linTAP : A Tableau Prover for Linear Logic

Heiko Mantel[1] Jens Otten[2]

[1] *German Research Center for Artificial Intelligence (DFKI)*
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
mantel@dfki.de
[2] *Fachgebiet Intellektik, Fachbereich Informatik, Darmstadt University of Technology*
Alexanderstr. 10, 64283 Darmstadt, Germany
jeotten@informatik.tu-darmstadt.de

Abstract. linTAP is a tableau prover for the multiplicative and exponential fragment $\mathcal{M}?\mathcal{LL}$ of Girards linear logic. It proves the validity of a given formula by constructing an analytic tableau and ensures the linear validity using prefix unification. We present the tableau calculus used by linTAP, an algorithm for prefix unification in linear logic, the linTAP implementation, and some experimental results obtained with linTAP.

1 Introduction

Linear logic [12] can be regarded as a refinement of classical as well as of intuitionistic logic. It subsumes these logics because both of them can be embedded into linear logic. Mainly, linear logic has become known as a very expressive logic of action and change. It has found applications in logic programming [14,20], planing [19], modeling concurrent computation [11], and other areas. Its expressiveness, however results in a high complexity. Validity is undecidable for propositional linear logic. The multiplicative fragment is already $\mathcal{NP}$-complete [16]. The complexity of the multiplicative exponential fragment ($\mathcal{MELL}$) is still unknown. Consequently, proof search in linear logic is difficult to automate.

Various calculi have been developed for linear logic. Beginning with the sequent calculus and proof nets by Girard [12], several optimizations have been proposed. More recently, the connection method has been extended to fragments of linear logic [8,9,15,17]. In this article, we propose a tableau calculus for $\mathcal{MELL}$ and for $\mathcal{M}?\mathcal{LL}$ which is the theoretical basis for our theorem prover linTAP.

linTAP is implemented in a very compact way but uses sophisticated techniques to reduce the search space and thus follows the idea of *lean theorem proving*. It was inspired by the classical tableau prover lean*TAP* [2,3] and by the intuitionistic tableau prover ileanTAP [21]. Like in ileanTAP, string unification is used to deal with the non-permutabilities specific to linear logic. This approach has been invented by Wallen in the context of matrix characterizations for non-classical logics [25]. The prefixes used by linTAP are motivated by a matrix characterization for $\mathcal{MELL}$ [17]. In our implementation of linTAP we use a lean*TAP* like technique for path checking and then try to unify the so-called prefixes of atoms which are closing the branches of the tableau proof like in ileanTAP. Some additional checks are required because of the resource sensitivity of linear logic. Some of these checks are tested already during proof construction.

N. V. Murray (Ed.): Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX'99, Saratoga Springs, NY, USA, LNAI 1617, pp. 217–231, June 1999.

After some preliminaries we propose a tableau calculus for $\mathcal{MELL}$ in Section 3. The application of a calculus rule to a formula de-constructs the formula and constructs a prefix for the resulting sub-formulas. An algorithm for the unification of such prefixes is presented in Section 4. A tableau calculus for $\mathcal{M}?\mathcal{LL}$, where ? and ! can only occur, respectively, positively and negatively, some details about our theorem prover linTAP, and some experimental results are discussed in Section 5. We conclude with some remarks on related and on future work.

2 Preliminaries

Linear logic [12] treats formulas like resources that disappear after their use unless they are explicitly marked as reusable. It can be seen as the outcome of removing the rules for contraction and weakening from the classical sequent calculus and re-introducing them in a controlled manner. Linear negation $^\perp$ is involutive like classical negation. The two traditions for writing the sequent rule for conjunction result in two different conjunctions $\otimes$ and & and two different disjunctions $\parr$ and $\oplus$. The constant `true` splits up into $\mathbf{1}$ and $\top$ and `false` into $\perp$ and $\mathbf{0}$. The unary connectives ? and ! mark formulas for a controlled application of weakening and contraction. Quantifiers $\forall$ and $\exists$ are added as usual.

Linear logic can be divided into the multiplicative, additive, and exponential fragment. While in the multiplicative fragment resources are used exactly once, resource sharing is enforced in the additive fragment. Exponentials mark formulas as reusable. All fragments exist on their own right and can be combined freely. The full power of linear logic comes from combining all of them.

In this article we focus on multiplicative exponential linear logic ($\mathcal{MELL}$ and $\mathcal{M}?\mathcal{LL}$), the combination of the multiplicative and exponential fragments, leaving the additive fragment and the quantifiers out of consideration. $^\perp$, $\otimes$, $\parr$, $\multimap$, $\mathbf{1}$, $\perp$, !, and ? are the connectives of $\mathcal{MELL}$. In $\mathcal{M}?\mathcal{LL}$, ? and ! only occur, respectively, with positive and negative polarity. Linear negation $^\perp$ expresses the difference between resources that are to be used up and resources to be produced. In order to use up $F^\perp$ a resource F must be produced. Having a resource $F_1 \otimes F_2$ means having F_1 as well as F_2. $F_1 \multimap F_2$ allows the construction of F_2 from F_1. $F_1 \parr F_2$ is equivalent to ${F_1}^\perp \multimap F_2$ and to ${F_2}^\perp \multimap F_1$. Having a resource $\mathbf{1}$ has no impact while nothing can be constructed when $\perp$ is used up. A resource $!F$ acts like a machine which produces any number of copies of F. During the construction of $!F$ only such machines can be used. ? is the dual to !.

We adopt Smullyan's uniform notation to $\mathcal{MELL}$. A *signed formula* $\varphi = F^k$ denotes an occurrence of F in Δ or Γ. Depending on the *label* F and its *polarity* $k \in \{+, -\}$, a signed formula will receive a *type* α, β, ν, π, o, τ, ω, or a according to the tables below. The functions $succ_1$ and $succ_2$ return the major signed subformulas of a signed formula. Note that during the decomposition of a formula the polarity switches only for $^\perp$ and $\multimap$. We use type symbols as meta-variables for signed formulas of the respective type, e.g. α stands for a signed formula of type α and a stands for atomic formulas, i.e. signed predicates.

The validity of a linear logic formula can be proven syntactically by using a sequent calculus. For multi-sets Γ and Δ of formulas $\Gamma \longrightarrow \Delta$ is called a *sequent*.

a	A^-	A^+
τ	$\bot^-$	$\mathbf{1}^+$
ω	$\mathbf{1}^-$	$\bot^+$

α	$(F_1 \otimes F_2)^-$	$(F_1 \parr F_2)^+$	$(F_1 \multimap F_2)^+$
$succ_1(\alpha)$	F_1^-	F_1^+	F_1^-
$succ_2(\alpha)$	F_2^-	F_2^+	F_2^+

β	$(F_1 \otimes F_2)^+$	$(F_1 \parr F_2)^-$	$(F_1 \multimap F_2)^-$
$succ_1(\beta)$	F_1^+	F_1^-	F_1^+
$succ_2(\beta)$	F_2^+	F_2^-	F_2^-

o	$(F^\bot)^-$	$(F^\bot)^+$
$succ_1(o)$	F^+	F^-

ν	$(!F)^-$	$?F^+$
$succ_1(\nu)$	F^-	F^+

π	$(?F)^-$	$!F^+$
$succ_1(\pi)$	F^-	F^+

Table 1. Uniform notation for signed $\mathcal{MELL}$ formulas

It can be understood as the specification of a transformation which constructs Δ from Γ. The formulas in Γ are connected implicitly by $\otimes$ while the formulas in Δ are connected implicitly by $\parr$.

A sequent calculus Σ_1' for $\mathcal{MELL}$ based on our uniform notation is depicted in Table 2. Omitting the π-rule yields a calculus for $\mathcal{M}?\mathcal{LL}$. In a rule, the sequents above the line are the *premises* and the one below is the *conclusion*. A *principal formula* is a formula that occurs in the conclusion but not in any premise. Formulas that occur in a premise but not in the conclusion are called *active*. All other formulas compose the *context*. Σ_1' is correct and complete wrt. Girard's original sequent calculus [12].

$$\frac{}{\longrightarrow A^+, A^-}\ axiom \qquad \frac{}{\longrightarrow \tau}\ \tau \qquad \frac{\longrightarrow \Upsilon}{\longrightarrow \Upsilon, \omega}\ \omega \qquad \frac{\longrightarrow \Upsilon, succ_1(o)}{\longrightarrow \Upsilon, o}\ o$$

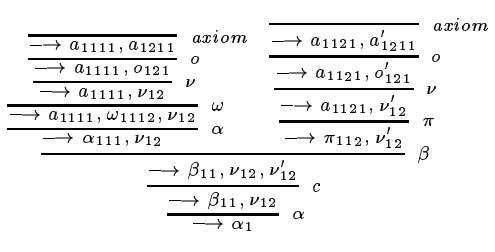
$$\frac{\longrightarrow \Upsilon, succ_1(\alpha), succ_2(\alpha)}{\longrightarrow \Upsilon, \alpha}\ \alpha \qquad \frac{\longrightarrow \Upsilon_1, succ_1(\beta) \quad \longrightarrow \Upsilon_2, succ_2(\beta)}{\longrightarrow \Upsilon_1, \Upsilon_2, \beta}\ \beta$$

$$\frac{\longrightarrow \Upsilon, succ_1(\nu)}{\longrightarrow \Upsilon, \nu}\ \nu \qquad \frac{\longrightarrow \boldsymbol{\nu}, succ_1(\pi)}{\longrightarrow \boldsymbol{\nu}, \pi}\ \pi \qquad \frac{\longrightarrow \Upsilon}{\longrightarrow \Upsilon, \nu}\ w \qquad \frac{\longrightarrow \Upsilon, \nu, \nu}{\longrightarrow \Upsilon, \nu}\ c$$

Table 2. Sequent calculus Σ_1' for $\mathcal{MELL}$ in uniform notation

In *analytic proof search*, one starts with the sequent to be proven and reduces it by application of rules until the *axiom*-rule or the τ-rule can be applied. There are several choice points within this process. As in classical logic, first, a principal formula must be chosen. Unless the principal formula has type ν, this choice determines which rule must be applied. Formulas of type ν are *generic*. They can be duplicated using the *contraction* rule c and are removed by the *weakening* rule w. When the β-rule is applied the context of the sequent must be split, i.e. Υ_1 and Υ_2 must be a partition of the context. Several solutions have been proposed in order to optimize these choices [1,10,23,6,13]. Additional difficulties arise from the rules *axiom*, τ, and π. The rules *axiom* and τ require an empty context which expresses that all formulas must be used up in a proof. The π rule requires that all formulas in the context are of type ν. The careful handling of the context reflects the resource sensitivity of linear logic.

Example 1. Figure 1 presents a Σ_1'-proof of $\varphi = (((A \parr \bot) \otimes !A) \parr ?(A^\bot))^+$. We abbreviate occurrences of subformulas of φ by position markers as shown in the table on the right. Note that any proof of φ requires that the contraction rule c is applied before the β-rule.



$lab(\varphi')$	φ'
$((A \parr \bot) \otimes !A) \parr ?(A^\bot)$	α_1
$(A \parr \bot) \otimes !A$	β_{11}
$A \parr \bot$	α_{111}
A	a_{1111}
$\bot$	ω_{1112}
$!A$	π_{112}
A	a_{1121}
$?(A^\bot)$	ν_{12}, ν'_{12}
$A^\bot$	o_{121}, o'_{121}
A	a_{1211}, a'_{1211}

Fig. 1. An example Σ'_1-proof.

3 A Tableau Calculus

The tableau calculus presented in this section is motivated by a matrix characterization for $\mathcal{MELL}$ [17].

Basic Definitions. We assume disjoint sets Φ^M, Ψ^M, Φ^E, and Ψ^E of characters. ϕ^M, ψ^M, ϕ^E, and ψ^E are used as meta-variables for characters from the respective set. Elements of Φ^M and Ψ^M are called *multiplicative* and elements of Φ^E, and Ψ^E are called *exponential*. Characters in Φ^M and Φ^E are called *variable* and characters in Ψ^M and Ψ^E are called *constant*. The intuition is that variable characters can be substituted while constant characters cannot. A *prefix* s is a string over these sets, i.e. $s \in (\Phi^M \cup \Psi^M \cup \Phi^E \cup \Psi^E)^*$. A *multiplicative string substitution* is a mapping $\sigma_M : \Phi^M \to (\Phi^M \cup \Psi^M)^*$. An *exponential string substitution* is a mapping $\sigma_E : \Phi^E \to (\Phi^M \cup \Psi^M \cup \Phi^E \cup \Psi^E)^*$. A *string substitution* is a mapping $\sigma : (\Phi^M \cup \Phi^E) \to (\Phi^M \cup \Psi^M \cup \Phi^E \cup \Psi^E)^*$ such that the restriction of σ to Φ^M is a multiplicative string substitution and the restriction to Φ^E is an exponential string substitution. We extend σ homomorphically to strings from $(\Phi^M \cup \Psi^M \cup \Phi^E \cup \Psi^E)^*$ where σ is the identity on constant characters.[1]

A *position* p is a string from $\mathcal{P} = \{l, r\}^* \cup \{0\}$. p is a *sub-position* of a position p' if p' is a proper prefix of p, e.g. lrl is a sub-position of lr. A *multiplicity* μ is a function which assigns natural numbers to positions, i.e. $\mu : \mathcal{P} \to \mathbb{N}$. Using multiplicities, we determine the number of duplicates of generic formulas in a tableau. We mark each occurrence of a formula in a tableau proof with a position. In a tableau for φ, φ is marked with position 0. If φ is marked with p then the left and right subformula of φ are marked with $p \circ l$ and $p \circ r$, respectively. For a generic formula φ the jth instance of the subformula is marked with $p \circ l^j$.

Definition 2. Let φ be a signed formula, s be a prefix, and p be a position. Then $\varphi : s\ (p)$ is called a *prefixed formula*. If φ is of type a, τ, ω, or ν then $[\varphi] : s\ (p)$ is *prefixed formula* as well. We refer to the later kind as *marked prefixed formulas*.

The type of a prefixed formula $\varphi : s\ (p)$ is the type of φ. We use the same meta-variables for prefixed formulas as for signed formulas. If necessary, we will point out what kind of formula is denoted by a specific meta-variable.

Definition 3. A *connection* is a one-element set containing a marked prefixed formula of type τ or a two-element set containing two marked atomic prefixed formulas with the same label and opposite signs. A *weakening map* is a set of marked prefixed formulas of type ω and of type ν with multiplicity 0.

[1] For $\mathcal{M}?\mathcal{LL}$ the set Ψ^E is not needed. Thus, Φ^M and Φ^E need not be distinguished.

For example, $\{[\mathbf{1}]^+ : s'\ (p')\}$ and $\{[A]^+ : s''\ (p''), [A]^- : s'''\ (p''')\}$ are connections; $\emptyset$, $\{[\bot]^+ : t'\ (q')\}$, and $\{[?F]^+ : t''\ (q''), [\mathbf{1}]^- : t'''\ (q''')\}$ are weakening maps.

Note, that Definition 3 imposes the same restrictions on the elements of a connection as the Σ_1'-rules τ and *axiom* do for the principal formulas in order to close a branch. It requires the same properties for the elements of a weakening map as the Σ_1'-rules ω and w do for the principal formulas in order to remove a formula. This resembles the relation between a proof according to the connection method and the set of sequent proofs represented by it [17]. It should be helpful to keep this in mind in order to grasp the intuition behind the following definitions.

Complementarity Conditions and Closed Tableaux. We now define some *complementarity conditions* which are crucial for our definition of closed tableaux in $\mathcal{MELL}$. Each condition is motivated by a property of the sequent calculus Σ_1' in Table 2 and, if possible, an intuitive explanation based on the resource sensitivity of linear logic is given. In the following we always assume $\mathcal{C}$ to be a set of connections, $\mathcal{W}$ to be a weakening map, and σ to be a string substitution.

- Resources can be used at most once and disappear after their use. In Σ_1' this is reflected by the lack of a general rule for contraction and by the context split in the β-rule. *$\mathcal{C}$ is linear* if each prefixed formula occurs in at most one connection. *$\mathcal{C}$ and $\mathcal{W}$ are linear* if $\mathcal{C}$ is linear and p is not a sub-position of p' for any $\varphi : s\ (p)$ which occurs in a connection from $\mathcal{C}$ and any $\varphi' : s'\ (p') \in \mathcal{W}$. Intuitively, this linearity condition says that a formula cannot contribute to an axiom in a corresponding sequent proof if it has been weakened and that it cannot contribute to more than one axiom.
- Resources cannot disappear without a reason. They must be consumed. In Σ_1' this is reflected by the lack of a general rule for weakening and by the requirement of an empty context in the rules *axiom* and τ. *$\mathcal{C}$ and $\mathcal{W}$ are relevant for a set of prefixed formulas Υ* if each $\varphi : s\ (p) \in \Upsilon$ occurs at least in one connection or a $\varphi' : s'\ (p') \in \Upsilon$ occurs in $\mathcal{W}$ where p is a sub-position of p'. Intuitively, relevance demands for a corresponding sequent proof that a formula must contribute to an axiom unless it has been weakened.
- In Σ_1', the context only is divided by the application of a β-rule. Let $\boldsymbol{\beta}$ be a set of prefixed formulas of type β and let $\beta_W = \{(\beta : s\ (p)) \in \boldsymbol{\beta} \mid$ *there is no* $(\varphi : s'\ (p')) \in \mathcal{W}$ *such that p is a sub-position of p'*$\}$. *$\mathcal{C}$ and $\mathcal{W}$ have the right cardinality for $\boldsymbol{\beta}$* if $|\mathcal{C}| = |\boldsymbol{\beta}_W| + 1$.
- In Σ_1', certain rule applications can be permuted while others cannot. The non-permutability of rules for linear logic has been investigated e.g. in [10]. The existence of a suitable order of non-permutable rule applications is expressed by the unifiability of prefixes. *$\mathcal{C}$ and $\mathcal{W}$ are unified by σ* if
  - for each $c \in \mathcal{C}$ the prefixes of all elements of c are identical under σ and
  - for each $\varphi \in \mathcal{W}$ there is a $c \in \mathcal{C}$ such that the prefix of φ is an initial substring of the prefix of the elements of c under σ.

Definition 4. Let φ be a prefixed formula.

1. The one-branch tree φ is a *tableau* for φ.
2. If T is a tableau for φ and T^* results from T by the application of a tableau expansion rule from Table 3 then T^* is a tableau for φ.

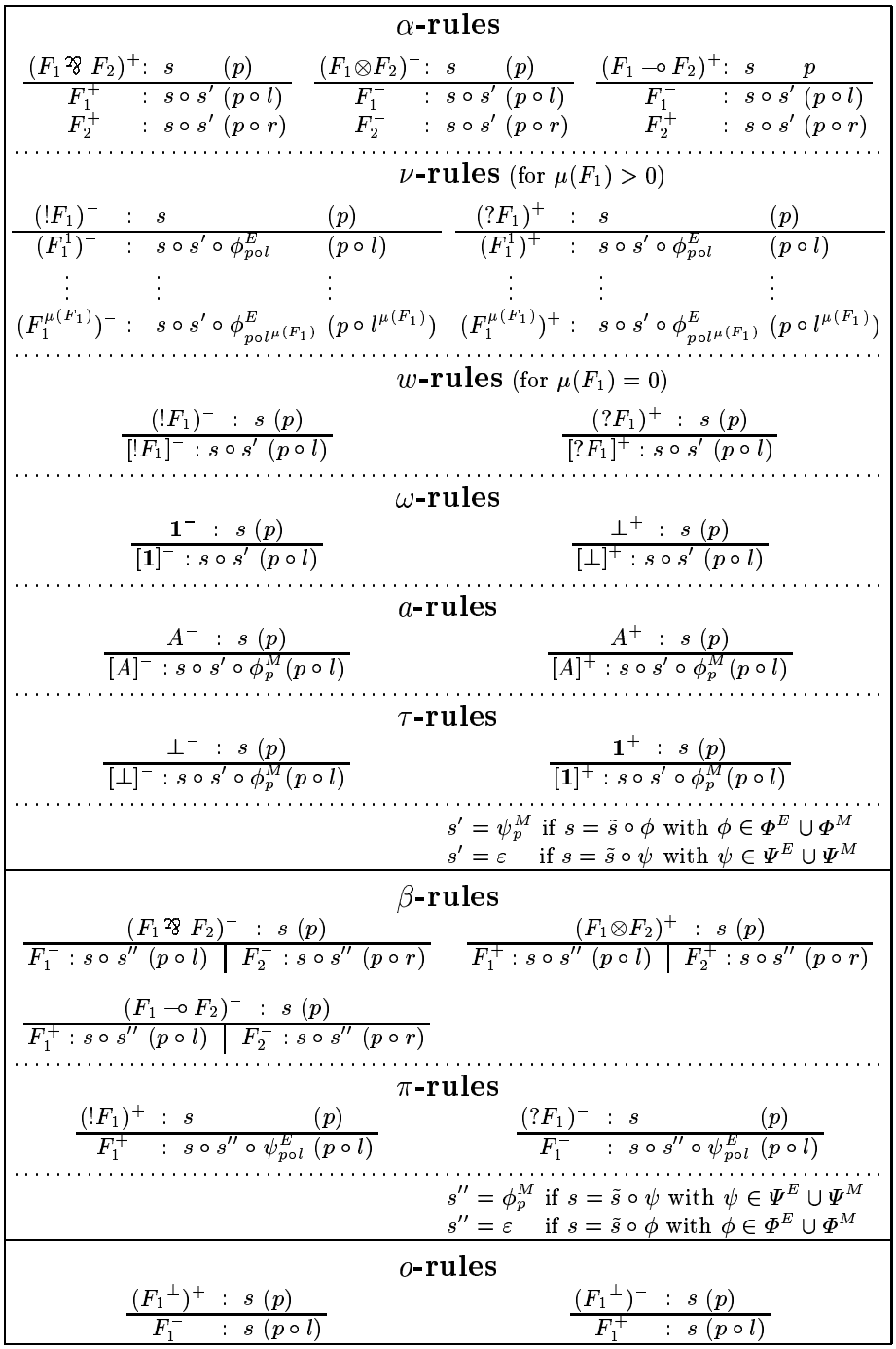


Table 3. A prefixed-based tableau calculus for $\mathcal{MELL}$

Expansion rules are applied as usual [7]. The application of a rule de-constructs a formula, possibly enlarges the prefix, and constructs a position. A tableau is *strict* if each occurrence of a formula is reduced at most once on a branch. In a strict tableau, prefixed formulas can be uniquely identified by their positions. In the sequel, we will consider only strict tableaux and will extensively use the isomorphism between formulas in a given tableaux and their positions.

Definition 5. A *branch of a tableau is closed* by a connection c if all elements of c occur on that branch.

Let T be a tableau for φ, Υ_T be the set of prefixed formulas of type a, τ, ω, and ν (with multiplicity 0) which occur in T, and β_T be the set of prefixed formulas of type β in T. Further, let $\mathcal{C}$ be a set of connections where the elements of connections are from Υ_T. Let $\mathcal{W}$ be a weakening map with elements from Υ_T. Let σ be a string substitution. Then $\mathcal{C}$ and $\mathcal{W}$ *fulfill linearity in* T if $\mathcal{C}$ and $\mathcal{W}$ are linear. $\mathcal{C}$ and $\mathcal{W}$ *fulfill relevance in* T if $\mathcal{C}$ and $\mathcal{W}$ are relevant for Υ_T. $\mathcal{C}$ and $\mathcal{W}$ *fulfill cardinality in* T if $\mathcal{C}$ and $\mathcal{W}$ have the right cardinality for β_T. $\mathcal{C}$, $\mathcal{W}$, and σ *fulfill unifiability for* T if $\mathcal{C}$ and $\mathcal{W}$ are unified by σ.

Definition 6. Let T be a tableau for a prefixed formula φ. Further, let $\mathcal{C}$ be a set of connections, $\mathcal{W}$ be a weakening map, and σ be a string substitution. T is *closed by* $\mathcal{C}$, $\mathcal{W}$, *and* σ *iff* the following conditions hold:

- Each branch of T is closed by a connection from $\mathcal{C}$.
- $\mathcal{C}$, $\mathcal{W}$, and σ fulfill linearity, relevance, cardinality, and unifiability for T.

Example 7. A tableau T for $((A \parr \bot) \otimes !A) \parr ?(A^{\bot})$ is depicted in Figure 2. The set of connections $\mathcal{C}$=$\{\{0llll, 0rlll\}, \{0lrll, 0rllll\}\}$ closes the branches of the tableau. Let $\mathcal{W} = \{0llrl\}$ be a weakening map and σ be a substitution with $\sigma(\phi^E_{0rl}) = \phi^M_{0l}\psi^M_{0ll}\phi^M_{aux1}$, $\sigma(\phi^M_{0lll}) = \phi^M_{aux1}\psi^M_{0rll}\phi^M_{0rll}$, $\sigma(\phi^E_{0rll}) = \phi^M_{0l}\psi^E_{0lrl}\phi^M_{aux2}$, and $\sigma(\phi^M_{0lrl}) = \phi^M_{aux2}\psi^M_{0rlll}\phi^M_{0rlll}$. Then T is closed by $\mathcal{C}$, $\mathcal{W}$, and σ.

The following theorems state that the tableau calculus in Table 3 is correct and complete. In order to follow the proof sketches prior knowledge of [17] is required.

Theorem 8 (Correctness). *If there is a closed tableau for a prefixed formula* $\varphi = F^+ : \psi^M_0$ (0) *for some multiplicity* μ *then* F *is valid.*

Proof Sketch: Let T be a tableau for φ with multiplicity μ which is closed by $\mathcal{C}$, $\mathcal{W}$, and σ. We construct a matrix proof for the matrix M of φ. The correctness of the matrix characterization in [17] then implies that φ is valid.

For every prefixed formula φ' in T there is a corresponding node n in M with the same label, polarity, type, and ancestors which are equivalent under this relation. Let m be an injective mapping which assigns to a formula in T a corresponding node in M. All non-special nodes in M are in the image of m. We define the application of m to sets as the application of m to the elements. For any path of leaves P through M there is branch B in T with marked formulas Υ_B such that $m(\Upsilon_B) \subseteq P$ holds. Let $\mathcal{C}_M = m(\mathcal{C})$ and $\mathcal{W}_M = m(\mathcal{W})$. Since all branches of T are closed by $\mathcal{C}$, $\mathcal{C}_M$ is spanning for M. $\mathcal{C}_M$ and $\mathcal{W}_M$ are linear,

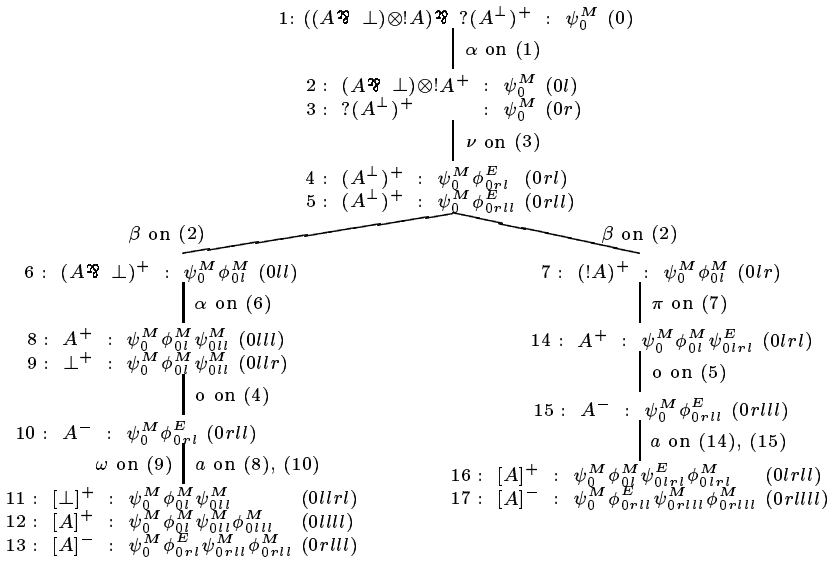


Fig. 2. An example tableau

relevant, and have the cardinality property for M. Marked formulas of type a, τ, ω, and ν have the same prefix (under renaming) as the correspondent nodes in M. Therefore, σ is a unifier for $\mathcal{C}_M$ and $\mathcal{W}_M$ in M. Thus, M is complementary for $\mathcal{C}_M$, $\mathcal{W}_M$, and σ.

Theorem 9 (Completeness). *If a formula F is valid then there exists a closed tableau for the prefixed formula $\varphi = F^+ : \psi_0^M\ (0)$.*

Proof Sketch: From any matrix proof of the matrix M of φ a closed tableau for φ can be constructed. The completeness of the matrix characterization then implies that a tableau for any valid formula exists.

The crucial step is that if there is a complementary matrix M' for φ with multiplicity μ' then there is a multiplicity μ, a set of connections $\mathcal{C}$, a weakening map $\mathcal{W}$ without elements of type ϕ^E, and a string substitution σ such that the matrix M for φ with multiplicity μ is complementary for $\mathcal{C}$, $\mathcal{W}$, and σ. Using $\mathcal{C}$, $\mathcal{W}$, and σ, a closed tableau for φ can be constructed.

4 Prefix Unification

The computation of a string substitution σ is one of the key components necessary to perform proof search in the prefix-based tableau calculus introduced in the previous section. A single string substitution σ has to unify the prefixes of each connection in the set $\mathcal{C}$. Furthermore the weakening map $\mathcal{W}$ has to fulfill the unifiability condition under this substitution σ. This condition can be reduced to unification since a prefix s is an initial substring of a prefix t iff $s \circ V$ and t can be unified where V is a new variable.

String unification in general is rather complicated but fortunately unifying prefixes is much easier since there are two *restrictions on prefixes*: prefixes are

strings without duplicates and in any two prefixes (corresponding to atoms of the same formula) equal characters can only occur within a common substring at the beginning of the prefixes. In [22] we introduced a prefix unification algorithm, so-called *T-String Unification*, to unify prefixes in matrix based proof methods for non-classical logics, i.e. intuitionistic logic and the modal logics D, K, D4, K4, S5, and T. Only minor modifications are necessary to adapt this algorithm to deal with the prefixes arising in our tableau calculus for $\mathcal{MELL}$: we have to distinguish between characters (i.e. positions) of type ϕ^M/ψ^M and type ϕ^E/ψ^E.

Similar to the ideas of Martelli and Montanari [18] we consider the process of unification as a sequence of transformation steps. We start with the given set of (prefix-) equations $\Gamma = \{p_1{=}t_1, \ldots, p_n{=}t_n\}$ and an empty substitution $\sigma{=}\emptyset$. Each transformation step replaces the tuple (Γ, σ) by a modified tuple (Γ', σ') where Γ' is the result of replacing one equation $\{p_i{=}t_i\}$ in Γ by $\{p_i'{=}t_i'\}$ and applying the substitution σ' to the resulting equation set. The algorithm is described by transformation rules " $\{s_i{=}t_i\}, \sigma \rightarrow \{s_i'{=}t_i'\}, \sigma'$ " which can be applied nondeterministically to the selected equation $\{s_i{=}t_i\} \in \Gamma$. The set Γ is solvable, iff there are some transformation steps transforming the tuple $(\Gamma, \emptyset)$ into the tuple $(\emptyset, \tilde{\sigma})$. In this case the substitution $\tilde{\sigma}$ represents an idempotent most general unifier for Γ. The set of all resulting most general unifiers is *minimal*. For technical reasons we divide the right part t_i of each equation into two parts $t_i^1|t_i^2$ where the left part contains the substring which is not yet assigned to a variable. Therefore we start with the set of prefixes $\Gamma = \{s_1{=}\varepsilon|t_1, \ldots, s_n{=}\varepsilon|t_n\}$.

Definition 10. Let $\mathcal{V}{=}\Phi^M \cup \Phi^E$ be a set of variables, $\mathcal{C}{=}\Psi^M \cup \Psi^E$ be a set of constants, $\tilde{\mathcal{V}}^M$ and $\tilde{\mathcal{V}}^E$ be disjoint sets of *auxiliary variables*, $\mathcal{V}'{=}\tilde{\mathcal{V}}^M \cup \tilde{\mathcal{V}}^E$ (with $\mathcal{V} \cap \mathcal{V}'{=}\emptyset$), $\mathcal{V}^M{=}\Phi^M \cup \tilde{\mathcal{V}}^M$, and $\mathcal{V}^E{=}\Phi^E \cup \tilde{\mathcal{V}}^E$. The set of *transformation rules for* $\mathcal{MELL}$ is defined in Table 4.

R1.	$\{\varepsilon = \varepsilon\|\varepsilon\},\ \sigma$	$\rightarrow$	$\{\},\ \sigma$	
R2.	$\{\varepsilon = \varepsilon\|t^+\},\ \sigma$	$\rightarrow$	$\{t^+ = \varepsilon\|\varepsilon\},\ \sigma$	
R3.	$\{Xs = \varepsilon\|Xt\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon\|t\},\ \sigma$	
R4.	$\{Cs = \varepsilon\|Vt\},\ \sigma$	$\rightarrow$	$\{Vt = \varepsilon\|Cs\},\ \sigma$	
R5.	$\{Vs = z\|\varepsilon\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon\|\varepsilon\},\ \{V\backslash z\}\cup\sigma$	
R6.	$\{Vs = \varepsilon\|C_1t\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon\|C_1t\},\ \{V\backslash\varepsilon\}\cup\sigma$	
R7.	$\{Vs = z\|C_1C_2t\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon\|C_2t\},\ \{V\backslash zC_1\}\cup\sigma$	($V \in \mathcal{V}^E$ or $C_1 \in \Psi^M$)
R8.	$\{Vs^+ = \varepsilon\|V_1t\},\ \sigma$	$\rightarrow$	$\{V_1t = V\|s^+\},\ \sigma$	($V_1 \in \mathcal{V}^E$ or $V \in \mathcal{V}^M$)
R8′.	$\{Vs^+ = \varepsilon\|V_1t\},\ \sigma$	$\rightarrow$	$\{V_1t = V\|s^+\},\ \{V\backslash\tilde{V}^M\}\cup\sigma$	($V_1 \in \mathcal{V}^M$ and $V \in \mathcal{V}^E$)
R9.	$\{Vs^+ = z^+\|V_1t\},\ \sigma$	$\rightarrow$	$\{V_1t = V'\|s^+\},\ \{V\backslash z^+V'\}\cup\sigma$	(if $V_1, V \in \mathcal{V}^E$ then $V' \in \tilde{\mathcal{V}}^E$ else $V' \in \tilde{\mathcal{V}}^M$)
R10.	$\{Vs = z\|Xt\},\ \sigma$	$\rightarrow$	$\{Vs = zX\|t\},\ \sigma$	($V \in \mathcal{V}^E$ or $X \in (\mathcal{V}^M \cup \Psi^M)$)
R10′.	$\{Vs = z\|Xt\},\ \sigma$	$\rightarrow$	$\{Vs = zX\|t\},\ \{X\backslash\tilde{V}^M\}\cup\sigma$	($V \in \mathcal{V}^M$ and $X \in \mathcal{V}^E$)

$s,t,z \in (\mathcal{V}\cup\mathcal{C}\cup\mathcal{V}')^*$ denote (arbitrary) strings, $s^+,t^+,z^+ \in (\mathcal{V}\cup\mathcal{C}\cup\mathcal{V}')^+$ denote non-empty strings. X, V, V_1, C, C_1 and C_2 denote single characters with $X \in \mathcal{V}\cup\mathcal{C}\cup\mathcal{V}'$, $V, V_1 \in \mathcal{V}\cup\mathcal{V}'$ (with $V{\neq}V_1$), and $C, C_1, C_2 \in \mathcal{C}$. V' and $\tilde{V}^M \in \tilde{\mathcal{V}}^M$ are new variables which do not occur in the substitution σ computed so far. To apply rule R10 or R10′ the following must hold: $V{\neq}X$, and $s{=}\varepsilon$ or $t{\neq}\varepsilon$ or $X \in \mathcal{C}$.

Table 4. Transformation rules for $\mathcal{MELL}$

These rules are identical with the transformation rules presented in [22]. We only added rules R8′ and R10′ (which are applied instead of rules R8 and R10 in certain cases) and some additional restrictions for the rules R7 and R9 (characters $X \in \mathcal{V}^E \cup \Psi^E$ cannot be assigned to variables $V \in \mathcal{V}^M$). We use the notation $\{x \backslash t \mid \sigma(x)=t \text{ and } x \neq t\}$ to specify a substitution σ and omit the string concatenation operator "$\circ$". See [22] for a graphical motivation of these rules, a more detailed description of the algorithm, and some complexity results.

Example 11. Consider the formula $((A \parr \bot) \otimes !A) \parr ?(A^\bot)$ from Example 7 and the unification of the two prefixes $\psi_0^M \phi_{0l}^M \psi_{0lrl}^E \phi_{0lrl}^M$ and $\psi_0^M \phi_{0rll}^E \psi_{0rlll}^M \phi_{0rlll}^M$. To keep the notation simple we substitute each character ϕ_p and ψ_p by $V_{|p|}$ and $C_{|p|}$, respectively, i.e. we start the unification process with the tuple $\{C_0^M V_2^M C_4^E V_4^M = \varepsilon | C_0^M V_4^E C_5^M V_5^M\}, \{\}$ and apply the transformation rules according to Table 4:

$\{C_0^M V_2^M C_4^E V_4^M = \varepsilon | C_0^M V_4^E C_5^M V_5^M\}, \{\} \xrightarrow{\text{R3}} \{V_2^M C_4^E V_4^M = \varepsilon | V_4^E C_5^M V_5^M\}, \{\}$
$\xrightarrow{\text{R8}} \{V_4^E C_5^M V_5^M = V_2^M | C_4^E V_4^M\}, \{\} \xrightarrow{\text{R10}} \{V_4^E C_5^M V_5^M = V_2^M C_4^E | V_4^M\}, \{\}$
$\xrightarrow{\text{R9}} \{V_4^M = V' | C_5^M V_5^M\}, \{V_4^E \backslash V_2^M C_4^E V'\} \xrightarrow{\text{R10}} \{V_4^M = V' C_5^M | V_5^M\}, \{V_4^E \backslash V_2^M C_4^E V'\}$
$\xrightarrow{\text{R10}} \{V_4^M = V' C_5^M V_5^M | \varepsilon\}, \{V_4^E \backslash V_2^M C_4^E V'\}$
$\xrightarrow{\text{R5}} \{\varepsilon = \varepsilon | \varepsilon\}, \{V_4^E \backslash V_2^M C_4^E V', V_4^M \backslash V' C_5^M V_5^M\}$
$\xrightarrow{\text{R1}} \{\}, \{V_4^E \backslash V_2^M C_4^E V', V_4^M \backslash V' C_5^M V_5^M\}$

The only successful transformation sequence, leading to a tuple $\{\}, \tilde{\sigma}$, yields the substitution $\tilde{\sigma} = \{V_4^E \backslash V_2^M C_4^E V', V_4^M \backslash V' C_5^M V_5^M\}$. Applying rule R10 instead of rule R8 (which is the only nondeterministical choice) does not lead to any successful transformation sequence. Thus the only (most general) unifier is $\{\phi_{0rll}^E \backslash \phi_{0l}^M \psi_{0lrl}^E V', \phi_{0lrl}^M \backslash V' \psi_{0rlll}^M \phi_{0rlll}^M\}$ where V' is a new introduced variable.

For the fragment $\mathcal{M}?\mathcal{LL}$ of linear logic (on which we will focus in the following section of this paper) we do not need to deal with characters of type ϕ^E or ψ^E. Furthermore *all* prefixes to be unified have the form $C_1 V_1 C_2 V_2 ... C_n V_n$ (where $V_i \in \mathcal{V}$ and $C_i \in \mathcal{C}$), allowing us to drop rules R2, R4, R6, and R7 (see also [15]).

Definition 12. Let $\mathcal{V}=\Phi^M$ be a set of variables, $\mathcal{C}=\Psi^M$ be a set of constants, and $\mathcal{V}'$ be a set of *auxiliary variables* (with $\mathcal{V} \cap \mathcal{V}' = \emptyset$). The set of *transformation rules for* $\mathcal{M}?\mathcal{LL}$ is defined in Table 5.

R1.	$\{\varepsilon = \varepsilon \mid \varepsilon\},\ \sigma$	$\rightarrow$	$\{\},\ \sigma$	
R3.	$\{Xs = \varepsilon \mid Xt\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon \mid t\},\ \sigma$	
R5.	$\{Vs = z \mid \varepsilon\},\ \sigma$	$\rightarrow$	$\{s = \varepsilon \mid \varepsilon\},\ \{V \backslash z\} \cup \sigma$	
R8.	$\{Vs^+ = \varepsilon \mid V_1 t\},\ \sigma$	$\rightarrow$	$\{V_1 t = V \mid s^+\},\ \sigma$	
R9.	$\{Vs^+ = z^+ \mid V_1 t\},\ \sigma$	$\rightarrow$	$\{V_1 t = V' \mid s^+\},\ \{V \backslash z^+ V'\} \cup \sigma$	
R10.	$\{Vs = z \mid Xt\},\ \sigma$	$\rightarrow$	$\{Vs = zX \mid t\},\ \sigma$	$(V \neq X$, and $s=\varepsilon$ or $t \neq \varepsilon$ or $X \in \mathcal{C})$

$s,t,z \in (\mathcal{V} \cup \mathcal{C} \cup \mathcal{V}')^*$ denote (arbitrary) strings, $s^+, z^+ \in (\mathcal{V} \cup \mathcal{C} \cup \mathcal{V}')^+$ denote non-empty strings. X, V, and V_1 denote single characters with $X \in \mathcal{V} \cup \mathcal{C} \cup \mathcal{V}'$ and $V, V_1 \in \mathcal{V} \cup \mathcal{V}'$ (with $V \neq V_1$). $V' \in \mathcal{V}'$ is a new variable which does not occur in the substitution σ computed so far.

Table 5. Transformation rules for $\mathcal{M}?\mathcal{LL}$

5 A Tableau Prover

In this section we present an implementation of the tableau calculus for the fragment $\mathcal{M}?\mathcal{LL}$. We first present the calculus and repeat some definitions.

A Tableau Calculus for $\mathcal{M}?\mathcal{LL}$. The tableau calculus for $\mathcal{M}?\mathcal{LL}$ is similar to the calculus for $\mathcal{MELL}$ presented in Table 3. Since ? and ! can only occur, respectively, with positive and negative polarity we do not need the π-rule anymore. Because of that there are no positions of type ϕ^E or ψ^E anymore. Furthermore the ν-rules use a stepwise contraction and the τ- and ω-rules are modified. Let $\mathcal{A}_F$ be the set of all predicate symbols in the formula F. A *tableaux* for a formula F is defined as usual (see Definition 4) but with the tableau expansion rules from Table 6 where $A_p \epsilon \mathcal{A}'$ is a predicate symbol and $X_p \epsilon \mathcal{X}$ is a predicate variable. Let T be a tableau, $\sigma{:}\Phi^M \to (\Phi^M \cup \Psi^M)$ be a string substitution, and $\sigma_{\mathcal{X}}{:}\mathcal{X} \to (\mathcal{A}_F \cup \mathcal{A}')$ be a predicate substitution.

Definition 13. A *branch of T is closed* iff it contains a *complementary* connection, i.e $\{[A]^- : s\ (p),\ [B]^+ : t\ (q)\}$ where $\sigma(s){=}\sigma(t)$ and $\sigma_{\mathcal{X}}(A){=}\sigma_{\mathcal{X}}(B)$.

Let T be a tableau, $\mathcal{C}$ be a set of connections, $\boldsymbol{\alpha}_T$ and $\boldsymbol{\beta}_T$ be the set of all positions of formulas of type α and β, respectively, in T.

Definition 14. A *tableau T is closed* iff (1.) every branch of T is closed by a $c \epsilon \mathcal{C}$ under σ and $\sigma_{\mathcal{X}}$, (2.) if $\{a, b\} \epsilon \mathcal{C}$ and $\{a, c\} \epsilon \mathcal{C}$ then $b{=}c$ *(linearity)*, (3.) $2|\mathcal{C}| = |\boldsymbol{\alpha}_T| + |\boldsymbol{\beta}_T| + 1$ *(relevance)*, and (4.) $2|\mathcal{C}| = 2|\boldsymbol{\beta}_T| + 2$ *(cardinality)*.

Theorem 15 (Correctness & Completeness). *A formula F is valid in the fragment $\mathcal{M}?\mathcal{LL}$ iff there is a closed tableau for the prefixed formula $F^+ : \psi_0\ (0)$.*

Proof (Sketch). We show that our calculi for $\mathcal{M}?\mathcal{LL}$ and $\mathcal{MELL}$ (without using rule π) are equivalent for the fragment of $\mathcal{M}?\mathcal{LL}$: both ν-rules are equivalent (consider an appropriate multiplicity μ); rules τ and ω are correct and complete, i.e $\mathbf{1}^+ \equiv \bot^- \equiv (A \multimap A)^+$ for a new predicate symbol A and $\bot^+ \equiv \mathbf{1}^- \equiv (X \multimap X)^-$ for an arbitrary predicate symbol X (so that one-element connections are omited and the weakening map is empty), since both $\mathbf{1}^+/\bot^-$ and $(A \multimap A)^+$ lead to leafs in the sequent proof and the rules for $\bot^+/\mathbf{1}^-$ can always applied at the leafs in the sequent proof; linearity and cardinality conditions are identical (with empty weakening map); if $\mathcal{C}$ is linear and $2|\mathcal{C}|{=}|\boldsymbol{\alpha}_T|{+}|\boldsymbol{\beta}_T|{+}1$ then every atomic formula occurs in $\mathcal{C}$ (relevance condition), since the number of leaves in the (binary) formula tree is equal to the number of inner nodes plus one.

The linTAP Implementation. The previous calculus has been implemented in Prolog (see Table 7). For the syntax of formulas we use the logical connectives "`~`" (negation $^\bot$), "`*`" (conjunction $\otimes$), "`@`" (disjunction $\parr$), "`@>`" (implication $\multimap$), the exponentials "`?`" and "`!`", the constants "`1`" (for $\mathbf{1}$) and "`0`" (for $\bot$), and Prolog atoms for atomic formulas. For example to express the formula $((A \parr \bot) \otimes !A) \parr ?(A^\bot)$ we use the Prolog term `((a@0)*!a)@ ?(~a).`

We use `0` and `1` to present the polarities $+$ and $-$, respectively. Positions are constructed from right to left and prefixes are represented by Prolog lists. Like in leanTAP we use two predicates for path checking: `fml` and `prove`.

α-rules

$$\frac{(F_1 \parr F_3)^+ \;:\; s \qquad (p)}{\begin{array}{ll} F_1^+ & : s \circ s' \;\; (p \circ l) \\ F_3^+ & : s \circ s' \;\; (p \circ r) \end{array}} \qquad \frac{(F_1 \otimes F_3)^- \;:\; s \qquad (p)}{\begin{array}{ll} F_1^- & : s \circ s' \;\; (p \circ l) \\ F_3^- & : s \circ s' \;\; (p \circ r) \end{array}}$$

$$\frac{(F_1 \multimap F_3)^+ \;:\; s \qquad (p)}{\begin{array}{ll} F_1^- & : s \circ s' \;\; (p \circ l) \\ F_3^+ & : s \circ s' \;\; (p \circ r) \end{array}} \qquad s' = \begin{cases} \psi_p \;, & \text{if } \; s = \tilde{s} \circ \phi, \;\; \phi \in \Phi^M \\ \varepsilon \;, & \text{else} \end{cases}$$

β-rules

$$\frac{(F_1 \parr F_2)^- \;:\; s \;\; (p)}{F_1^- : s \circ s' \;\; (p \circ l) \;\big|\; F_2^- : s \circ s' \;\; (p \circ r)} \qquad \frac{(F_1 \otimes F_2)^+ \;:\; s \;\; (p)}{F_1^+ : s \circ s' \;\; (p \circ l) \;\big|\; F_2^+ : s \circ s' \;\; (p \circ r)}$$

$$\frac{(F_1 \multimap F_2)^- \;:\; s \;\; (p)}{F_1^+ : s \circ s' \;\; (p \circ l) \;\big|\; F_2^- : s \circ s' \;\; (p \circ r)} \qquad s' = \begin{cases} \phi_p \;, & \text{if } \; s = \tilde{s} \circ \psi, \;\; \psi \in \Psi^M \\ \varepsilon \;, & \text{else} \end{cases}$$

ν-rules

$$\frac{(!F_1)^- \colon\; s \;\; (p)}{\bot^- \;:\; s \;\; (p)} \qquad \frac{(!F_1)^- \colon\; s \qquad (p)}{\begin{array}{ll} F_1^- & : s \circ s' \;\; (p \circ l) \\ (!F_1)^- \colon & s \circ s' \;\; (p \circ r) \end{array}} \qquad \frac{(?F_1)^+ \colon\; s \;\; (p)}{\bot^+ \;:\; s \;\; (p)} \qquad \frac{(?F_1)^+ \colon\; s \qquad (p)}{\begin{array}{ll} F_1^+ & : s \circ s' \;\; (p \circ l) \\ (?F_1)^+ \colon & s \circ s' \;\; (p \circ r) \end{array}}$$

$$s' = \begin{cases} \psi_p \;, & \text{if } \; s = \tilde{s} \circ \phi, \;\; \phi \in \Phi^M \\ \varepsilon \;, & \text{else} \end{cases}$$

o-rules

$$\frac{(F_1{}^{\bot})^+ \;:\; s \;\; (p)}{F_1^- \;:\; s \;\; (p \circ l)} \qquad \frac{(F_1{}^{\bot})^- \;:\; s \;\; (p)}{F_1^+ \;:\; s \;\; (p \circ l)}$$

τ-rules

$$\frac{\bot^- \;:\; s \;\; (p)}{(A_p \multimap A_p)^+ \;:\; s \;\; (p \circ l)} \qquad \frac{\mathbf{1}^+ \;:\; s \;\; (p)}{(A_p \multimap A_p)^+ \;:\; s \;\; (p \circ l)}$$

ω-rules

$$\frac{\mathbf{1}^- \;:\; s \;\; (p)}{(X_p \multimap X_p)^- \;:\; s \;\; (p \circ l)} \qquad \frac{\bot^+ \;:\; s \;\; (p)}{(X_p \multimap X_p)^- \;:\; s \;\; (p \circ l)}$$

***a(tom)*-rules**

$$\frac{A^+ \;:\; s \qquad (p)}{[A]^+ \colon\; s \circ s' \circ \phi_p \;\; (p)} \qquad \frac{A^- \;:\; s \qquad (p)}{[A]^- \colon\; s \circ s' \circ \phi_p \;\; (p)} \qquad s' = \begin{cases} \psi_p \;, & \text{if } \; s = \tilde{s} \circ \phi, \;\; \phi \in \Phi^M \\ \varepsilon \;, & \text{else} \end{cases}$$

Table 6. A prefixed-based tableau calculus for $\mathcal{M}?\mathcal{LL}$

`fml(F,Pol,P,F1,F2,F3,PrN,Ctr)` is used to specify the rules of our prefix-based tableau calculus. It succeeds if there is a rule to expand the formula `F`. `Pol` is the polarity, `P`, `F1`, `F2`, `F3`, and `PrN` are the position p, formulas F_1, F_2, F_3, and the new prefix character s', respectively. `Ctr` is bound to `c` if a contraction (rule) is applied. According to our calculus we need 18 clauses to specify all rules.

```
%%% Specification of Tableau Rules

fml(A,   Pol, P,[A,Pol],[],      [],[P],0):-var(A).
fml((A@B), 0, P, (A,0), [],   (B,0),[P],0).          fml((A@B), 1, _, (A,1), (B,1),[],[_],0).
fml((A*B), 1, P, (A,1), [],   (B,1),[P],0).          fml((A*B), 0, _, (A,0), (B,0),[],[_],0).
fml((A@>B),0, P, (A,1), [],   (B,0),[P],0).          fml((A@>B),1, _, (A,0), (B,1),[],[_],0).
fml((!A),  1, P, (A,1), [],((!A),1),[P],c).          fml((!_),  1, _, (1,1), [],   [], [],0).
fml((?A),  0, P, (A,0), [],((?A),0),[P],c).          fml((?_),  0, _, (0,0), [],   [], [],0).
fml(0,     1, P, (P,0), [],   (P,1),[P],0).          fml(1,     1, _, (X,0), (X,1),[],[_],0).
fml(1,     0, P, (P,0), [],   (P,1),[P],0).          fml(0,     0, _, (X,0), (X,1),[],[_],0).
fml((~A),  0, _, (A,1), [],      [], [],0).          fml((~A),  1, _, (A,0), [],   [], [],0).
fml(A,   Pol, P,[A,Pol],[],      [],[P],0).

%%% Path Checking

prove([(F,Pol),Pre,P],UnExp,Lits,Exp,ExpLim,PU,At,Bt,C,C1) :-
   fml(F,Pol,P,F1,F2,F3,PrN,Ctr), append(_,[Lp],Pre),                  % look up tableau rule
   ( Ctr=c -> Exp<ExpLim, Exp1 is (Exp+1) ; Exp1=Exp, ! ),             % control contraction
   ( (F2\=[],var(Lp);F2=[],\+var(Lp)) -> Pre1=Pre;append(Pre,PrN,Pre1) ),% Pre1 is new prefix
   ( F3=[] -> UnExp1=UnExp, At=At3;
              UnExp1=[[F3,Pre1,r(P)]|UnExp], At=[P|At3] ),             % update UnExp1
   prove([F1,Pre1,l(P)],UnExp1,Lits,Exp1,ExpLim,PU1,At1,Bt1,C,C2),     % continue with F1
   ( F2=[] -> PU=PU1, At3=At1, Bt=Bt1, C1=C2 ;
     prove([F2,Pre1,r(P)],UnExp1,Lits,Exp1,ExpLim,PU2,At2,Bt2,C2,C1),  % continue with F2
     append(PU1,PU2,PU), union(At1,At2,At3), union([P|Bt1],Bt2,Bt) ).

prove([[Lit,Pl],Pr,P],_,[[[L,Pl1],Pr1,P1]|Lits],_,_,PU,At,Bt,C,C1) :-  % close branch
   ( Lit=L, Pl is 1-Pl1, Lit=L,  At=[], Bt=[],                 % connection found ?
     (member([P,S],C)  -> (S=P1 -> C1=C, PU=[]) ;              % relevance condition
     (member([P1,S],C) -> (S=P  -> C1=C, PU=[]) ;
      PU=[[Pr,_]=[Pr1,_]], C1=[[P,P1],[P1,P]|C])) ) ;          % add prefixes and connection
   prove([[Lit,Pl],Pr,P],[],Lits,_,_,PU,At,Bt,C,C1).           % otherwise check next literal
prove(Lit,[Next|UnExp],Lits,Exp,ExpLim,PU,At,Bt,C,C1) :-       % add Lit to current branch
   prove(Next,UnExp,[Lit|Lits],Exp,ExpLim,PU,At,Bt,C,C1).      % expand Next formula

%%% T-String Unification

t_string_unify([]).
t_string_unify([S=T|G]):- flatten(S,S1,[]), flatten(T,T1,[]),        % flatten prefix lists
                          tunify(S1,[],T1), t_string_unify(G).       % solve first equation
tunify([],[],[]).                                                    % transfor. rule R1
tunify([X1|S],[],[X2|T])      :- X1==X2, !, tunify(S,[],T).          %      -''-      R3
tunify([V|S],Z,[])            :- V=Z, tunify(S,[],[]).               %      -''-      R5
tunify([V,X|S],[],[V1|T])     :- var(V1), tunify([V1|T],[V],[X|S]).  %      -''-      R8
tunify([V,X|S],[Z1|Z],[V1|T]) :- var(V1), append([Z1|Z],[Vnew],V),    %      -''-      R9
                                 tunify([V1|T],[Vnew],[X|S]).
tunify([V|S],Z,[X|T])         :- (S=[]; T\=[]; \+var(X)) ->          %      -''-      R10
                                 append(Z,[X],Z1), tunify([V|S],Z1,T).

flatten(A,[A|B],B) :- (var(A); A\=[], A\=[_|_]), !.                  % flatten list
flatten([],A,A).
flatten([A|B],C,D) :- flatten(A,C,E), flatten(B,E,D).
```

Table 7. The source code of linTAP

`prove([(F,Pol),Pre,P],UnExp,Lits,Exp,ExpLim,PU,At,Bt,C,C1)` performs the actual proof search. `(F,Pol)` is the formula currently expanded, `Pre` its prefix s, `P` its position p. `UnExp` and `Lits` represent lists of formulas not yet expanded and the atomic formulas on the current branch of the tableau. `Exp` is the number of contractions on the current branch, `ExpLim` the maximum number of contractions allowed on a branch, `PU` a list of prefix equations, `At` and `Bt` represent the sets $\boldsymbol{\alpha}_T$ and $\boldsymbol{\beta}_T$, and `C` represents the current set of connections $\mathcal{C}$ (more precisely each connection is stored twice, i.e. `C`$=\bigcup_{\{p,q\}\in\mathcal{C}}[[p,q],[q,p]]$).

After a tableau has been found the prefix equations in `PU` have to be solved. This is done by `t_string_unify(PU)` and `tunify(S,[],T)` where `PU` is a system of string equations to be unified, `S` and `T` are two strings to be unified (see [22]).

The following goal succeeds if the formula `F` is valid in $\mathcal{M}?\mathcal{LL}$ using not more than `ExpLim` contractions on each branch:

```
prove([(F,0),[0],0],[],[],0,ExpLim,PU,At,Bt,[],C),
      length(Bt,Nbt), length(C,Nc), Nc=:=(2*Nbt)+2,       % cardinality
      length(At,Nat), Nc=:=Nat+Nbt+1,                     % linearity
      t_string_unify(PU).                    % unifiability of prefixes
```

Some experimental results. There are only a few provers and even fewer examples available for $\mathcal{M}?\mathcal{LL}$. Table 8 contains some problems for $\mathcal{M}?\mathcal{LL}$. The timings for these (valid) formulas are given in Table 9.[2] We compare linTAP[3] (with iterative deepening) with the sequent calculus provers llprover (implemented in Prolog; see [24]) and linseq, and with the resolution prover linres (both last-mentioned provers are implemented in Scheme and compiled to C; see [23]).

F_1	$((A \parr \bot) \otimes A) \parr ?(A^\bot)$
F_2	$!(A \otimes C \multimap B) \otimes !(B \otimes D \otimes D \multimap C \otimes D) \otimes A \otimes C \otimes D \otimes D \multimap C \otimes D$
F_3	$(C_1^\bot \otimes (C_2^\bot \otimes (\ldots (C_{11}^\bot \otimes C_{12}^\bot)..))) \parr (C_{12} \parr (C_{11} \parr \ldots \parr (C_2 \parr C_1)..))$
F_4	$(C_1^\bot \otimes (C_2^\bot \otimes (\ldots (C_{11}^\bot \otimes C_{12}^\bot)..))) \parr ((C_{12} \otimes (D_{12} \parr D_{12}^\bot)) \parr$ $((C_{11} \otimes (D_{11} \parr D_{11}^\bot)) \parr \ldots \parr ((C_2 \otimes (D_2 \parr D_2^\bot)) \parr (C_1 \otimes (D_1 \parr D_1^\bot)))..))$
F_5	$D \otimes !(D \multimap C \otimes Q) \otimes !(D \otimes Q \otimes Q \multimap I) \multimap C \otimes ?Q$
F_6	$D \otimes D \otimes D \otimes !(D \multimap C \otimes Q) \otimes !(D \otimes Q \otimes Q \multimap I) \multimap C \otimes I \otimes C$
F_7	$D \otimes D \otimes D \otimes D \otimes Q \otimes Q \otimes !(D \multimap C \otimes Q) \otimes !(D \otimes Q \otimes Q \multimap I) \multimap C \otimes I \otimes C \otimes I$
F_8	$D \otimes D \otimes D \otimes Q \otimes Q \otimes Q \otimes !(D \multimap C \otimes Q) \otimes !(D \otimes Q \otimes Q \multimap I) \multimap C \otimes I \otimes X$

Table 8. Some problems for $\mathcal{M}?\mathcal{LL}$

F_i	llprover	linseq	linres	linTAP
F_1	0.08	0.02	0.02	< 0.01
F_2	61.05	0.17	0.05	0.03
F_3	–	0.33	0.08	0.05
F_4	–	–	–	0.28

F_i	llprover	linseq	linres	linTAP
F_5	0.95	0.03	0.03	< 0.01
F_6	–	0.15	0.05	0.13
F_7	–	4.63	0.07	13.67
F_8	n/a	n/a	n/a	11.75

Table 9. Timings for the problems from Table 8

[2] Measured on a Sun SPARC10 in seconds; "–" means that no proof was found within 100 seconds. F_2 is the most difficult example from [24] (linTAP solves all other problems from [24] in less than 30ms); F_3 is from [23]. The predicates in F_5 to F_8 can be interpreted as follows: D="dollar", Q="quarter", C="Coke", I="ice-cream". Formula F_6, e.g., then expresses the following situation: for one dollar I can buy a Coke and get a quarter back, and for one dollar and two quarters I can buy an ice-cream; so if I have three dollars I can buy two Cokes and one ice-cream (see [5] for a similar approach on deductive planning). It is possible to use (free) variables: formula F_8 contains a variable X which will be bound to an appropriate predicate symbol (i.e. I) to make the formula valid (only linTAP offers this feature).

[3] The linTAP implementation uses an additional technique to simplify formulas of type α of the form $F \odot \omega$ or $\omega \odot F$ replacing them by F (where $\odot \epsilon \{\parr, \otimes, \multimap\}$).

6 Conclusion

We have presented prefix-based tableau calculi for $\mathcal{MELL}$ and $\mathcal{M}?\mathcal{LL}$. We encoded the additional non-permutabilities arising in multiplicative linear logic by an additional string unification. These calculi are the basis for our tableau prover linTAP. linTAP is not only a very compact implementation but compares favorable with other (larger) implementations. Due to the compact code the program can easily be modified for special purposes or applications.

Future work include the extension to larger fragments of linear logic and the comparison of linTAP with a connection driven proof search procedure (see [15]).

Besides the original leanTAP implementation for classical first-order logic, lean tableau provers are also available for various non-classical logics, i.e. first-order intuitionistic logic (ileanTAP, [21]), and the propositional modal logics K, KD, KT, and S4 (ModLeanTAP, [4]). The linTAP implementation fills the gap for the multiplicative linear logic. The source code of linTAP can be obtained at `http://www.intellektik.informatik.tu-darmstadt.de/~jeotten/linTAP/`.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
2. B. Beckert, J. Posegga. leanTAP: Lean Tableau-Based Theorem Proving. *12^{th} Conference on Automated Deduction*, LNAI 814, pp. 793–797. Springer, 1994.
3. B. Beckert, J. Posegga. leanTAP: Lean Tableau-Based Deduction. *Journal of Automated Reasoning*, 15 (3), pp. 339–358, 1995.
4. B. Beckert, R. Goré. Free Variable Tableaux for Propositional Modal Logics. *Proc. 6^{th} TABLEAUX Conference*, LNAI 1227, pp. 91–106, Springer 1997.
5. W. Bibel. Let's plan it deductively!. In *IJCAI-97*, Morgan Kaufmann, 1997.
6. I. Cervesato, J.S. Hodas, F. Pfenning. Efficient resource management for linear logic proof search. In *Extensions of Logic Programming*, LNAI 1050, pages 67–81. Springer, 1996.
7. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 1996.
8. B. Fronhöfer. *The action-as-implication paradigm*. CS Press, 1996.
9. D. Galmiche. Connection methods in linear logic fragments and proof nets. Technical report, CADE–13 workshop on proof search in type-theoretic languages, 1996.
10. D. Galmiche & G. Perrier. On proof normalization in linear logic. *TCS*, 135:67–110, 1994.
11. V. Gehlot and C. Gunter. Normal process representatives. *Sixth Annual Symposium on Logic in Computer Science*, pages 200–207, 1991.
12. J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
13. J. Harland and D. Pym. Resource-Distribution via Boolean Constraints. *14^{th} Conference on Automated Deduction*, LNCS 1249, pp. 222–236. Springer, 1997.
14. J.S. Hodas & D. Miller. Logic programming in a fragment of linear logic. *Journal of Information and Computation*, 110(2):327–365, 1994.
15. C. Kreitz, H. Mantel, J. Otten, S. Schmitt. Connection-Based Proof Construction in Linear Logic. *14^{th} Conference on Automated Deduction*, LNCS 1249, pp. 207–221. Springer, 1997.
16. P. Lincoln and T. Winkler. Constant-only multiplicative linear logic is NP-complete. *TCS*, 135:155–169, 1994.
17. H. Mantel, C. Kreitz. A Matrix Characterization for MELL. *Logics in Artificial Intelligence*, JELIA '98, LNAI 1489, pp. 169–183, Springer, 1998.
18. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM TOPLAS*, 4:258–282, 1982.
19. M. Masseron, C. Tollu, J. Vauzeilles. Generating plans in linear logic. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS, Springer,1991.
20. D. Miller. FORUM: A Multiple-Conclusion Specification Logic. *TCS*, 165(1):201-232, 1996.
21. J. Otten. ileanTAP: An Intuitionistic Theorem Prover. *Proc. 6^{th} TABLEAUX Conference*, LNAI 1227, pp. 307–312, Springer 1997.
22. J. Otten, C. Kreitz. T-String-Unification: Unifying Prefixes in Non-Classical Proof Methods. *Proc. 5^{th} TABLEAUX Workshop*, LNAI 1071, pp. 244–260, 1996.
23. T. Tammet. Proof strategies in linear logic. *JAR*, 12:273–304, 1994.
24. N. Tamura. User's Guide of a Linear Logic Theorem Prover (llprover). Technical report. Faculty of Engineering, Kobe University, Japan, 1995.
25. L. Wallen. *Automated deduction in nonclassical logic*. MIT Press, 1990.