol{derivNeg}$ d ws) is a derivation of $\mathsf{graft}(D_d, ws)$, which is exactly $\mathsf{subtree}(v)$. For Case 4 where v is a $⅋^+$-node, since D is the dominator structure of a correct essential net, the sink of v is in $\mathsf{subtree}(succ_D(v))$. Besides, the sink has no negative successor in the essential net associated with $\mathsf{subtree}(succ_D(v))$, and so, by Remark 6.3, it determines a premise in the sequent proved by ($\boldsymbol{deriv}$ $succ_D(v)$).

In the case of (N), we shall just consider Case 3 where w is a $\otimes^-$-node, assuming that the dominator substructure D_d is extensible over the list $(w : ws)$. Let N^- be the essential net of the sequent proved by d. In N^-, the D-predecessor of w is a negative node that has no negative successor (in N^-) and is not a sink. By Remark 6.3, it determines a premise in the sequent proved by d. By the induction hypothesis (P), ($\boldsymbol{deriv}$ v) a derivation of $\mathsf{subtree}(v)$ where v is the positive successor of the $\otimes^-$-node w. Thus the construction of d' as described is well-defined. Now it follows from the assumption that $D_{d'}$ is extensible over $[w_1, \cdots, w_n]{+\!\!+}ws$. By the induction hypothesis (N), $\boldsymbol{derivNeg}$ d' $([w_1, \cdots, w_n]{+\!\!+}ws)$ is a derivation of $\mathsf{graft}(D_{d'}, [w_1, \cdots, w_n]{+\!\!+}ws)$, which is just $\mathsf{graft}(D_d, w : ws)$, as desired. Termination of ($\boldsymbol{deriv}$ v) and ($\boldsymbol{derivNeg}$ d ws) (assuming that D_d is extensible over ws) follows from the fact that at each step, the relevant measure strictly decreases. □

6.3 Sequentializing proof nets in linear time

As a corollary, we obtain a linear-time algorithm, Algorithm F, for sequentializing proof nets. The permutation operations in stage 4 of the Algorithm are formal

Algorithm F

Input: a proof net Γ
Output: a derivation of $\vdash \Gamma$

1. Run the Trip Translation Algorithm with Γ and any DR-switching and any conclusion as input.
2. Run Algorithm E on the output essential net of 1.
3. Apply the linear-time transformation of Proposition 2.1 to the output derivation of 2.
4. Perform appropriate permutation operations to obtain a derivation of Γ.

details. E.g. we may have to swap the two ends of an axiom link (this is because the polarisation in stage 1 may assign + to $a^{\perp}$). In addition, sometimes it will be necessary to change $A \parr B$ to $B \parr A$. The reason is that the translation from polarized MLL into IMLL takes $B^{+} \parr^{+} A^{-}$ to $A \multimap B$ but the translation back from IMLL to MLL gives $A^{\perp} \parr B$. The crucial point is that the order by which connectives are introduced in the output derivation of Algorithm E gives enough information for the reconstruction of a derivation of the (original) input proof net.

ACKNOWLEDGMENTS

We are grateful to François Lamarche, Gianluigi Bellin and Stefano Guerrini for helpful discussions on Linear Logic, and to Adam Buchsbaum and Bob Tarjan for advice on dominator trees. This work was partially funded by EU TMR projects AppSem and LINEAR and EPSRC project GR/L27787. Andrzej Murawski's research was supported by a University of Oxford Scatcherd European Scholarship and a British Overseas Research Students Award 98032135.

REFERENCES

Alstrup, S., Harel, D., Lauridsen, P., and Thorup, M. 1999. Dominators in linear time. *SIAM J. Comput. 28 (6)*, 2117–2132.

Bellin, G. and Scott, P. J. 1994. Remarks on the π-calculus and linear logic. *Theor. Comput. Sci. 135*, 11–65.

Buchsbaum, A. 2000. Private email communication.

Buchsbaum, A., Kaplan, H., Rogers, A., and Westbrook, J. 1998. A new, simpler linear-time dominators algorithm. *ACM Trans. Program. Lang. Syst. 6*, 1265–1296.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press.

Danos, V. 1990. La logique linéaire appliquée à l'étude de divers processus de normalisation et principalement du λ-calcul. Ph.D. thesis, Université de Paris.

Danos, V. and Regnier, L. 1989. The structures of multiplicatives. *Arch. Math. Logic 28*, 181–203.

Fredman, M. and Willard, D. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci. 48*, 533–551.

Gabow, H. 1990. Data structure for weighted matching and nearest common ancestors with linking. In *1st Annual ACM-SIAM Symp. on Discrete Algorithms*, 434–443.

Gabow, H. N. and Tarjan, R. E. 1985. A linear-time algorithm for a special case of disjoint set union. *J. of Comput. Syst. Sci. 30*, 209–221.

Girard, J.-Y. 1987. Linear logic. *Theor. Comput. Sci. 50*, 1–102.

Guerrini, S. 1999. Correctness of multiplicative proof nets is linear. In *14th Annual IEEE Symp. on Logic in Computer Science*. IEEE Computer Science Society, 454–463.

Harel, D. 1985. A linear time algorithm for finding dominators in flow graphs and related problems. In *17th Annual ACM Symp. on Theory of Computing*, 185–194.

Lamarche, F. 1994. Proof nets for intuitionistic linear logic 1: Essential nets. Preprint ftp-able from *Hypatia*.

Lengauer, T. and Tarjan, R. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 121–141.

Murawski, A. S. and Ong, C.-H. L. 2000a. A linear-time algorithm for verifying MLL proof nets via essential nets. In *Proceedings of the Oxford-Microsoft Symp. in Celebration of the Work of C. A. R. Hoare*, Davis, Roscoe, and Woodcock, Eds. Palgrave. Cornerstones in Computing Series, 289–302.

Murawski, A. S. and Ong, C.-H. L. 2000b. Dominator trees and fast verification of proof nets. In *15th IEEE Symp. on Logic in Computer Science*. IEEE Computer Society Press, 181–191.

Murawski, A. S. and Ong, C.-H. L. 2003. Exhausting Strategies, Joker Games and Full Completeness for IMLL with Unit. *Theor. Comput. Sci. 294(1/2)*, 269–305.

Received December 2003; revised July 2004; accepted July 2004