Dominator Trees and Fast Verification of Proof Nets

A. S. Murawski* C.-H. L. Ong†

Oxford University Computing Laboratory

{andrzej,Luke.Ong}@comlab.ox.ac.uk

Abstract

We consider the following decision problems:

> PROOFNET *: Given a multiplicative linear logic (MLL) proof structure, is it a proof net?*
>
> ESSNET *: Given an essential net (of an intuitionistic MLL sequent), is it correct?*

In this paper we show that linear-time algorithms for ESSNET *can be obtained by constructing the dominator tree of the input essential net. 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 these linear-time algorithms can be optimized to simple one-pass algorithms – each node of the input structure is visited at most once. As another application of dominator trees, we obtain linear-time algorithms for* sequentializing *proof nets (i.e. given a proof net, find a derivation for the underlying MLL sequent) and essential nets.*

1 Introduction

By MLL we mean the multiplicative $(\otimes, \parr, -^{\perp})$-fragment of linear logic due to Girard [11]. We are concerned with algorithms for the decision problem:

> PROOFNET: Given an MLL proof structure, is it a proof net?

In last year's LICS, Guerrini [12] gave the first linear-time algorithm for PROOFNET, which was an important and surprising discovery. His approach was based on a reformulation of Danos' contractibility criterion [6], but expressed as a kind of unification problem; the algorithm uses a special linear-time case of the Disjoint Set Union Problem [10]. In this paper, we take a different 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 has proposed a variant system of proof nets, called *essential nets* [14]. 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 $a, b, \cdots$ are 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 a linearly balanced[1] 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 simple linear-time algorithm for the decision problem:

> ESSNET: Given an essential net (of a linearly balanced IMLL sequent), is it correct?

Our approach uses a linear-time algorithm for constructing the dominator tree of a flowgraph; several such algorithms have recently been obtained by various groups [1, 5, 9]. A *flowgraph* is just a directed graph such that every node of the graph is reachable from a fixed start node. 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 start node 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). We can now easily assemble a linear-time algorithm for ESSNET: Given an essential net, first check

* On leave from Nicholas Copernicus University, Toruń, Poland.

† www.comlab.ox.ac.uk/oucl/work/luke.ong/

[1] I.e. every atom that occurs in the sequent does so exactly twice, once positively and once negatively.

acyclicity, then verify Condition L by a depth-first search of the dominator tree of the input net; the tree can be constructed by any of the recently discovered fast algorithms. Details of the approach are in Section 3.

A pleasing consequence is that we can then 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 [2, 3], 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 algorithm thus obtained, though linear time, makes several passes over the input proof structure. In Section 4, we show that the respective linear-time algorithms for ESSNET and PROOFNET 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 B and C 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 5, by a method of systematic decomposition (of essential nets), we obtain linear-time algorithms, Algorithms D and E respectively, for *sequentializing* essential nets and proof nets.

We have recently proposed in [16] a linear-time algorithm for ESSNET which verifies Condition L directly 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 [10] for efficient bookkeeping[3], reducing its asymptotic complexity to linear time. The approach in this paper is quite different and simpler, which we regard as an improvement.

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

[3]Gabow and Tarjan's algorithm [10] is used here for a different purpose from the application in [12].

2 Proof nets, essential nets and linear-time reduction

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[4] as follows: we let $\Gamma, \Delta, \cdots$ range over finite sequences of formulas:

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

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

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 [7] 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 a 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 [11] has been simplified by Danos and Regnier [7] 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 $\vdash \Gamma$.

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

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:

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

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

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

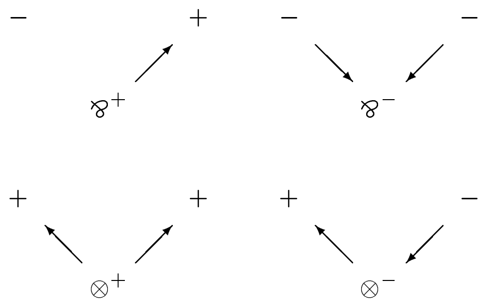
By a simple induction over the rules of the IMLL Sequent Calculus, we have the following:

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 function 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 variant MLL formula $\ulcorner A \urcorner$ which is constructed from *polarized* atoms (namely $a^+, b^+, c^+, \cdots, a^-, b^-, c^-, \cdots$) and polarized connectives (namely $\otimes^+, \otimes^-, \parr^+$ and $\parr^-$) by the following rules: $\ulcorner a \urcorner = a^+$, $\llcorner a \lrcorner = a^-$ and

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

The ***essential net*** [14] 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 by 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***.

Figure 1. The directional rules

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 Table 1. (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, and a $\parr^+$-node of $\mathcal{E}(\vdash \ (a \multimap b) \otimes (b \multimap a))$ violates Condition L, but the essential net in Example 2.2 is correct. An important result in [14] is a characterisation 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. We give a new proof of the Theorem in [18]. □

Remark 2.4. In this paper, essential nets are by definition directed graphs of a certain kind constructed from *linearly balanced* IMLL sequents. For such nets (but not in general – consider $\mathcal{E}(\vdash a \multimap b)$), it is straightforward to prove that the *reachability condition*, i.e. every node is reachable from the root, is a consequence of acyclicity.

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. (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).

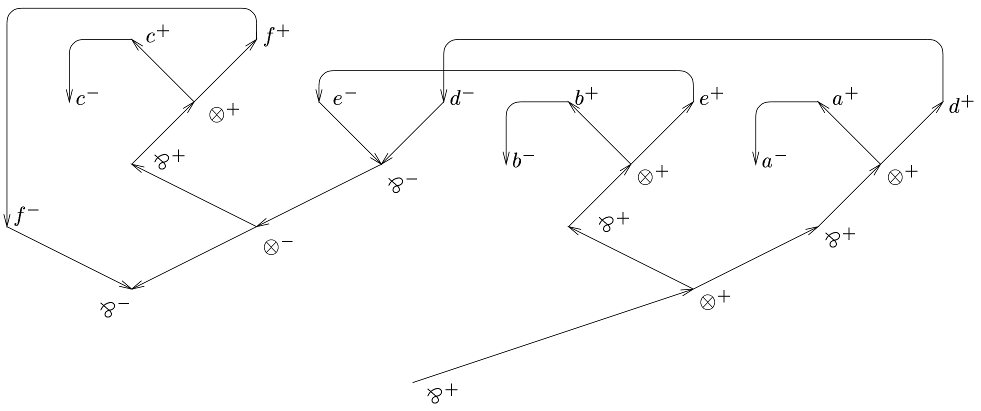


Table 1. The essential net of $\vdash\ f \otimes ((c \multimap (c \otimes f)) \multimap (e \otimes d)) \quad \multimap \quad (b \multimap (b \otimes e)) \otimes (a \multimap (a \otimes d))$**.**

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 $⅋$-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 (so that e.g. $a^\perp$ could be assigned $+$ and hence appear as a^+ in the output net). E.g. consider the case where the input sequent is $\Vdash a ⅋ a^\perp$ 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 terminates and returns $\mathcal{E}(\Gamma, B, \sigma)$ which is the essential net of some IMLL sequent $C_1, \cdots, C_n \vdash A$ such that $\Vdash \Gamma$ is MLL-provable iff $\Vdash \underline{C_1}, \cdots, \underline{C_n}, \overline{A}$ is. Thus if Γ is a proof net, $\mathcal{E}(\Gamma, B, \sigma)$ is a correct essential net.*

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 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 $\Vdash (a^\perp ⅋ b^\perp) ⅋ (a ⅋ b)$ and $\Vdash (a^\perp \otimes b^\perp) ⅋ (a \otimes b)$ respectively.
(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 from Theorem 2.5. For (ii), suppose for some choice B of conclusion, and some DR-switching σ for Γ, the directed graph $\mathcal{E}(\Gamma, B, \sigma)$ is the essential net of $C_1, \cdots, C_n \vdash A$. If the essential net $\mathcal{E}(\Gamma, B, \sigma)$ is correct,

Algorithm *Trip Translation*

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

1. Initialize T to the singleton tree consisting of the node v_0 corresponding to B. (N.B. ***assign***(p, v) assigns polarity p to node v of T, and ***link***(w, v) makes w a successor node of v in T.)

2. Run ***polarize***$(+, v_0)$, where:

   polarize$(p : \mathit{Polarity}, v : \mathit{Node}) \quad =$

   if (v already visited) then return NO **;** ***assign***(p, v) **;**

   Cases of (p, v) :

   – $(+,$ end of axiom link ending in $v')$ $\quad\rightarrow\quad$ ***link***(v', v) **;** ***polarize***$(-, v')$
   – $(+, \otimes$-node$)$ where v_1 and v_2 are the nodes above v $\quad\rightarrow\quad$ ***link***(v_1, v) **;** ***polarize***$(+, v_1)$ **;** ***link***(v_2, v) **;** ***polarize***$(+, v_2)$
   – $(+, \parr$-node$)$ $\quad\rightarrow\quad$ ***link***(v', v) **;** ***polarize***$(+, v')$, where v' is the node above v indicated by $\sigma(v)$
   – $(-, v)$ $\quad\rightarrow$

     if v is a $\otimes$-node then ***link***(v', v) **;** ***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 $\quad\rightarrow\quad$ ***link***(w, v) **;** ***polarize***$(-, w)$
     – $\parr$-node $\quad\rightarrow$

       if (w already visited) then if w is assigned "$-$" then ***link***(w, v)
       else ***link***(w, v) **;** if ((w, v) same side as $\sigma(w)$) then ***polarize***$(-, w)$

3. If every node has been visited by ***polarize***$(+, v_0)$ then return YES else return NO.

by Theorem 2.3, $C_1, \cdots, C_n \vdash A$ is IMLL-provable. Since $\Vdash \Gamma$ is MLL-provable iff $\Vdash \underline{C_1}, \cdots, \underline{C_n}, \overline{A}$ is, by Proposition 2.1, we have Γ is a proof net. As each node is visited by ***polarize*** at most once, the Trip Translation Algorithm is linear-time. $\square$

3 Simple linear-time verification algorithms

Recall that an essential net is correct just in case it is acyclic and every $\parr^+$-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.

A key idea is that 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 easy to see that the ordering has a tree structure. The ***dominator tree*** of N is induced by the edges $(\mathit{idom}(v), v)$, where v ranges over the non-root nodes of N and where $\mathit{idom}(v)$ is the *immediate dominator* of v i.e. $\mathit{idom}(v)$ dominates v, and every other node that dominates v also dominates $\mathit{idom}(v)$. 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. We stress that the domination ordering and the induced dominator tree can be defined for any directed graph such that every node is reachable from some fixed start 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 3.1. The dominator tree of the essential net in Example 2.2 is given in Figure 2. (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.)

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 $\otimes^-$-

[5] In [14] Lamarche calls it the *dependency* ordering, but we prefer to use what seems to us standard terminology in Algorithmics.

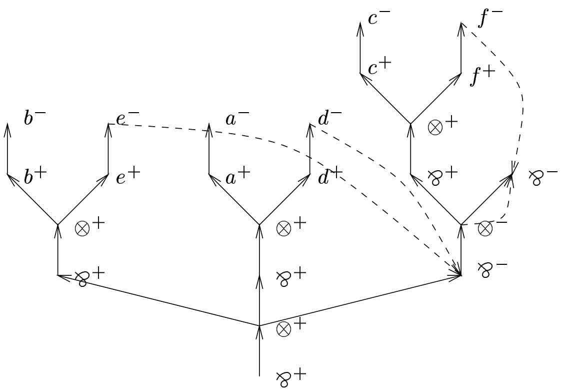


Figure 2. The dominator tree of an essential net.

node has exactly one positive successor and a finite number of negative successors, at most one of which is a $\otimes^-$-node and the rest are $\parr^-$-nodes. 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.

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 [15] in 1979 after a series of successive improvements by many authors. By using ingenious data structures, Harel [13] was able to reduce its complexity to linear time; his result was later refined by Alstrup, Harel, Lauridsen and Thorup in [1]. 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 [8]) which are costly to use.

Recently (in 1998) Buchsbaum, Kaplan, Rogers and Westbrook have introduced another linear-time algorithm [5], 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 [10] 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.

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 5). 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 3.1, each of the $\parr^-$-node 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 A, for constructing dominator trees of essential nets. Each iteration of the for-loop in Algorithm A 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 A depends on how fast one can compute the LCA (least common ancestors) queries. By using a multi-level algorithm due to Gabow [9], 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 A as part of a procedure to decide if an essential net is correct. If we do so naïvely, each node will be visited four times: acyclicity check and topological sort of the input net, construction of the dominator tree, and verification of Condition L for each $\parr^+$-node by a depth-first search of the dominator tree. This number can

[6] Buchsbaum [4] 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 A

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

1 Sort the input net topologically. (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. ($\textbf{\textit{link}}(w, v)$ 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 $\textbf{\textit{link}}(i, j)$

  3.2 else $\textbf{\textit{link}}(i, \mathsf{LCA}(j_1, j_2))$, where j_1 and j_2 are the two predecessors of i in N.

be reduced to one and we show how it can be done in the next section.

4 One-pass verification algorithms

We first give Algorithm B, 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 C.

A one-pass algorithm for ESSNET

In Algorithm B, $\textbf{\textit{link}}(w, v)$ and $\mathsf{LCA}(v, w)$ are the constant-time operations of Gabow [9]. Note that the Algorithm returns the dominator tree of the input essential net N in case N is correct.

We prove that Algorithm B always terminates and is correct.

Proposition 4.1. *(i) Algorithm B always terminates; if it returns YES, the input essential net is correct.*

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

(iii) If Algorithm B 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 $\parr^-$-node (which is a node of the cycle). Since the algorithm only visits a $\parr^-$-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 $\parr^+$-node satisfies Condition L. Thus if the Algorithm returns YES, the input net must be correct.
(ii) A node is added to T just before it is visited. If a $\parr^+$-node p dominates its sink $\mathit{sink}(p)$ 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 $\mathsf{LCA}(p, \mathit{sink}(p)) = p$.
(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 $\parr^-$-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 $\parr^-$-nodes through which v can be reached must have been visited. Hence v must have been visited as well – contradiction. $\square$

A one-pass algorithm for PROOFNET

Our next algorithm, Algorithm C, is a significant refinement of Algorithm B: 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 B 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 $\parr$-node below must then be given negative polarity. However, because polarities are not stored anywhere, when a $\parr$-node is visited with positive polarity, we need to ascertain that its sink has not been visited before. As before, the operations $\textbf{\textit{link}}(w, v)$ 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 4.2. *Algorithm C always terminates and is correct.*

Proof. The first command of $\textbf{\textit{visit2}}(p, v)$, which checks if v has already been visited, guarantees termination of the Algorithm. If the Algorithm returns YES, each node has been

[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 C 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 B

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 ***link***(w, v) makes w a successor node of v in T.)

2. Run ***visit***(v_0), where:

   visit(v) $=$ *Cases of* v :

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

3. If ***visit***(v_0) has visited every node then return YES else return NO.

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 C corresponding to each type of polarized nodes are exactly those in Algorithm B (as explained in the para. before the Algorithm). By the correctness of Algorithm B, 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 4.3. Algorithm C 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 C is very concise, though it is possibly slightly less clear.

5 Linear-time sequentialization algorithms

In this Section, we first present Algorithm D, a linear-time algorithm for *sequentializing* essential nets: it takes a correct essential net of an IMLL sequent as input and returns a derivation of that 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 D, and ending with a modicum of "bureaucratic" operations on derivations).

Sequentializing essential nets

Take a correct essential net N and fix a linear ordering $>$ of its nodes by a topological sort (so that the root is the least in the ordering, and that if there is an path from v to another node v' then $v < v'$). 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 A are ordered. (In Algorithm

Algorithm C

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 ***link***(w, v) makes w a successor node of v in T.)

2. Run ***visit2***$(+, v_0)$, where:

   visit2$(p : \textit{Polarity}, v : \textit{Node}) \quad =$

   if (v already visited) then return NO **;**

   Cases of (p, v) :

   – $(+$, end of axiom link ending in $v')$ $\quad \rightarrow \quad$ ***link***(v', v) **;** ***visit2***$(-, v')$
   – $(+, \otimes$-node), where v_1 and v_2 are the nodes above v $\quad \rightarrow \quad$ ***link***(v_1, v) **;** ***visit2***$(+, v_1)$ **;** ***link***(v_2, v) **;** ***visit2***$(+, v_2)$
   – $(+, \parr$-node) $\quad \rightarrow$

     if either of the two nodes above v has been visited then return NO
     else ***link***(v', v) **;** ***visit2***$(+, v')$, where v' is one of the two

   – $(-, v)$ $\quad \rightarrow$

     if v is a $\otimes$-node then ***link***(v', v) **;** ***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 $\quad \rightarrow \quad$ ***link***(w, v) **;** ***visit2***$(-, w)$
     – $\parr$-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 ***link***$(w, \mathsf{LCA}(v, v'))$ **;** ***visit2***$(-, w)$

3. If every node has been visited by ***visit2***$(+, v_0)$ then return YES else return NO

A, 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. We give a linear-time algorithm, called Algorithm D, 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***. We write *PosNode* (respectively *NegNode*) for the type of positive nodes (respectively negative nodes) of T_N, and write *Derivation* for the type of IMLL derivations. We define functions

> **Algorithm** D
>
> *Input:* a correct essential net $N = \mathcal{E}(\Gamma \vdash A)$
> *Output:* a derivation of $\Gamma \vdash A$
>
> 1. Run Algorithm A 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.

$$\begin{array}{rcl} \textbf{\textit{deriv}} & : & \textit{PosNode} \to \textit{Derivation} \\ \textbf{\textit{derivNeg}} & : & \textit{Derivation} \to [\textit{NegNode}\,] \to \textit{Derivation} \end{array}$$

by recursion, using a standard functional programming notation. We write $\mathit{succ}_{T_N}(v)$ to mean the successor of v in T_N.

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

1. positive atom a^+ (say) and $\mathit{succ}_{T_N}(v)$ is terminal in $T_N \quad \to \quad$ "$a \vdash a$"

2. positive atom a^+ (say) and $\mathit{succ}_{T_N}(v)$ is not terminal in $T_N \quad \to \quad$ ***derivNeg*** "$a \vdash a$" $[\,w\,]$

   where $\mathit{succ}_{T_N}(\mathit{succ}_{T_N}(v)) = w$

3. $\otimes^+$-node $\quad \to \quad$ ***derivNeg*** $d\,[\,w_1, \cdots, w_n\,]$

   where: d is the derivation formed from ***deriv*** v_1 and ***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. $\parr^+$-node $\quad \to \quad d$

   where: d is formed from ***deriv*** $\mathit{succ}_{T_N}(v)$ by the rule ($\multimap$-r) where the formula eliminated is given by the sink of v.

The definition of ***derivNeg*** has two clauses.

$$\begin{array}{rcl} \textbf{\textit{derivNeg}}\ d\,[\,] & = & d \\ \textbf{\textit{derivNeg}}\ d\,(w : ws) & = & \textit{Cases of } w : \end{array}$$

1. $\parr^-$-node and terminal in $T_N \quad \to \quad$ ***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. $\parr^-$-node which is not terminal in $T_N \quad \to$ ***derivNeg*** $d'\ (w' : ws)$

   where: d' is the same as that in the previous case and $w' = \mathit{succ}_{T_N}(w)$

3. $\otimes^-$-node $\quad \to \quad$ ***derivNeg*** $d'\ ([w_1, \cdots, w_n] {+\!\!+} ws)$

   where: d' is the derivation formed from ***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) where v is the positive successor, and $w_1, \cdots, w_n$ are the negative successors (in ascending topological order), of w in T_N.

A proof of the correctness of Algorithm D is given in the full version of the paper [17].

Sequentializing proof nets

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

> **Algorithm** E
>
> *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 D 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.

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 D gives enough information for the reconstruction of a derivation of the (original) input proof net.

Acknowledgements

We are grateful to 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. Murawski is supported by a University of Oxford Scatcherd European Scholarship and a British Overseas Research Students Award 98032135.

References

[1] S. Alstrup, D. Harel, P. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28 (6):2117–2132, 1999.

[2] G. Bellin. Proof nets for classical MLL and linear lambda terms. A March 1992 posting to LINEAR `http://www.csl.sri.com/linear/mailing-list-traffic/www/all.html`, 1992.

[3] G. Bellin and P. J. Scott. Remarks on the π-calculus and linear logic. *Theoretical Computer Science*, 135:11–65, 1994.

[4] A. Buchsbaum. Private email communication. January 2000.

[5] A. Buchsbaum, H. Kaplan, A. Rogers, and J. Westbrook. A new, simpler linear-time dominators algorithm. *Transactions on Programming Languages and Systems*, 6:1265–1296, 1998.

[6] V. Danos. *La logique linéaire Appliquée à l'étude de divers processus de normalisation et principalement du λ-calcul*. PhD thesis, Université de Paris, 1990.

[7] V. Danos and L. Regnier. The structures of multiplicatives. *Archive for Mathematical Logic*, 28:181–203, 1989.

[8] M. Fredman and D. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer Science and Systems*, 48:533–551, 1994.

[9] H. Gabow. Data structure for weighted matching and nearest common ancestors with linking. In *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[10] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[11] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[12] S. Guerrini. Correctness of multiplicative proof nets is linear. In *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 454–463. IEEE Computer Science Society, 1999.

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

[14] F. Lamarche. Proof Nets for Intuitionistic Linear Logic 1: Essential Nets. Preprint ftp-able from *Hypatia*, 1994.

[15] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[16] A. S. Murawski and C.-H. L. Ong. A linear-time algorithm for verifying MLL proof nets via essential nets. In Davis, Roscoe, and Woodcock, editors, *Collected Papers of the Oxford-Microsoft Symposium in Celebration of the Work of C. A. R. Hoare*. McMillan, 2000? Submitted, 11 pp, Cornerstones in Computing Series, ftp-able from Ong's homepage.

[17] A. S. Murawski and C.-H. L. Ong. Dominator trees and fast verification of proof nets. 18 pp. Preprint. Ftp-able from Ong's homepage, 2000.

[18] A. S. Murawski and C.-H. L. Ong. Exhausting Strategies, Joker Games and Full Completeness for IMLL with Unit. Submitted to *Theoretical Computer Science*, 41 pp., ftp-able from Ong's homepage, 2000.