Journal of Automated Deduction manuscript No.
(will be inserted by the editor)

Automated Checking of Proof Transformations

Vivek Nigam · Giselle Reis · Leonardo Lima

Received: date / Accepted: date

Abstract The proof of many foundational results in structural proof theory, such as the admissibility of the cut rule and the completeness of the focusing discipline, rely on permutation lemmas. It is often a tedious and error prone task to prove such lemmas as they involve many cases. Indeed, some cut-elimination results in the literature had to be corrected, even withdrawn, due to missing cases of needed permutation lemmas. This paper proposes an automated method to prove permutation lemmas. Proof systems are specified in a linear logical framework with subexponentials. From these specifications, we extract logic programs to enumerate all cases involved in the proof of a permutation lemma, and to check which cases are satisfied and which cannot be inferred to be satisfied. Finally, we print all cases in a reader friendly format (using LaTeX) very close to figures appearing in proof theory textbooks. This work is implemented as the tool Quati and tested for a number of proof systems: linear logic, LJ, LK, MLJ, S4, among others.

Keywords Sequent calculus · Permutations · Linear logic · Answer set programming

1 Introduction

Permutation lemmas play an important role in proof theory. Many foundational results about proof systems rely on the fact that some rules permute over others. For instance, permutation lemmas are used in Gentzen-style cut-elimination proofs [8], the completeness proof of focusing disciplines [1,15], and the proof of Herbrand's

Vivek Nigam
Federal University of Paraíba, Brazil
E-mail: vivek.nigam@gmail.com

Giselle Reis
Carnegie Mellon University, Qatar
E-mail: giselle@cmu.edu

Leonardo Lima
Federal University of Paraíba, Brazil
E-mail: leonardo.alfs@gmail.com

theorem [10]. Proving permutation lemmas, however, is often a tedious and error-prone task as there may be many cases to consider. Take the case of permuting $\wedge_l$ over $\rightarrow_l$ in the intuitionistic calculus LJ. To show whether these two rules permute, one needs to check *every* case in which $\rightarrow_l$ occurs above $\wedge_l$ in a derivation. When using a multiplicative calculus, there are four possible derivations: two allow a permutation while the other two do not. Here's one of each:

$$\dfrac{\dfrac{\varphi_1 \quad \varphi_2}{\Gamma \vdash A \quad \Gamma', P, Q, B \vdash F}\,\rightarrow_l}{\Gamma, \Gamma', A \rightarrow B, P, Q \vdash F}\,\wedge_l \quad \rightsquigarrow \quad \dfrac{\varphi_2}{\dfrac{\varphi_1 \quad \Gamma', P, Q, B \vdash F}{\Gamma \vdash A \quad \Gamma', P \wedge Q, B \vdash F}\,\wedge_l}\,\rightarrow_l$$

$$\dfrac{\dfrac{\varphi_1 \quad \varphi_2}{\Gamma, P \vdash A \quad \Gamma', Q, B \vdash F}\,\rightarrow_l}{\Gamma, \Gamma', A \rightarrow B, P, Q \vdash F}\,\wedge_l \quad \rightsquigarrow \quad ?$$

The combinatorial nature of permutation lemmas can be observed in this example. While there are "only" four cases to consider for this pair of rules, for proving completeness of the focusing discipline, for example, one needs to study which permutations are allowed and therefore all pairs of rules are considered [15]. Moreover, the fact that the cases are rarely documented makes it hard to check the correctness of the transformations. For instance, the cut-elimination result for bi-intuitionistic logic given by Rauszer [26] was later found to be incorrect [4] exactly because one of the permutation lemmas was not true.

This paper improves the state-of-the-art by proposing an automated method, *i.e.*, with a click of a button, to check for a given proof system which permutation lemmas are true. Our method has been implemented in the tool Quati. Quati[1] not only returns yes/no answers, it prints, in a user readable format similar to proof theory textbooks [27], the cases for which it was able to check that the permutation holds and the cases for which it was not able to check it.

Proof systems are specified in the linear logical framework with subexponentials (SELL) [20,23]. As demonstrated in our previous work, a large number of proof systems can be specified in SELL (e.g. LL, LJ, LK, MLJ and S4). Moreover, all these specifications are adequate on the level of derivations [21]. This means that there is a one-to-one correspondence between the rules of the given proof system and the focused derivations of its SELL specification. From a given SELL specification, we extract logic programs [7] which are used for two purposes:

1. **Enumerate Derivations:** Given a (schema-)sequent $\mathcal{S}$ and an inference rule ρ, we construct a logic program that enumerate all possible ways of applying ρ to $\mathcal{S}$. Our procedure is sound and complete in the sense that each *answer set* for the program corresponds to exactly one instance of ρ's application.
2. **Infer Provability:** For two given (schema-)sequents, $\mathcal{S}_1$ and $\mathcal{S}_2$, the second program checks whether the sequent $\mathcal{S}_2$ is provable, when assuming that $\mathcal{S}_1$ is also provable. The answer is positive if the logic program has at least one answer set. As this problem is undecidable in general, we show the soundness of our method.

[1] Quati is a mammal from the raccoon family native to South America. Its name comes from Tupi-guarani, a language spoken by Native Indians in Brazil, and means "long nose".

$$\frac{}{\vdash a^{\perp}, a}\ I \qquad \frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{}{\vdash 1}\ 1 \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp}\ \perp$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B}\ \& \qquad \frac{}{\vdash \Gamma, \top}\ \top \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \oplus_2$$

Fig. 1: Linear logic introduction rules without the (sub)exponentials.

One main advantage of using answer set programming is that it enables the use of powerful off-the-shelf answer set solvers [16,12]. For our implementation in particular, we have used DLV [12].

Finally, we also propose a procedure to extract reader friendly figures, similar to the ones used in proof theory textbooks [27]. The system will print, using the object logic rules, the permutations cases it was able to infer and those that failed.

This paper is organized as follows. We start by reviewing how Linear Logic with Subexponentials can be used as a framework for proof systems in Section 2. Then in Section 3 we propose to use answer-set programs to enumerate all possible derivations involving inference rules and in Section 4 we demonstrate how to check for the existence of permutations. Section 3 show how to extract reader friendly figures for permutations and Section 6 briefly describes our implementation Quati. Finally, we finish in Sections 7 and Section 8 by discussing related and future work.

2 Linear Logical Framework with Subexponentials

Linear logic with subexponentials (SELL) [20] has been used [23] to specify a wide range of proof systems with complicated structural properties, such as the multi-conclusion proof system for intuitionistic logic (MLJ), modal logics (S4) among others. Moreoever, these specifications have a very strong level of adequacy, namely on the level of derivations [21], which means that a partial derivation (or a single rule) in the object system corresponds to exactly one derivation in SELL. We review in this section the (focused) proof system SELL and how proof systems are specified. For further details, see [23].

2.1 SELL

We briefly review some of LL's basic concepts. The multiplicative and additive fragment (MALL) has two types of conjunction and disjunction: the multiplicative ones, $\otimes$ and $\oplus$, and the additive ones $\&$ and $\parr$. Moreover, MALL has two units 1 and $\perp$. Their introduction rules are depicted in Figure 1 including the identity.

Contraction and weakening are controlled in LL by the so called exponentials with introduction rules:[2]

$$\frac{\Gamma, ?F, ?F}{\vdash \Gamma, ?F} \qquad \frac{\Gamma}{\vdash \Gamma, ?F} \qquad \frac{\Gamma, F}{\vdash \Gamma, ?F} \qquad \frac{?\Gamma, F}{\vdash ?\Gamma, !F}$$

[2] Exchange rule is implicit as we consider sequents to contain multiset of formulas.

However, as already known to Girard in his original LL paper [9,5], the exponentials are not canonical in the following sense. If we admit labelled (or colored versions) of exponentials $!^r, ?^r$ and $!^b, ?^b$ and add the corresponding colored rules, then it is not possible to prove the equivalence of the same formula, F, when marked with different exponentials:

$$?^r F \equiv ?^b F \qquad \text{and} \qquad !^r F \equiv !^b F$$

where $F \equiv G$ is defined as usual $(F^{\perp} \parr G) \otimes (F \parr G^{\perp})$. This means that linear logic admits many labelled exponentials, called *subexponentials* [20].

While linear logic with subexponentials (SELL) is still linear logic, SELL is considerably more expressive than LL allowing the specification of algorithmic specifications [20], a wider range of proof systems [23], a general framework for linear authorization logics [18,19], and also for the specification of Concurrent Constraint Programming language with modal operators [22,24].

The key idea is that with subexponentials, the formulas in a SELL sequent can be organized into multiple contexts, $\vdash ?^{s_1}\Theta_1, ?^{s_2}\Theta_2, \ldots, ?^{s_n}\Theta_n, \Gamma$, whereas the formulas in LL sequents can only be organized into two contexts, $\vdash ?\Theta, \Gamma$ (unbounded and bounded formulas) [1]. This allows a finer control of the contexts using SELL logical connectives.

Formally, SELL is a family of logics, each specified by a subexponential signature $\Sigma = \langle I, \preceq, U\rangle$, where I is the set of subexponential names, $U \subseteq I$ are the subexponentials that allow for weakening and contraction, and $\preceq$ is a pre-order on the elements of I upwardly closed with respect to U, *i.e.*, if $s_1 \in U$ and $s_1 \preceq s_2$, then $s_2 \in U$. Given such a subexponential signature, SELL includes the rules in Figure 1 and the following rules for each $s, s_1, \ldots, s_n \in I$ and $u \in U$:

$$\frac{\vdash \Theta, F}{\vdash \Theta, ?^s F}\ ?^s \qquad \frac{\vdash \Theta, ?^u F, ?^u F}{\vdash \Theta, ?^u F}\ C \qquad \frac{\vdash \Theta}{\vdash \Theta, ?^u F}\ W \qquad \frac{\vdash ?^{s_1}\Theta_1, \ldots, ?^{s_n}\Theta_n, F}{\vdash ?^{s_1}\Theta_1, \ldots, ?^{s_n}\Theta_n, !^s F}\ !^s\star$$

where $\star$ in the $!^s$ rule is the side condition: for all $1 \leq i \leq n$, $s_i \preceq s$.

Notice that SELL allows for the specification of an arbitrary number of subexponential labels, for which some may allow contraction and weakening and the remaining not. This means that formulas may be marked with subexponentials, $?^s F$, and behave in a bounded or unbounded fashion depending on whether s belongs or not to U in the subexponential signature Σ.

Finally, the $!^s$ rule's side condition specifies that a formula $!^s F$ can only be introduced if the context has formulas marked with $?^{s_i}$ such that $s_i \preceq s$. In our previous work, we use this feature for the specification of a number of structural restriction on proof systems and computational systems and languages. We review this in more detail in Section 2.3.

2.2 SELLF – Linear Logical Framework with Subexponentials

We now revisit SELLF: the focused proof system for SELL. Focusing [1], first introduced for linear logic, is a discipline where proofs are organized into two alternating phases. The negative phase where all invertible rules are applied eagerly and positive phase where non-invertible rules are applied to a chosen formula and its sub-formulas exaustively.

To introduce focusing formally, we first classify as negative the formulas whose main connective is $⅋, \&, \forall$, the formulas $\bot, \top$ and atomic formulas. The remaining formulas are classified as positive. Figure 2 contains the focused proof system SELLF, which is a rather straightforward generalization of Andreoli's original system. There are two kinds of arrows in this proof system. Sequents with $\Downarrow$ belong to the *positive* phase and introduce the logical connective of the "focused" formula (the one to the right of the arrow): building proofs of such sequents may require non-invertible proof steps to be taken. Sequents with $\Uparrow$ belong to the *negative* phase and decompose the formulas on the right in such a way that only invertible inference rules are applied. The structural rules $D_1, D_l, R\Uparrow$, and $R\Downarrow$ make the transition between a negative and a positive phase.

Similarly as in the usual presentation of linear logic, there is a pair of contexts to the left of $\Uparrow$ and $\Downarrow$ of sequents, written here as $\mathcal{K} : \Gamma$. The second context, Γ, collects the formulas whose main connective is not a question-mark, behaving as the bounded context in linear logic. But differently from linear logic, where the first context is a multiset of formulas whose main connective is a question-mark, we generalize $\mathcal{K}$ to be an *indexed context*, which is a mapping from each index in the set I (for some given and fixed subexponential signature) to a finite multiset of formulas, in order to accommodate for more than one subexponential in SELLF. In Andreoli's focused system for linear logic, the index set contains a single subexponential, ∞, and $\mathcal{K}[\infty]$ contains the set of unbounded formulas. Figure 3 contains different operations used in such indexed contexts. For example, the operation $(\mathcal{K}_1 \otimes \mathcal{K}_2)$, used in the tensor rule, specifies the resulting indexed context obtained by merging two contexts $\mathcal{K}_1$ and $\mathcal{K}_2$.

Focusing allows the composition of a collection of inference rules of the same polarity into a "macro-rule." Consider, for example, the formula $N_1 \oplus N_2 \oplus N_3$, where all N_1, N_2, and N_3 are negative formulas. Once focused on, the only way to introduce such a formula is by using a "macro-rule" of the form:

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow N_i}{\vdash \mathcal{K} : \Gamma \Downarrow N_1 \oplus N_2 \oplus N_3}$$

where $i \in \{1, 2, 3\}$. In this paper, we will encode proof systems in SELLF in such a way that the "macro-rules" of each formula in the specification matches exactly one of the inference rules of the encoded systems.

Finally, to improve readability, we will often show explicitly the formulas appearing in the image of the indexed context, $\mathcal{K}$, of a sequent. For example, if the set of subexponential indexes is $\{x_1, \ldots, x_n\}$, then the following sequent

$$\vdash \Theta_1 \stackrel{:}{x_1} \Theta_2 \stackrel{:}{x_2} \cdots \Theta_n \stackrel{:}{x_n} \Gamma \Uparrow L$$

denotes the SELLF sequent $\vdash \mathcal{K} : \Gamma \Uparrow L$, such that $\mathcal{K}[x_i] = \Theta_i$ for all $1 \leq i \leq n$. We will also assume the existence of a maximal unbounded subexponential called ∞, which is greater than all other subexponentials. This subexponential is used to mark the linear logic specification of proof systems explained in the next section.

Nigam [17] proves the following completeness of SELLF.

Theorem 1 *For any subexponential signature Σ, SELLF is sound and complete with respect to SELL.*

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A \quad \vdash \mathcal{K} : \Gamma \Uparrow L, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \& B}\ \&
\qquad
\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A, B}{\vdash \mathcal{K} : \Gamma \Uparrow L, A \parr B}\ \parr
\qquad
\frac{}{\vdash \mathcal{K} : \Gamma \Uparrow L, \top}\ \top$$

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, \bot}\ \bot
\qquad
\frac{\vdash \mathcal{K} : \Gamma \Uparrow L, A\{c/x\}}{\vdash \mathcal{K} : \Gamma \Uparrow L, \forall x.A}\ \forall
\qquad
\frac{\vdash \mathcal{K} +_l A : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, ?^l A}\ ?^l$$

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow A_i}{\vdash \mathcal{K} : \Gamma \Downarrow A_1 \oplus A_2}\ \oplus_i
\qquad
\frac{\vdash \mathcal{K}_1 : \Gamma \Downarrow A \quad \vdash \mathcal{K}_2 : \Delta \Downarrow B}{\vdash \mathcal{K}_1 \otimes \mathcal{K}_2 : \Gamma, \Delta \Downarrow A \otimes B}\ \otimes (\mathcal{K}_1 = \mathcal{K}_2)|_{\mathcal{U}}
\qquad
\frac{\vdash \mathcal{K} : \Gamma \Downarrow A\{t/x\}}{\vdash \mathcal{K} : \Gamma \Downarrow \exists x.A}\ \exists$$

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow 1}\ 1 (\mathcal{K}[\mathcal{I} \setminus \mathcal{U}] = \emptyset)
\qquad
\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l A}\ !^l (\mathcal{K}[\{x \mid l \not\preceq x \wedge x \notin \mathcal{U}\}] = \emptyset)$$

$$\frac{}{\vdash \mathcal{K} : \Gamma \Downarrow a^\perp}\ I, a \in \Gamma \cup \mathcal{K}[\mathcal{I}] \text{ and } \Gamma \cup \mathcal{K}[\mathcal{I} \setminus \mathcal{U}] \subseteq \{a\}$$

$$\frac{\vdash \mathcal{K} +_l P : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ D_l (l \in \mathcal{U})
\qquad
\frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} +_l P : \Gamma \Uparrow \cdot}\ D_l (l \notin \mathcal{U})
\qquad
\frac{\vdash \mathcal{K} : \Gamma \Downarrow P}{\vdash \mathcal{K} : \Gamma, P \Uparrow \cdot}\ D_1$$

$$\frac{\vdash \mathcal{K} : \Gamma \Uparrow N}{\vdash \mathcal{K} : \Gamma \Downarrow N}\ R \Downarrow
\qquad
\frac{\vdash \mathcal{K} : \Gamma, S \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, S}\ R \Uparrow$$

Fig. 2: Focused linear logic system with subexponentials (assuming that all atoms a are negative and $a^\perp$ are positive; L is a list of formulas; Γ is a multi-set of positive formulas and literals; S is a positive formula or a literal; P is a non-negative literal and N is a negative formula).

$$(\mathcal{K}_1 \otimes \mathcal{K}_2)[i] = \begin{cases} \mathcal{K}_1[i] \cup \mathcal{K}_2[i] & \text{if } i \notin \mathcal{U} \\ \mathcal{K}_1[i] & \text{if } i \in \mathcal{U} \end{cases}
\qquad
\mathcal{K}[\mathcal{S}] = \bigcup \{\mathcal{K}[i] \mid i \in \mathcal{S}\}$$

$$(\mathcal{K} +_l A)[i] = \begin{cases} \mathcal{K}[i] \cup \{A\} & \text{if } i = l \\ \mathcal{K}[i] & \text{otherwise} \end{cases}
\qquad
\mathcal{K} \leq_i [l] = \begin{cases} \mathcal{K}[l] & \text{if } i \preceq l \\ \emptyset & \text{if } i \not\preceq l \end{cases}$$

$(\mathcal{K}_1 = \mathcal{K}_2)\,|_{\mathcal{S}}$ is true if and only if $(\mathcal{K}_1[j] = \mathcal{K}_2[j])$ for $j \in \mathcal{S}$

Fig. 3: Specification of operations on contexts ($i \in I$, $\mathcal{S} \subseteq I$).

2.3 Encoding Proof Systems by Example

This section revisits the necessary technical machinery for this paper illustrating by example how proof systems can be encoded in SELLF. A more detailed description can be found in [23,14].

We use two meta-level predicates $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ to encode object logic formulas, $\lfloor F \rfloor$, $\lceil F \rceil$, the former denoting an object logic formula appearing on the left-hand-side of a sequent and the latter denoting a formula on the right hand side. Thus a SELLF sequent $\vdash \lfloor F_1 \rfloor, \ldots, \lfloor F_n \rfloor, \lceil G_1 \rceil, \ldots, \lceil G_m \rceil$ specifies the object logic sequent $F_1, \ldots, F_n \vdash G_1, \ldots, G_m$. We write $\lfloor \Gamma \rfloor$ for the collection of atomic formulas $\{\lfloor F \rfloor \mid F \in \Gamma\}$, similarly for $\lceil \Delta \rceil$.

SELLF also provides mechanisms to organize such a collection of formulas by using subexponentials. For example, in LJ, we could use the signature $\Sigma = \langle \{l, r, \infty\}, \preceq, \{l\} \rangle$ with three subexponentials where $l \preceq \infty$ and $r \preceq \infty$. The LJ sequent $\Gamma \longrightarrow F$ is encoded as the SELLF sequent:

$$\vdash \mathcal{L} \mathrel{\dot{\otimes}} \lfloor \Gamma \rfloor \mathrel{\dot{\imath}} \lceil F \rceil \mathrel{\dot{\imath}} \cdot \Uparrow \cdot$$

where $\mathcal{L}$ is the theory specifying LJ inference rules. Although this might look redundant, each notation has its purpose. The predicates map object logic formulas to linear logic (atomic) formulas, and the left-right distinction is needed because left and right introduction rules for each connective are different. As we will see, applying a rule to a formula on the object logic amounts to "rewriting" the atomic formula in LL. The left and right subexponentials, on the other hand, are used when object level rules have conditions on the context. Take $\neg_l$ on an LJ system with negation, for example. It can only be applied if the right side of the sequent is empty. This is implicit on the way the rule is written in sequent calculus, and it will be captured by subexponentials in LL which will only allow the proof to continue if the subexponentials r is empty.

Object level introduction rules are defined using *bipoles*:

Definition 1 A *monopole* formula is a SELLF formula that is built up from atoms and occurrences of the negative connectives $(\&, \parr, \forall)$, with the restriction that, for all subexponentials t, $?^t$ has atomic scope and that all atomic formulas, A, are necessarily under the scope of a subexponential question-mark, $?^s A$, for some s. A *bipole* is a formula built from monopoles and negated atoms using only positive connectives $(\otimes, \oplus, \exists)$, with the additional restriction that $!^s$, $s \in I$, can only be applied to a monopole. We shall also insist that a bipole is either a negated atom or has a top-level positive connective.

The last restriction on bipoles forces them to be different from monopoles: bipoles are always positive formulas. Using the linear logic distributive properties, monopoles are equivalent to formulas of the form

$$\forall x_1 \ldots \forall x_p [\&_{i=1,\ldots,n} \parr_{j=1,\ldots,m_i} ?^{t_{i,j}} A_{i,j}],$$

where $A_{i,j}$ is an atomic formula and $t_{i,j} \in I$. Similarly, bipoles can be rewritten as formulas of the form

$$\exists x_1 \ldots \exists x_p [\oplus_{i=1,\ldots,n} \otimes_{j=1,\ldots,m_i} C_{i,j}],$$

where $C_{i,j}$ are either negated atoms, monopole formulas, or the result of applying $!^s$ to a monopole formula for some $s \in I$.

It turns out that one can match exactly the shape of a derivation with the shape of the inference rule the bipole encodes. Consider, for example, the following bipole $F = \exists A \exists B.[\lfloor A \supset B \rfloor^{\perp} \otimes (!^l \lceil A \rceil \otimes \lfloor B \rfloor)]$ encoding the $\supset_l$ rule for intuitionistic logic. Assuming the signature $\langle \{l, r, \infty\}, \{l \prec \infty, r \prec \infty\}, \{l, \infty\} \rangle$, the only way to introduce F in SELLF is by using a derivation of the following form, where $F \in \mathcal{L}$:

$$\dfrac{\dfrac{\vdash \mathcal{L} \mathrel{\dot{\otimes}} \lfloor \Gamma \rfloor, \lfloor A \supset B \rfloor \mathrel{\dot{\imath}} \lceil A \rceil \mathrel{\dot{\imath}} \cdot \Uparrow \cdot \quad \vdash \mathcal{L} \mathrel{\dot{\otimes}} \lfloor \Gamma \rfloor, \lfloor A \supset B \rfloor, \lfloor B \rfloor \mathrel{\dot{\imath}} \lceil G \rceil \mathrel{\dot{\imath}} \cdot \Uparrow \cdot}{\vdash \mathcal{L} \mathrel{\dot{\otimes}} \lfloor \Gamma \rfloor, \lfloor A \supset B \rfloor \mathrel{\dot{\imath}} \lceil G \rceil \mathrel{\dot{\imath}} \cdot \Downarrow F}}{\vdash \mathcal{L} \mathrel{\dot{\otimes}} \lfloor \Gamma \rfloor, \lfloor A \supset B \rfloor \mathrel{\dot{\imath}} \lceil G \rceil \mathrel{\dot{\imath}} \cdot \Uparrow \cdot}$$

The derivation above corresponds exactly to the left implication introduction rule for intuitionistic logic with premises $\Gamma, A \supset B \longrightarrow A$ and $\Gamma, A \supset B, B \longrightarrow G$, and conclusion $\Gamma, A \supset B \longrightarrow G$. This adequacy is classified as *on the level of derivations* [21]. Notice the role of $!^l$ on the atom $\lceil A \rceil$. In order to introduce it, it must be the case that the context of subexponential r is empty. That is, the

formula $\lceil G \rceil$ is necessarily moved to the right branch. Our previous work [23] shows how to encode a number of proof systems with this level of adequacy, such as LJ, LK, a multi-conclusion system for intuitionistic logic, and several modal logics.

Definition 2 Specification of object logic rules of inference:

i. In its most general form, the clause specifying the *cut rule* has the form to the left, while the clause specifying the *initial rule* has the form to the right:

$$Cut = \exists A.!^a?^b\lfloor A \rfloor \otimes !^c?^d\lceil A \rceil \qquad \text{and} \qquad Init = \exists A.\lfloor A \rfloor^\perp \otimes \lceil A \rceil^\perp$$

where a, c are subexponentials that may or may not appear, depending on the structural restrictions imposed by the proof system.

ii. The *structural rules* are specified by clauses of the form below, where $i, j \in I$:

$$\exists A.[\lfloor A \rfloor^\perp \otimes (?^i\lfloor A \rfloor \parr \cdots \parr ?^i\lfloor A \rfloor)] \qquad \text{or} \qquad \exists A.[\lceil A \rceil^\perp \otimes (?^j\lceil A \rceil \parr \cdots \parr ?^j\lceil A \rceil)].$$

iii. Finally, an *introduction clause* is a closed bipole formula of the form

$$\exists x_1 \ldots \exists x_n[(q(\diamond(x_1, \ldots, x_n)))^\perp \otimes B]$$

where $\diamond$ is an object-level connective of arity n ($n \geq 0$) and $q \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil\}$. Furthermore, B does not contain negated atoms and an atom occurring in B is either of the form $q(x_i)$ or $q(x_i(y))$ where $1 \leq i \leq n$. In the former, x_i has type obj while in the latter x_i has type $d \to obj$ and y is a variable (of type d) quantified (universally or existentially) in B (in particular, $y \notin \{x_1, \ldots, x_n\}$).

Remark 1 We use SELL as the specification language of proof systems because, up to the best of our knowledge, SELL is currently the state-of-the-art of logical frameworks, particularly for the specification of proof systems. It allows to capture declaratively, *i.e.*, using logical connectives, complicated structural restrictions of (object) proof systems. Due to the declarative nature of SELL specifications, the construction of Answer Set Programs is reduced to specifying the logical meaning of connectives.

3 Building derivations

Given a SELLF specification of a proof system, we show how to extract logic programs whose answer sets specify all possible results of applying an object logic rule to a (partially determined) sequent. We first review the basics of answer set programming and then show the set of constraints that they use for our task.

3.1 Answer set programming

Answer set programming (ASP) [7] is a logic programming paradigm tailored to handle computationally difficult problems (typically NP). Given a logic program (in a Prolog-like language), the solver finds the truth assignments (answer-sets) which renders the program valid.

Syntax Let $\mathcal{K}$ be a set of propositional variables. A *default literal* is an atomic formula preceded by *not*. A propositional variable and a default literal are both literals. A rule r is an ordered pair $Head(r) \leftarrow Body(r)$, $Head(r)$ is a literal or the constant $\bot$ (for falsity) and $Body(r)$ is a finite set of literals. A rule with $Head = L$ and $Body(r) = \{L_1, \ldots, L_n\}$ is written $L \leftarrow L_1, \ldots, L_n$.

Semantics An *interpretation* M of $\mathcal{K}$ is a subset of $\mathcal{K}$. An atomic formula, A, is true in M, written $M \vDash A$, if $A \in M$, otherwise false. A literal *not* A is true in M, written $M \vDash not\ A$, if $A \notin M$, otherwise false. An *answer set program* (which we call LP) is a set of rules. An interpretation M is an *answer set* of an LP P if $M' = M \cup \{not\_A \mid A \notin M\}$ and $M' = least(P \cup \{not\_A \mid A \notin M\})$, where *least* is the smallest model of the *definite logic program* obtained from the argument program by replacing all occurrences of *not* A by a new atomic formula not_A. In the remainder of this paper, we will not explicitly write the set $\mathcal{K}$, but assume that it consists exactly of the symbols appearing explicitly in the programs. Moreover, as usual, we consider variables appearing in programs as a shorthand for the set of all its possible ground instantiations.

The interpretation of the default negation *not* assumes a *closed-world* assumption of programs. That is, we assume to be true only the facts that are explicitly supported by a rule. For example, the following program with three rules has two answer-sets $\{a, c\}$ and $\{b\}$:

$$a \leftarrow not\ b \qquad b \leftarrow not\ a \qquad c \leftarrow a$$

Finally, one can also specify a constraint in ASP by using a rule whose head is the falsity, denoted by the symbol $\bot$. For example, the rule specifying that b cannot be true is:

$$\bot \leftarrow b$$

Thus, the program resulting from adding this rule to the program above has a single answer-set, namely $\{a, c\}$.

3.2 Deriving one bipole

As shown previously, the application of an inference rule in the object logic is equivalent to the derivation of its corresponding formula (a bipole) in focused linear logic with subexponentials. The various ways in which this bipole can be derived, starting with a specific sequent, corresponds to the possible ways the object logic rule can be applied to the corresponding sequent. Since our aim is to determine the permutability of two inference rules, we need to use sequents as generic as possible. When doing these proofs by hand, we use context variables, usually denoted by Γ and Δ that represent an arbitrary (multi-)set of formulas. In order to automate this procedure, we implemented what we call *schema-sequents*.

Definition 3 *Schema-sequents* are partially determined sequents containing formulas and context variables. The context variables are parametrized by indices.

Given a linear logic schema-sequent, we derive a bipole and generate constraints at the same time. The constraints are added to an LP and the models of this LP will

Predicate name	Description
$in(F, \Gamma, n)$	Formula F occurs in Γ n times.
$union(\Gamma^1, \Gamma^2, \Gamma)$	Γ is the result of the multi-set union of Γ^1 and Γ^2.
$minus(\Gamma^1, F, \Gamma)$	Γ is the result of removing one occurrence of F from Γ^1.
$empty(\Gamma)$	The context Γ is empty.

Table 1: Predicates used for describing constraints on contexts

represent the possible derivations (i.e., the possible results of the rule application on the level of the object-logic).

The constraints (Table 1) are multi-set properties of the (linear logic) contexts, and they are generated as the inference rules of SELLF are applied. By design, the properties of a context in a sequent S are described using solely the contexts that occur in the premises of the rule whose conclusion is S. The same way that the main formula of an inference rule in (cut-free) sequent calculus is determined by a combination of the auxiliary formulas in the premises, the context of the conclusion will be determined only by the contexts of the premises.

This is achieved by what we call *context indexing*. Since the derivation construction is done bottom-up, each time an inference rule is applied, new indices may be created for the contexts at the premises, and a constraint is generated defining the conclusion context in terms of these new ones. Although we use SELLF, let's look at this with a simple example in multiplicative intuitionistic logic. Assume the following inference rules for conjunction and implication on the right:

$$\frac{\Gamma \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A \wedge B}\ \wedge_r \qquad \frac{\Gamma, A \vdash}{\Gamma \vdash \neg A}\ \neg_r$$

Now suppose we have the following schema-sequent $\Gamma^0 \vdash p \wedge \neg q$. We will first apply $\wedge_r$, which splits the conclusion context Γ^0. Therefore, we create two new indices for the contexts of the premises, say, Γ^1 and Γ^2 and the constraint $union(\Gamma^1, \Gamma^2, \Gamma^0)$. Now the sequent on the right premise is $\Gamma^2 \vdash \neg q$, to which we apply $\neg_r$. This rule will add q to the left context, therefore we create a new context variable Γ^3 and define Γ^2 by the constraint $minus(\Gamma^3, q, \Gamma^2)$. At the end, we have the following schema-derivation with the set of constraints on the right:

$$\dfrac{\Gamma^1 \vdash p \quad \dfrac{\Gamma^3 \vdash}{\Gamma^2 \vdash \neg q}\ \neg_r}{\Gamma^0 \vdash p \wedge \neg q}\ \wedge_r \qquad \begin{array}{l} minus(\Gamma^3, q, \Gamma^2) \\ union(\Gamma^1, \Gamma^2, \Gamma^0) \end{array}$$

Our SELLF sequent is one-sided and determined by the subexponential signature. In this case, we will have a schema-sequent with one indexed context variable for each subexponential (corresponding to $\mathcal{K}$) and another one for Γ. The right side of $\Uparrow$ and $\Downarrow$ will be represented simply by a list of formulas, since the is the *working zone* in a derivation. Indeed, notice that this list is empty at the beginning and at the end of a bipole. We now list SELLF rules that generate constraints and update the indices of the contexts. All other rules keep the context *as is*.

$R\Uparrow$ - *release*

$$\frac{\vdash \mathcal{K} : \Gamma, S \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, S}\ R\Uparrow$$

The release rule stores a literal or positive formula for later processing when the proof is in a negative phase. This formula is stored in the bounded context Γ, containing the formulas not marked with any subexponential. Since this context is modified, its index is updated. Let Γ^i represent this context in the conclusion sequent and Γ^j the new context in the premise, then the release up rule generates the following constraint: $minus(\Gamma^j, S, \Gamma^i)$.

$?^l$ - *question mark*

$$\frac{\vdash \mathcal{K} +_l A : \Gamma \Uparrow L}{\vdash \mathcal{K} : \Gamma \Uparrow L, ?^l A} \; ?^l$$

Analogous to the release up rule, the question mark rule will also store a formula, but this time to a subexponential context. If Θ^i and Θ^j are the corresponding contexts in the conclusion and premise, respectively, the application of this rule generates the constraint: $minus(\Theta^j, A, \Theta^i)$.

1 - one

$$\frac{}{\vdash \mathcal{K} : \cdot \Downarrow 1} \; 1, \text{ given } \mathcal{K}[\mathcal{I} \setminus \mathcal{U}] = \emptyset$$

In order for the one rule to succeed, all bounded contexts must be empty. Therefore, for every Θ^i_s in the conclusion such that s is a bounded subexponential, this rule generates the constraint $empty(\Theta^i_s)$. It also generates $empty(\Gamma^i)$ for the context Γ.

$\otimes$ - *tensor*

$$\frac{\vdash \mathcal{K}_1 : \Gamma \Downarrow A \quad \vdash \mathcal{K}_2 : \Delta \Downarrow B}{\vdash \mathcal{K}_1 \otimes \mathcal{K}_2 : \Gamma, \Delta \Downarrow A \otimes B} \; \otimes, \text{ given } (\mathcal{K}_1 = \mathcal{K}_2)|_{\mathcal{U}}$$

The tensor rule splits the bounded context between the premises, while copying the unbounded contexts to both premises. Since the unbounded contexts will be the same, their indices are not updated, but the indices of the bounded context need to be updated. Let s be a bounded subexponential and Θ^i_s be its context in the conclusion. New contexts Θ^j_s and Θ^k_s are created for each premise and the constraint $union(\Theta^j_s, \Theta^k_s, \Theta^i_s)$ is generated. The same is done for the Γ context.

$!^l$ - *bang*

$$\frac{\vdash \mathcal{K} \leq_l : \cdot \Uparrow A}{\vdash \mathcal{K} : \cdot \Downarrow !^l A} \; !^l, \text{ given } \mathcal{K}[\{x \mid l \not\geq x \wedge x \notin \mathcal{U}\}] = \emptyset$$

The bang rule is more involved. Its side condition states that every *bounded* subexponential context that is smaller than or not related to l must be empty, as well as Γ. Additionally, the operation $\mathcal{K} \leq_l$ will erase the formulas from the *unbounded* subexponential contexts that are smaller than or not related to l. In other words, the application of the bang rule will require the emptiness of every context that is smaller than or not related to l. If it is a bounded context, it must already be empty in the conclusion of the rule and if it is an unbounded context, it will have its formulas erased. Therefore, for every Θ^i_s in the conclusion such that $l \not\geq s$, we either assert $empty(\Theta^i_s)$ if s is bounded or $empty(\Theta^k_s)$ for a new index k if s is unbounded. As usual, Γ is treated as in the bounded case.

I - initial

$$\frac{}{\vdash \mathcal{K} : \Gamma \Downarrow A_t^{\perp}} \text{ I, given } A_t \in (\Gamma \cup \mathcal{K}[\mathcal{I}]) \text{ and } (\Gamma \cup \mathcal{K}[\mathcal{I} \setminus \mathcal{U}]) \subseteq \{A_t\}$$

For the initial rule, we need that the dual of the atom is in the context and, additionally, that there are no other bounded formulas present. The location of this dual formula can be anywhere, therefore the application of this rule will generate different constraint sets, one for each possibility. Let's assume the dual formula is in some subexponential s (bounded or unbounded). Then, every bounded subexponential $l \neq s$ (and Γ) should be empty, therefore, the constraints $empty(\Theta_l^i)$ are generated for Θ_l^i in the conclusion. Additionally, it must be the case that A_t is actually in s, but we do not want to just put it there. It must *be* there as a consequence of the operations performed in the proof given the initial sequent. This requirement is represented in the LP program as rule with head $\perp$.

$$\perp \leftarrow not\ in(A_t, \Theta_s^i, N), N > 0.$$

It means that, if we cannot derive that F occurs Θ_s^i, then fail. This restriction alone is enough if s is unbounded, but if it is bounded, we need to also guarantee that A_t is the *only* formula there. This is again achieved by rule with head $\perp$:

$$\perp \leftarrow in(A_t, \Theta_s^i, N), N > 1.$$
$$\perp \leftarrow in(A_t, \Theta_s^i, N), N > 0, in(F, \Theta_s^i, M), M > 0, F \neq A_t.$$

Meaning that if there are other copies of A_t or another formula $F \neq A_t$ in the context, it fails.

Let $\mathcal{C}$ denote the set of constraint-sets generated so far and $\mathcal{C}_I$ the set of all constraint-sets for each subexponential. Then the new constraint set is the cartesian product $\mathcal{C} \times \mathcal{C}_I$. In practice, there are not many subexponentials, so the number of sets in $\mathcal{C}_I$ is small and, moreover, many combinations will turn out to have no models.

$\oplus$ - *plus*

$$\frac{\vdash \mathcal{K} : \Gamma \Downarrow A_i}{\vdash \mathcal{K} : \Gamma \Downarrow A_1 \oplus A_2} \oplus_i$$

The $\oplus$ rule does not do anything with the context, and therefore it does not generate new constraints. We list it here because it is responsible for generating alternative derivations. When applying it, we must choose to continue with either A or B. Since we want *all* possible derivation, we need to consider both choices. Uppon applying $\oplus$, we duplicate the derivation and constraint set. As a result, at the end of the bipole, we have potentially a list of pairs $\langle$proof tree, constraint set$\rangle$, where the valid proof trees will be those whose constraint-set has a model.

3.3 Building a bipole

Using the constraints generated by each rule application we can build a schematic bipole accompanied by a constraint set. In fact, there might be many possible schematic bipoles (due to the application of $\oplus$), each with many possible constraint

sets (due to the application of *init*). Each constraint set is appended to the logic program in Figure 4, and the answer sets for this program correspond to the valid bipoles. It is possible that no or multiple models are generated for a constraint set corresponding to a particular schematic bipole.

Let's look at the bipole $\lfloor A \supset B \rfloor^{\perp} \otimes !^l ?^r \lceil A \rceil \otimes ?^l \lfloor B \rfloor$ for LJ's $\supset_l$ as an example. We will derive it starting with a schema sequent representing an object logic sequent $\Gamma, A \supset B, C \wedge D \vdash \Delta$, where Δ has at most one formula. This schema sequent consists of the following context variables and constraint set:

$$\vdash \Gamma_\infty; \Gamma_l^0; \Gamma_r^0; \Gamma^0 \Uparrow \qquad in(\lfloor A \supset B \rfloor, \Gamma_l^0, 1), in(\lfloor C \wedge D \rfloor, \Gamma_l^0, 1).$$

Since there are no applications of $\oplus$, only one schematic bipole is generated.

$$\dfrac{\dfrac{}{\vdash \Gamma_\infty^0; \Gamma_l^1; \Gamma_r^1; \Gamma^1 \Downarrow \lfloor A \supset B \rfloor^{\perp}}\, I \qquad \dfrac{\dfrac{\dfrac{\vdash \Gamma_\infty^0; \Gamma_l^3; \Gamma_r^5; \Gamma^3 \Uparrow}{\vdash \Gamma_\infty^0; \Gamma_l^3; \Gamma_r^3; \Gamma^3 \Uparrow ?^r \lceil A \rceil}\, ?^r}{\vdash \Gamma_\infty^0; \Gamma_l^3; \Gamma_r^3; \Gamma^3 \Downarrow !^l ?^r \lceil A \rceil}\, !^l \qquad \dfrac{\dfrac{\vdash \Gamma_\infty^0; \Gamma_l^5; \Gamma_r^4; \Gamma^4 \Uparrow}{\vdash \Gamma_\infty^0; \Gamma_l^4; \Gamma_r^4; \Gamma^4 \Uparrow ?^l \lfloor B \rfloor}\, ?^l}{\vdash \Gamma_\infty^0; \Gamma_l^4; \Gamma_r^4; \Gamma^4 \Downarrow ?^l \lfloor B \rfloor}\, R\Downarrow}{\vdash \Gamma_\infty^0; \Gamma_l^2; \Gamma_r^2; \Gamma^2 \Downarrow !^l ?^r \lceil A \rceil \otimes ?^l \lfloor B \rfloor}\, \otimes}{\vdash \Gamma_\infty^0; \Gamma_l^0; \Gamma_r^0; \Gamma^0 \Downarrow \lfloor A \supset B \rfloor^{\perp} \otimes !^l ?^r \lceil A \rceil \otimes ?^l \lfloor B \rfloor}\, \otimes}{\vdash \Gamma_\infty^0; \Gamma_l^0; \Gamma_r^0; \Gamma^0 \Uparrow}\, D_\infty$$

Of all constraint sets generated, only the one below has models. Each constraint has a color that corresponds to the rule in the bipole responsible for generating it. The first constraints have no color because they are the initial ones, defining the schema sequent to which the rule will be applied. The two models generated for this constrain set (with the program in Figure 4) contain either $in(\lfloor C \wedge D \rfloor, \Gamma_l^3, 1)$ or $in(\lfloor C \wedge D \rfloor, \Gamma_l^5, 1)$. They correspond exactly to the two object logic derivations where the $\wedge$ formula either goes to the left or right premise of $\supset_l$.

$in(\lfloor A \supset B \rfloor, \Gamma_l^0, 1).$
$in(\lfloor C \wedge D \rfloor, \Gamma_l^0, 1).$
$union(\Gamma_l^1, \Gamma_l^2, \Gamma_l^0).$
$union(\Gamma_r^1, \Gamma_r^2, \Gamma_r^0).$
$union(\Gamma^1, \Gamma^2, \Gamma^0).$
$:- in(\lfloor A \supset B \rfloor, I, \Gamma_l^1), I = 0.$
$:- in(\lfloor A \supset B \rfloor, I, \Gamma_l^1), I > 1.$
$:- in(F, \_, \Gamma_l^1), F! = \lfloor A \supset B \rfloor.$
$emp(\Gamma_r^1).$

$emp(\Gamma^1).$
$union(\Gamma_l^3, \Gamma_l^4, \Gamma_l^2).$
$union(\Gamma_r^3, \Gamma_r^4, \Gamma_r^2).$
$union(\Gamma^3, \Gamma^4, \Gamma^2).$
$emp(\Gamma_r^3).$
$emp(\Gamma^3).$
$minus(\Gamma_r^5, \lceil A \rceil, \Gamma_r^3).$
$minus(\Gamma_l^5, \lfloor B \rfloor, \Gamma_l^4).$

Informally, the procedure first computes a set of schematic bipoles with constraint sets: $\{\langle \mathcal{B}_1, C_1 \rangle, ..., \langle \mathcal{B}_n, C_n \rangle\}$. Then it uses the logic program in Figure 4 to compute the models of each C_i. For each model M found, it adds a pair $\langle \mathcal{B}_i, M \rangle$ to the returned set. This means that each schematic bipole might occur more than once

```
% in(F, Ctx, N) -> Formula F occurs in Ctx N times
% in_unique(F, I, C) -> Represents one occurrence of F in C
% union(C1, C2, C) -> C = C1 U C2
% minus(C1, F, C0) -> C0 = C1 - F
% emp(C) -> C is the empty set

%%%%%%%%%%%%%%%% Clauses for multi-set operations in contexts %%%%%%%%%%%%%%%%
#maxint=5.

% Distinguishes formula occurrences
% in(F, C, 3) becomes { in_unique(F, 1, C), in_unique(F, 2, C), in_unique(F, 3, C) }
in_unique(F, I, C) :- in(F, C, N), #int(I), 1 <= I, I <= N.
% Distributes each copy individually
in_unique(F, I, C1) v in_unique(F, I, C2) :- in_unique(F, I, C), union(C1, C2, C).
% Avoids duplicated results
:- in_unique(F, I, C), in_unique(F, I1, C1), in_unique(F, I2, C2), I1 > I2, union(C1, C2, C).

% C0 = C1 - {F}
% Every formula occurring in C0 is also in C1
in_unique(F, I, C1) :- minus(C1, _, C0), in_unique(F, I, C0).
% C1 has one extra occurrence of F
contained(F, C) :- max_index(F, _, C).
% Index 1 if it's the first occurrence
in_unique(F, 1, C1) :- minus(C1, F, C0), not contained(F, C0).
% Otherwise index max_index + 1
not_max_index(F, I, C) :- in_unique(F, I, C), in_unique(F, J, C), J > I.
max_index(F, I, C) :- in_unique(F, I, C), not not_max_index(F, I, C).
in_unique(F, I, C1) :- minus(C1, F, C0), contained(F, C0), max_index(F, J, C0), I = J+1.

:- in_unique(F, I, C), I > 0, emp(C).
emp(C1) :- emp(C), union(C1, C2, C).
emp(C2) :- emp(C), union(C1, C2, C).
emp(C) :- emp(C1), emp(C2), union(C1, C2, C).
```

Fig. 4: Answer set program for finding models of valid derivations (the `in_unique` clauses are needed to deal with structural rules).

in the returned set, in case its constraints had more than one model. Remember that a model is simply a set of constraints that are true.

Definition 4 Let $\mathcal{S}$ be a schema-sequent, C an initial constraint set and B a bipole. We define the function $deriveBipole(\mathcal{S}, C, B)$ as the process of computing the set of pairs $\{\langle \mathcal{B}_1, M_1\rangle, ..., \langle \mathcal{B}_n, M_n\rangle\}$ where each $\mathcal{B}_i$ is a schematic derivation of B described by a valid model M_i.

Theorem 2 *Let $\mathcal{S}$ be a SELLF schema-sequent, B a bipole and C a set of in constraints. Then $deriveBipole(\mathcal{S}, C, B)$ computes all possible ways of deriving B on $\mathcal{S}$, given the information in C.*

Proof Given the structure of a bipole, the choice points during its derivation are at applications of $\oplus$ and $\otimes$. The *deriveBipole* procedure considers both choices of each $\oplus$ by duplicating the derivation, and all possible options of splitting a context on $\otimes$ is the result of the logic program generated.

Handling structural rules When constructing bipoles for structural rules (mostly contraction), we need that the logic program distinguish between different occurrences of the same formula in a context. This is achieved by the implementation of a `in_unique(F, I, C)` clause, which is equivalent to adding a unique index `I` to

each occurrence of `F` in `C`. These are generated from the fact that a formula occurs `N` times in a context, as mentioned in the comment of the code in Figure 4.

One might think that a direct numeric implementation would be easier. Indeed we have tried that with the following code:

```
in(F, C1, N) :- union(C1, C2, C0), in(F, C0, M), greater_than_zero(N), N <= M.
in(F, C2, N) :- union(C1, C2, C0), in(F, C0, M), greater_than_zero(N), N <= M.
:- union(C1, C2, C0), in(F, C1, I), in(F, C2, J), in(F, C0, N), M = I+J, N != M.
```

The reason why this fails is because, given the facts `in("a", s, 3), union(s1, s2, s)`, the two first clauses generate `in("a", s1, N), in("a", s2, N)` for every possible `N`, i.e, $1 \leq N \leq 3$. These are not "filtered" by the third clause, as one could imagine, which simply fails and returns no models.

3.4 Deriving multiple bipoles

We show now how to find derivations of two bipoles, one on top of the other. It is a simple matter of iterating the procedure defined on the previous section.

Suppose we want to find the valid derivations where a rule r_1 is applied below a rule r_2. Let B_1 and B_2 be the two linear logic bipoles encoding those rules, respectively. The first thing to do is to create the initial schema-sequent $\mathcal{S}$, containing context variables for each subexponential in the encoding of the object logic. Associated to this initial sequent we have the initial set of constraints C_0. This is obtained simply by asserting that generic formulas containing the main connective for r_1 and r_2 are in one of the context variables[3].

We can thus compute the first set of bipoles:

$$deriveBipoles(\mathcal{S}, C_0, B_1) = \{\langle \mathcal{B}_1, M_1 \rangle, ..., \langle \mathcal{B}_n, M_n \rangle\}$$

For each pair $\langle \mathcal{B}_i, M_i \rangle$, we take the open leaves $\mathcal{O}_1, ..., \mathcal{O}_{k_i}$ and derive the second bipole on top of them. So for each open leaf, we may have many possibilities for continuing it:

$$\begin{gathered} deriveBipoles(\mathcal{O}_1, M_i, B_2) = \{\langle \mathcal{B'}_1, M'_1 \rangle, ..., \langle \mathcal{B'}_{n_1}, M'_{n_1} \rangle\} \\ \vdots \\ deriveBipoles(\mathcal{O}_{k_i}, M_i, B_2) = \{\langle \mathcal{B'}_1, M'_1 \rangle, ..., \langle \mathcal{B'}_{n_{k_i}}, M'_{n_{k_i}} \rangle\} \end{gathered}$$

To get all possible derivations for each $\mathcal{B}_i$, we need to combine all possible ways of continuing an open leaf of $\mathcal{B}_i$, including not deriving anything at all on it. All those combinations can be obtained by adding one dummy element to each set above, representing the empty derivation, and taking the cartesian product of all sets. Given a choice for the open leaves, we check if the union of M_i with the M_{n_j}'s has one or more models. If so, the bipoles are plugged together and returned with the corresponding model(s).

[3] In principle, there might be several initial constraint sets if each formula can be placed in different contexts. In Quati we use annotations to identify the context side (on the object logic sequent) of each subexponential and thus reduce the number of options. For simplicity, we will consider only one initial constraint set, but the generalization is straighforward.

Let's see how this works for the example above. The first set of bipoles was already computed and we have two models for the same bipole. Each model places the $\wedge$ formula on one open leaf.

$$\{\langle \mathcal{B}, M_1\rangle, \langle \mathcal{B}, M_2\rangle\}$$

This schematic bipole $\mathcal{B}$ has two open leaves:

$$\mathcal{O}_1 : \vdash \Gamma^0_\infty; \Gamma^3_l; \Gamma^5_r; \Gamma^3 \Uparrow$$
$$\mathcal{O}_2 : \vdash \Gamma^0_\infty; \Gamma^5_l; \Gamma^4_r; \Gamma^4 \Uparrow$$

We try to derive the bipole B_2 for $\wedge_l$, i.e., $\lfloor C \wedge D\rfloor^{\perp} \otimes ?^l \lfloor C\rfloor \parr ?^l \lfloor D\rfloor$, on those leaves starting first with model M_1 and later with model M_2. Suppose M_1 is the model with $in(\lfloor C \wedge D\rfloor, \Gamma^3_l, 1)$ and M_2 the one containing $in(\lfloor C \wedge D\rfloor, \Gamma^5_l, 1)$. Then:

$$deriveBipoles(\mathcal{O}_1, M_1, B_2) = \{\langle \mathcal{B}'_1, M'_1\rangle\}$$
$$deriveBipoles(\mathcal{O}_2, M_1, B_2) = \{\}$$

$$deriveBipoles(\mathcal{O}_1, M_2, B_2) = \{\}$$
$$deriveBipoles(\mathcal{O}_2, M_2, B_2) = \{\langle \mathcal{B}'_2, M'_2\rangle\}$$

The open leaves that do not contain the main formula for B_2 will end up not generating any models. The combinations and final bipoles in this case are trivial.

This example serves to show that, albeit the combinatorial nature of the algorithm, the sizes of the sets make this feasible. This is in part because many invalid constraint sets are discarded along the way, which is only possible due to the power of ASP.

4 Inferring Provability

We now show how to automate also using answer-set programs a solution to the following problem which we call the *implication derivability problem*:

Given two derivations Ξ_1 and Ξ_2, if Ξ_1 can be completed to a proof, can Ξ_2 also be completed to a proof?

The solution of this problem can be used in general to determine whether two rules permute. Consider for instance, the permutation shown in the Introduction:

$$\dfrac{\dfrac{\varphi_1}{\Gamma \vdash A} \quad \dfrac{\varphi_2}{\Gamma', P, Q, B \vdash F}\,\rightarrow_l}{\Gamma, \Gamma', A \to B, P, Q \vdash F}\,\wedge_l \quad \leadsto \quad \dfrac{\dfrac{\varphi_1}{\Gamma \vdash A} \quad \dfrac{\varphi_2}{\Gamma', P, Q, B \vdash F}\,\wedge_l}{\Gamma, \Gamma', P \wedge Q, A \to B \vdash F}\,\rightarrow_l$$

If the derivation to the left can be completed to a proof, then the derivation to the right where the inference rules are permuted is also provable. However, in general, solving the implication derivability problem is undecidable.

Theorem 3 *For the proof system LK for classical logic, the implication derivability problem is undecidable.*

Proof We reduce the implication derivability problem to the problem of checking whether the sequent $\vdash G$ for a classical formula G has an LK proof which is a problem known to be undecidable.

We can check whether a classical logic formula G has an LK proof by instantiating the implication derivability problem as follows: Set Ξ_1 to be a derivation with a single open sequent $\vdash \top$, which is always provable, and set Ξ_2 to be $\vdash G$. This instance of the implication derivability problem has affirmative answer if and only if G has an LK proof. Since the latter problem is undecidable, the implication derivability problem is also undecidable.

Given that the implication derivability problem is undecidable in general, there is no hope to devise a sound and complete method for answering this problem.[4] We provide, therefore, a sound but incomplete method which using the machinery built in Section 3 to check when a rule permutes over another rule.

Our method emulates the reasoning used by proof theorists which normally consists in trying to match the open sequents of Ξ_1 with the open premises of Ξ_2:

1. Let $\mathcal{S}_1^1, \ldots, \mathcal{S}_1^n$ be the open sequents of Ξ_1 and let $\mathcal{S}_2^1, \ldots, \mathcal{S}_2^m$ be the open sequents of Ξ_2;
2. We check for each $1 \leq i \leq m$, whether there is $1 \leq j \leq n$ such that the sequent $\mathcal{S}_2^i$ can be *matched* by the sequent $\mathcal{S}_1^j$. Here we are using a loose notion of matching. We will formalize it later in this section;
3. If so, then the answer is affirmative for the implication derivability problem and negative otherwise.

For example, in the derivations above, every open sequents in the derivation to the right also appear in the derivation to the left.

Our goal here is more elaborate. We will depart from a SELL theory encoding the proof rules of a proof system as detailed in Section 2.3 and use the answer-set programs extracted from these theories as detailed in Section 3 to check which pairs rules of the encoded proof system permute. We follow the steps below:

1. Suppose we would like to check whether rule r_1 permutes over rule r_2.
2. Construct all derivation schemas with application of r_1 appearing below possibly multiple applications of r_2 as detailed in Section 3. Similarly, construct all derivation schemas with r_2 appearing below possibly multiple applications of r_1. We obtain collections of derivations and answer-sets for when r_1 appears below r_2, represented by $r1/r_2$, and for when r_2 appears below r_1, represented by r_2/r_1:

$$Der(r1/r2) = \langle \Xi_1^1, M_1^1 \rangle, \ldots, \langle \Xi_1^n, M_1^n \rangle$$
$$Der(r_2/r_1) = \langle \Xi_2^1, M_2^1 \rangle, \ldots, \langle \Xi_2^m, M_2^m \rangle$$

3. For each $\langle \Xi_2^j, M_2^j \rangle \in Der(r_2/r_1)$, we search a $\langle \Xi_1^i, M_1^i \rangle \in Der(r_1/r_2)$ which matches $\langle \Xi_2^j, M_2^j \rangle$ as follows:
   (a) For each open sequent $\mathcal{S}_2$ in $\langle \Xi_2^j, M_2^j \rangle$, we check whether there is a matching open sequent $\mathcal{S}_1$ in $\langle \Xi_1^i, M_1^i \rangle$ that matches $\mathcal{S}_2$.

[4] One may try to devise different more specific versions for the implication derivability problem, *e.g.*, assuming that the symbols in the derivations are the same. Such definitions tend, however, to be much more elaborate with many cases [13].

The interesting part is step 3 as the remaining steps of enumerating derivation schemas reduces to brute force using the machinery described in Section 3. For this step, we rely on the following lemmas:

Lemma 1 *The following inference rule is admissible in $SELL_\Sigma$ for a subexponential signature $\Sigma = \langle I, \preceq, U\rangle$:*

$$\frac{\vdash ?^{s_1} F, \Gamma}{\vdash ?^{s_2} F, \Gamma} \textit{ subred, provided } s_1 \preceq s_2$$

Proof The proof follows from SELL cut-elimination:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{F, F^\perp}\, I}{\vdash ?^{s_2} F, F^\perp}\, ?^{s_2}}{\vdash ?^{s_2} F, !^{s_1} F^\perp}\, !^{s_1} \qquad \vdash ?^{s_1} F, \Gamma}{\vdash ?^{s_2} F, \Gamma}\, Cut$$

Lemma 2 *Let $\mathcal{S}_1 = \quad \vdash \mathcal{K}_1 : \Gamma \Uparrow \cdot$ and $\mathcal{S}_2 = \quad \vdash \mathcal{K}_2 : \Gamma \Uparrow \cdot$ be two $SELLF_\Sigma$ sequents for a subexponential signature $\Sigma = \langle I, \prec, U\rangle$. The sequent $\mathcal{S}_2$ is provable provided the sequent $\mathcal{S}_1$ is provable, if for all subexponentials $s \in I$, at least one of the following conditions is satisfied:*

1. *(Same Formulas) $s \notin U$ and $\mathcal{K}_1[s] = \mathcal{K}_2[s]$;*
2. *(Subset of Unbounded Formulas) $s \in U$ and $\mathcal{K}_1[s] \subseteq \mathcal{K}_2[s]$;*
3. *(Upwardly closed) $\mathcal{K}_2[s] \subseteq \mathcal{K}_1[s]$ and for each formula in $F \in \mathcal{K}_1[s] \setminus \mathcal{K}_2[s]$, there is a subexponential $s' \in U$ such that $s \preceq s'$ and $F \in \mathcal{K}_2[s']$.*

Proof Assuming that $\mathcal{S}_1$ is provable, we show that $\mathcal{S}_2$ is also provable. We do so by constructing a derivation ending with $\mathcal{S}_2$ and with only premise $\mathcal{S}_1$ in SELL:

$$\frac{\mathcal{S}_1}{\mathcal{S}_2}$$

From the completeness Theorem 1, we get that $\mathcal{S}_2$ is provable in SELLF.

For the contexts $\mathcal{K}_2[s]$ that satisfy the first condition (Same Formulas) nothing is to be done as the contexts match. For the contexts $\mathcal{K}_2[s]$ that satisfy the second condition (Subset of Unbounded Formulas), since the $s \in U$, we simply weaken the formulas in $\mathcal{K}_1$, obtaining the derivation:

$$\frac{\mathcal{S}_1}{\mathcal{S}_2'}\, n \times W$$

Now we handle the contexts that satisfy condition three (Upwardly closed), we simply apply Lemma 1 obtaining a derivation:

$$\begin{array}{l} \dfrac{\mathcal{S}_1}{\mathcal{S}_2'}\, n \times W \\ \dfrac{}{\mathcal{S}_2}\, m \times \text{ subred} \end{array}$$

This completes the proof.

We are going to use the conditions in Lemma 2 to solve the step 3. In particular, given two derivation schemas $\langle \Xi_1, M_1\rangle$ and $\langle \Xi_2, M_2\rangle$, we check whether the

Predicate name	Description
`proveIf(S2,S1)`	The sequent `S2` is provable if the sequent `S1` is provable.
`ctx(C,SE,TP,S,tree)`	`C` is the context's name, `SE` is the subexponential label, `TP` is the type of subexponential, `S` is the sequent index, `tree` is the corresponding tree.
`not_proveIf(S2,S1)`	The sequent `S2` is not necessarily provable if the sequent `S1` is provable.
`condition{1,2,3}(S2,S1,SE)`	The context `SE` of sequent `S2` satisfies the corresponding condition, 1,2,3, with respect to the sequent `S1`.
`in_sequent_tree{1,2}(F,SE,TP,M,S)`	The formula `F` is in the subexponential context of type `TP` (linear or unbounded) with multiplicity `M` in the sequent `S` of the derivation `tree1` or `tree2`.
`in_context_tree{1,2}(F,SE,TP,S)`	Similar as in `in_sequent_tree`, but without taking the formula's multiplicity into account.
`in_unbctx_geq(F,SE,S)`	Formula `F` appears in `tree2` in some unbounded context with subexponential greater than `SE`.
`geq(SE1,SE2)`	The subexponential `SE1` is greater or equal than the subexponential `SE2`.

Table 2: Predicates used for checking the implication derivability problem where `tree1` corresponds to Ξ_1 and `tree2` to Ξ_2.

conditions in Lemma 2 are satisfied by constructing an answer-set program, called `proveIf`, from the two derivation schemas.

The clauses of `proveIf` are depicted in Figure 5. It uses the predicates described in Table 2. We are interested in proving the predicate `proveIf(S2,S1)` for all sequents `S2` of $\langle \Xi_2, M_2 \rangle$. From the derivation schema $\langle \Xi_1, M_1 \rangle$, called `tree1`, and $\langle \Xi_2, M_2 \rangle$, called `tree2`, we extract the following information:

- The `in` facts from M_1;
- Label the open sequents of Ξ with fresh identifiers, `S1`, `S2`, ...;
- Construct the `ctx` facts specifying the sequents in `tree1` and `tree2`;
- Assume that `S1`, ..., `Sn` are the labels of the open sequents of Ξ_2. We construct the clause:

```
ok :- proveIf(S1,_),...,proveIf(Sn,_)
```

  I.e., it is possible to infer `ok` if it is possible to find a corresponding sequent in `tree1` for each open sequent in `tree2`.

We also need the subexponential relation of the system specification. The relation is specified in the program by clauses of the form `geq(SE1,SE2)` specifying that the subexponential `SE1` is greater than the subexponential `SE2`.

Given these facts, we use the clauses in Figure 5. The first clause just specifies that we are able to conclude `proveIf(S2,S1)` if we are not able to derive `not_proveIf(S2,S1)`. This predicate, in its turn, can only be derived if it is the case that, for some subexponential `SE` none of the conditions in Lemma 2 are satisfied. Each condition is specified using a clause.

Condition 1 (same formulas) states that if the subexponential is linear[5] then the contexts should be the same. This is achieved by using the aggregate `#count`

[5] We are currently not considering affine or relevant subexponentials since we could not find use cases yet.

```
proveIf(S2, S1) :- not not_proveIf(S2, S1),
                   ctx(_, _, _, S2, tree2), ctx(_, _, _, S1, tree1).

not_proveIf(S2,S1) :- ctx(_,SE,_,_,_), ctx(_,_,_,S1,tree1), ctx(_,_,_,S2,tree2),
                      not condition1(S2,S1,SE),
                      not condition2(S2,S1,SE),
                      not condition3(S2,S1,SE).

in_sequent_tree1(F,SE,TP,S,M) :- in(F,C,M), M > 0, ctx(C,SE,TP,S,tree1).
in_sequent_tree2(F,SE,TP,S,M) :- in(F,C,M), M > 0, ctx(C,SE,TP,S,tree2).

in_context_tree1(F,SE,TP,S) :- in_sequent_tree1(F,SE,TP,S,M).
in_context_tree2(F,SE,TP,S) :- in_sequent_tree2(F,SE,TP,S,M).

% Condition 1: SE is linear and the contexts are the same in both sequents
condition1(S2,S1,SE) :- ctx(_,SE,lin,S1,tree1), ctx(_,SE,lin,S2,tree2),
  #count{F : in_sequent_tree1(F,SE,lin,S1,N), in_sequent_tree2(F,SE,lin,S2,M),
             M != N} = 0,
  #count{G : in_context_tree1(G,SE,lin,S1), not in_context_tree2(G,SE,lin,S2)} = 0,
  #count{H : in_context_tree2(H,SE,lin,S2), not in_context_tree1(H,SE,lin,S1)} = 0.

% Condition 2: SE is unbounded and the context in S1 is a subset of the one in
% S2 (i.e., formulas can be weakened to get S2)
condition2(S2,S1,SE) :- ctx(_,SE,unb,S1,tree1), ctx(_,SE,unb,S2,tree2),
  #count{F : in_sequent_tree1(F,SE,unb,S1,N), in_sequent_tree2(F,SE,unb,S2,M),
             N > M} = 0,
  #count{I : in_context_tree1(I,SE,unb,S1), not in_context_tree2(I,SE,unb,S2)} = 0.

% Condition 3: all formulas occurring in S1 but not in S2, are in S2 but in a
% greater unbounded subexponential
condition3(S2,S1,SE) :- ctx(_,SE,TP,S1,tree1), ctx(_,SE,TP,S2,tree2),
  #count{F : in_context_tree1(F,SE,TP,S1), not in_context_tree2(F,SE,TP,S2),
             in_unbctx_geq(F,SE,S2)} = X,
  #count{G : in_context_tree1(G,SE,TP,S1), not in_context_tree2(G,SE,TP,S2)} = X.

in_unbctx_geq(F,SE,S) :- geq(SE2,SE), in_context_tree2(F,SE2,unb,S).

% If not all the leaves of the second tree are provable, no models are generated
:- not ok.
```

Fig. 5: Answer-set program for checking whether an open sequent `S2` in derivation schema `tree2` is provable when the open sequent `S1` in derivation schema `tree1` is provable.

and checking if the number of formulas with different multiplicity and the number of formulas that occur in one sequent but not in the other are all zero. This guarantees both multi-sets to be equal.

Condition 2 (subset of unbounded formulas) states that if the subexponential is unbounded, the context `S2` may have more formulas than the context in `S1`. This is checked by counting the number of different formulas that occur in `S1` more times than in `S2` and the number of formulas occurring in `S1` but not in `S2`. Both these numbers should be zero.

Condition 3 (upwardly closed) states that if a formula occurs in `SE` in sequent `S1` but not in `SE` in sequent `S2`, provability is still guaranteed if this formula occurs in an `SE2` greater than `SE` in `S2`. Again this check is performed using the `#count` aggregate and making sure that the number of different formulas occurring in `S1` but not in `S2` is equal to the number of different formulas with the same properties and also occurring in `S2` at a greater (unbouned) subexponential.

Finally, the last constraints enforces that a model is only generated if `ok` is derivable, that is, all sequents in `tree2` can be proved given the provability of

some sequent in `tree1`. Given the explanation above, the proof of the following theorem is immediate:

Theorem 4 *Let $\langle \Xi_1, M_1 \rangle$ and $\langle \Xi_2, M_2 \rangle$ be two derivation schemas whose derivations correspond to Ξ_1' and Ξ_2'. Let $\mathcal{P}$ be the answer-set program constructed from $\langle \Xi_1, M_1 \rangle$ and $\langle \Xi_2, M_2 \rangle$ as described above. If $\mathcal{P}$ has a model, then the implication derivability problem for Ξ_1' and Ξ_2' has a positive answer.*

5 Extracting Reader Friendly Figures

This section specifies the rewriting algorithm, which is basically a process of transforming a derivation schema $\langle \Xi, M \rangle$ into an object logic derivation. It is important to remind that the same Ξ may map to different object logic derivations depending on M. The function responsible to traverse the proof tree was omitted because it only consists of a usual implementation of a post-order depth-first search algorithm. Hence, the algorithm is applied top-down in the tree, rewriting each sequent. The rewriting algorithm processes the sequents in the tree in post-order. Each time a new sequent is processed, a rewriting table is updated. This table maps context variables Γ_s^i to a pair consisting of a list of other context variables and a list of formulas. During the whole process the following invariant is maintaned: each context variable is only rewritten to context variables created on the leafs. Therefore, when processing a constraint, all the information is propagated from the leaves. At the end of the process, the rewriting table will contain all that is necessary to rewrite the derivation. The function for rewriting a sequent is described in Algorithm 1. To illustrate the process, consider the schema derivation and set of constraints $\langle \Xi, M \rangle$ in Figure 6[6].

Since we are using the one-sided version of linear logic with subexponentials, the derivation schema is one-sided. However, the encoded calculus of the object logic may be two-sided. Thus, to take that into consideration the system needs more information from the user, who should specify which type of formulas (of the form $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$ or both) and how many (a single or many formulas) a subexponential context is supposed to have. In this case, in particular, this is done by adding the following declarations to the specification:

```
subexpctx l many lft.    subexpctx r one rght.
```

The declaration on the left specifies that the contexts for the subexponential l have many formulas, but all of them are of the form $\lfloor \cdot \rfloor$. Similarly for the declaration on the right, but in this case only one formula is allowed. The rewriting algorithm will create object logic contexts only for subexponentials declared this way. The rewriting proceeds as follows: initially, for each sequent, we choose the constraints $in(\_, \Gamma)$, $emp(\Gamma)$, $union(\_, \_, \Gamma)$ and $minus(\_, \_, \Gamma)$ such that Γ is a context variable in a sequent. Constraints are associated to sequents in this way. The first sequent processed is the leftmost closed leaf. There are two constraints associated to this sequent: $in(\lfloor A \wedge B \rfloor), \Gamma_l^1, 1)$ and $emp(\Gamma_r^1)$. Since the hashtable is empty, the context Γ_l^1 has no rewriting rule. Therefore we add the following to the table (line 11):

[6] This derivation is adapted and does not correspond exactly to Quati's output. On the system, many LaTeX restrictions can't be avoided.

Algorithm 1 Rewriting algorithm

```
1: function COMPUTE_REWRITE_SEQUENT(sequent, model, rewrite_ht)
2:     for each Γ in sequent.contexts do
3:         if sequent is a leaf then
4:             for each constraint cstr related to Γ in model do
5:                 if cstr = EMP(Γ) then
6:                     if Γ was not rewritten yet then
7:                         rewrite_ht.add Γ ([·], [·])
8:                 if cstr = IN(F, Γ, n) then
9:                     if Γ was not rewritten yet then
10:                        if sequent is a closed leaf and Γ is bounded then
11:                            rewrite_ht.add Γ ([·], [F_1, ..., F_n])
12:                        else
13:                            rewrite_ht.add Γ ([Γ_new], [F_1, ..., F_n])
14:                            where Γ_new is a fresh context variable
15:                    else
16:                        ([Γ_1, ..., Γ_j], [F_1', ..., F_k']) ← rewrite_ht.get Γ
17:                        if F is different from every F_i' in [F_1', ..., F_k'] then
18:                            rewrite_ht.replace Γ ([Γ_1, ..., Γ_j], [F_1', ..., F_k', F_1, ..., F_n])
19:        else
20:            for each constraint cstr related to Γ in model do
21:                if cstr = EMP(Γ) then
22:                    if Γ was not rewritten yet then
23:                        rewrite_ht.add Γ ([·], [·])
24:                    else
25:                        ([Γ_1, ..., Γ_j], [·]) ← rewrite_ht.get Γ
26:                        for each Γ_i in [Γ_1, ..., Γ_j] do
27:                            rewrite_ht.replace Γ_i ([·], [·])
28:                if cstr = UNION(Γ_1, Γ_2, Γ) then
29:                    if Γ was not rewritten yet then
30:                        if Γ_1 was not rewritten yet then
31:                            rwt_1 ← ([Γ_1], [·])
32:                        else
33:                            rwt_1 ← rewrite_ht.get Γ_1
34:                        if Γ_2 was not rewritten yet then
35:                            rwt_2 ← ([Γ_2], [·])
36:                        else
37:                            rwt_2 ← rewrite_ht.get Γ_2
38:                        rewrite_ht.add Γ (rwt_1 ∪ rwt_2)
39:                if cstr = SETMINUS(Γ_0, F, Γ) then
40:                    (sub_lst_0, formulas_0) ← rewrite_ht.get Γ_0
41:                    if Γ was not rewritten yet then
42:                        rewrite_ht.add Γ (sub_lst_0, formulas_0 − F)
43:                    else
44:                        (sub_lst, formulas) ← rewrite_ht.get Γ
45:                        if sub_lst ≠ sub_lst_0 and formulas = formulas_0 − F then
46:                            unify_variables (sub_lst_0, sub_lst, rewrite_ht)
```

$$\Gamma_l^1 \to ([\cdot], [\lfloor A \wedge B \rfloor])$$

Furthermore, Γ_r^1 has also no rewriting rule in the table, so line 7 is executed and the table is updated to:

$$\Gamma_l^1 \to ([\cdot], [\lfloor A \wedge B \rfloor]) \qquad\qquad \Gamma_r^1 \to ([\cdot], [\cdot])$$

$$
\dfrac{
\dfrac{}{\Gamma^0_\infty;\Gamma^1_l;\Gamma^1_r;\Gamma^1 \Downarrow \lfloor A \wedge B\rfloor^\perp}
\quad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\Gamma^0_\infty;\Gamma^5_l;\Gamma^3_r;\Gamma^3 \Downarrow \lfloor C \supset D\rfloor^\perp}
\quad
\dfrac{
\dfrac{\dfrac{\Gamma^0_\infty;\Gamma^7_l;\Gamma^7_r;\Gamma^5 \Uparrow}{\Gamma^0_\infty;\Gamma^7_l;\Gamma^5_r;\Gamma^5 \Uparrow ?^r\lceil C\rceil}}{\Gamma^0_\infty;\Gamma^7_l;\Gamma^5_r;\Gamma^5 \Downarrow !^l ?^r\lceil C\rceil}
\quad
\dfrac{\dfrac{\Gamma^0_\infty;\Gamma^9_l;\Gamma^6_r;\Gamma^6 \Uparrow}{\Gamma^0_\infty;\Gamma^8_l;\Gamma^6_r;\Gamma^6 \Uparrow ?^l\lfloor D\rfloor}}{\Gamma^0_\infty;\Gamma^8_l;\Gamma^6_r;\Gamma^6 \Downarrow ?^l\lfloor D\rfloor}
}{\Gamma^0_\infty;\Gamma^6_l;\Gamma^4_r;\Gamma^4 \Downarrow !^l ?^r\lceil C\rceil \otimes ?^l\lfloor D\rfloor}
}{\Gamma^0_\infty;\Gamma^4_l;\Gamma^2_r;\Gamma^2 \Downarrow \lfloor C \supset D\rfloor^\perp \otimes !^l ?^r\lceil C\rceil \otimes ?^l\lfloor D\rfloor}
}{\Gamma^0_\infty;\Gamma^4_l;\Gamma^2_r;\Gamma^2 \Uparrow}
}{\Gamma^0_\infty;\Gamma^3_l;\Gamma^2_r;\Gamma^2 \Uparrow ?^l\lfloor B\rfloor}
}{\Gamma^0_\infty;\Gamma^2_l;\Gamma^2_r;\Gamma^2 \Uparrow ?^l\lfloor A\rfloor\, ?^l\lfloor B\rfloor}
}{\Gamma^0_\infty;\Gamma^2_l;\Gamma^2_r;\Gamma^2 \Uparrow ?^l\lfloor A\rfloor \parr ?^l\lfloor B\rfloor}
}{\Gamma^0_\infty;\Gamma^2_l;\Gamma^2_r;\Gamma^2 \Downarrow ?^l\lfloor A\rfloor \parr ?^l\lfloor B\rfloor}
}{\dfrac{\Gamma^0_\infty;\Gamma^0_l;\Gamma^0_r;\Gamma^0 \Downarrow \lfloor A \wedge B\rfloor^\perp \otimes ?^l\lfloor A\rfloor \parr ?^l\lfloor B\rfloor}{\Gamma^0_\infty;\Gamma^0_l;\Gamma^0_r;\Gamma^0 \Uparrow}}
$$

$$
M = \left\{
\begin{array}{c}
union(\Gamma^1_l,\Gamma^2_l,\Gamma^0_l), union(\Gamma^1_r,\Gamma^2_r,\Gamma^0_r), union(\Gamma^1,\Gamma^2,\Gamma^0), \\
union(\Gamma^5_l,\Gamma^6_l,\Gamma^4_l), union(\Gamma^3_r,\Gamma^4_r,\Gamma^2_r), union(\Gamma^3,\Gamma^4,\Gamma^2), \\
union(\Gamma^7_l,\Gamma^8_l,\Gamma^6_l), union(\Gamma^5_r,\Gamma^6_r,\Gamma^4_r), union(\Gamma^5,\Gamma^6,\Gamma^4), \\
minus(\Gamma^4_l,\lfloor B\rfloor,\Gamma^3_l), minus(\Gamma^3_l,\lfloor A\rfloor,\Gamma^2_l), minus(\Gamma^7_r,\lceil C\rceil,\Gamma^5_r), \\
minus(\Gamma^9_l,\lfloor D\rfloor,\Gamma^8_l), emp(\Gamma^1_r), emp(\Gamma^1), emp(\Gamma^3_r), \\
emp(\Gamma^3), emp(\Gamma^5_r), emp(\Gamma^5), in(\lfloor A\rfloor,\Gamma^6_l,1), in(\lfloor A\rfloor,\Gamma^4_l,1), \\
in(\lfloor A\rfloor,\Gamma^8_l,1), in(\lfloor A\rfloor,\Gamma^3_l,1), in(\lfloor A\rfloor,\Gamma^9_l,1), \\
in(\lfloor B\rfloor,\Gamma^6_l,1), in(\lfloor B\rfloor,\Gamma^4_l,1), in(\lfloor B\rfloor,\Gamma^8_l,1), \\
in(\lfloor B\rfloor,\Gamma^9_l,1), in(\lceil C\rceil,\Gamma^7_r,1), in(\lfloor D\rfloor,\Gamma^9_l,1), \\
in(\lfloor A \wedge B\rfloor),\Gamma^1_l,1), in(\lfloor A \wedge B\rfloor),\Gamma^0_l,1), in(\lfloor C \supset D\rfloor,\Gamma^2_l,1), \\
in(\lfloor C \supset D\rfloor,\Gamma^0_l,1), in(\lfloor C \supset D\rfloor,\Gamma^5_l,1), \\
in(\lfloor C \supset D\rfloor,\Gamma^4_l,1), in(\lfloor C \supset D\rfloor,\Gamma^3_l,1)
\end{array}
\right\}
$$

Fig. 6: Schema derivation Ξ and its set of constraints.

The next sequent to be processed is the other closed leaf. This case is pretty much the same as the case before, lines 7 and 11 are executed and the rewriting table is updated to:

$$
\begin{array}{ll}
\Gamma^1_l \rightarrow ([\cdot],[\lfloor A \wedge B\rfloor]) & \Gamma^1_r \rightarrow ([\cdot],[\cdot]) \\
\Gamma^5_l \rightarrow ([\cdot],[\lfloor C \supset D\rfloor]) & \Gamma^3_r \rightarrow ([\cdot],[\cdot])
\end{array}
$$

Now it is time to process the leftmost open leaf. There is only one constraint associated to this sequent, which is $in(\lceil C\rceil,\Gamma^7_r,1)$. Line 13 is executed because Γ^7_r was not rewritten yet. Γ_{new} must be a fresh context variable, so it is initially calculated as the maximum index of a context appearing in the derivation plus 1. Then, we continue to increase this number as it is necessary. In this case, 9 is the greatest index, therefore the hashtable is updated to:

$$
\begin{array}{ll}
\Gamma^1_l \rightarrow ([\cdot],[\lfloor A \wedge B\rfloor]) & \Gamma^1_r \rightarrow ([\cdot],[\cdot]) \\
\Gamma^5_l \rightarrow ([\cdot],[\lfloor C \supset D\rfloor]) & \Gamma^3_r \rightarrow ([\cdot],[\cdot]) \\
\Gamma^7_r \rightarrow ([\Gamma^{10}_r],[\lceil C\rceil]) &
\end{array}
$$

The next sequent to be processed is the last leaf. In this case, there are three constraints associated to this sequent: $in(\lfloor A \rfloor, \Gamma_l^9, 1)$, $in(\lfloor B \rfloor, \Gamma_l^9, 1)$ and $in(\lfloor D \rfloor, \Gamma_l^9, 1)$. Notice that the order of processing doesn't really matter. For the sake of the example, let's consider the order in M. The first constraint to be processed is $in(\lfloor A \rfloor, \Gamma_l^9, 1)$, and we get into the same case as the last sequent. Line 13 is executed and $\Gamma_r^9 \to \left([\Gamma_r^{11}], [\lfloor A \rfloor]\right)$ is added to the hashtable. When the second one is processed, $in(\lfloor B \rfloor, \Gamma_l^9, 1)$, Γ_l^9 already exists as a key in the hashtable. Then, lines 16 to 18 are executed. Basically, as $\lfloor A \rfloor \neq \lfloor B \rfloor$, it just adds B to the list of formulas and keeps the context variable Γ_l^{11} the same. Similarly, the same thing happens when $in(\lfloor D \rfloor, \Gamma_l^9, 1)$ is processed. In the end, the hashtable will contain the following:

$$\Gamma_l^1 \to ([\cdot], [\lfloor A \wedge B \rfloor]) \qquad \Gamma_r^1 \to ([\cdot], [\cdot])$$
$$\Gamma_l^5 \to ([\cdot], [\lfloor C \supset D \rfloor]) \qquad \Gamma_r^3 \to ([\cdot], [\cdot])$$
$$\Gamma_r^7 \to \left([\Gamma_r^{10}], [\lceil C \rceil]\right) \qquad \Gamma_r^9 \to \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor, \lfloor D \rfloor]\right)$$

The sequent just below the last one is the next to be processed. There is only one constraint associated to this sequent, $minus(\Gamma_l^9, \lfloor D \rfloor, \Gamma_l^8)$. This case is pretty straightforward, we just need to remove a formula from the content of Γ_l^9 and add to the hashtable this content associated to the key Γ_l^8. Line 42 is executed and the resulting hashtable is:

$$\Gamma_l^1 \to ([\cdot], [\lfloor A \wedge B \rfloor]) \qquad \Gamma_r^1 \to ([\cdot], [\cdot])$$
$$\Gamma_l^5 \to ([\cdot], [\lfloor C \supset D \rfloor]) \qquad \Gamma_r^3 \to ([\cdot], [\cdot])$$
$$\Gamma_r^7 \to \left([\Gamma_r^{10}], [\lceil C \rceil]\right) \qquad \Gamma_r^9 \to \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor, \lfloor D \rfloor]\right)$$
$$\Gamma_l^8 \to \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor]\right)$$

The same cases will happen until we reach the sequent with the following context variables: $\Gamma_l^6; \Gamma_r^4$. There are two constraints associated to this sequent, $union(\Gamma_l^7, \Gamma_l^8, \Gamma_l^6)$ and $union(\Gamma_r^5, \Gamma_r^6, \Gamma_r^4)$. In the first case, lines 31 and 37 are executed because Γ_l^7 was not rewritten yet, so $rwt_1 = \left([\Gamma_l^7], [\cdot]\right)$ and Γ_l^8 was already rewritten, so $rwt_2 = \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor]\right)$. Therefore, updating the hashtable we get:

$$\Gamma_l^1 \to ([\cdot], [\lfloor A \wedge B \rfloor]) \qquad \Gamma_r^1 \to ([\cdot], [\cdot])$$
$$\Gamma_l^5 \to ([\cdot], [\lfloor C \supset D \rfloor]) \qquad \Gamma_r^3 \to ([\cdot], [\cdot])$$
$$\Gamma_r^7 \to \left([\Gamma_r^{10}], [\lceil C \rceil]\right) \qquad \Gamma_r^9 \to \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor, \lfloor D \rfloor]\right)$$
$$\Gamma_l^8 \to \left([\Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor]\right) \qquad \Gamma_l^6 \to \left([\Gamma_l^7, \Gamma_l^{11}], [\lfloor A \rfloor, \lfloor B \rfloor]\right)$$

The processing of the remnant sequents are all similar to the previous cases. This derivation is quite simple because the inference rules are multiplicative, if it were not for this, it becomes a bit tricky to process the *minus* constraints. Suppose that we need to process the following constraints, in this order: $in\left(\lfloor A \rfloor, \Gamma_l^{10}\right)$, $in\left(\lfloor B \rfloor, \Gamma_l^{11}\right)$, $minus\left(\Gamma_l^{10}, \lfloor A \rfloor, \Gamma_l^9\right)$, $minus\left(\Gamma_l^{11}, \lfloor B \rfloor, \Gamma_l^9\right)$. You can assume that Γ_l^{10} and Γ_l^{11} belong to sequents that are open leafs and 11 is the greatest index of the proof tree. The resolution of the first three constraints are straightforward and were already discussed. Thus, the resulting hashtable is the following:

$$\Gamma_l^{10} \to ([\Gamma_l^{12}], [\lfloor A \rfloor]) \qquad \Gamma_l^{11} \to ([\Gamma_l^{13}], [\lfloor B \rfloor])$$
$$\Gamma_l^{9} \to ([\Gamma_l^{12}], [\cdot])$$

However, it is not clear how we should process $minus\left(\Gamma_l^{11}, \lfloor B \rfloor, \Gamma_l^{9}\right)$, because Γ_l^9 was already rewritten. First of all, we need to check if the conditions are satisfied so the context variables can be unified. In this case, $sub\_lst_0 = [\Gamma_l^{13}]$, $formulas_0 = [\lfloor B \rfloor]$, $sub\_lst = [\Gamma_l^{12}]$ and $formulas = [\cdot]$. We can easily notice that the conditions in line 45 are satisfied and line 46 is executed. In this particular case, the function *unify_variables* will generate a fresh new context variable, Γ_l^{14} and will replace every occurrence of Γ_l^{12} and Γ_l^{13} in the hashtable to Γ_l^{14}. In the end, the hashtable will end up like this:

$$\Gamma_l^{10} \to ([\Gamma_l^{14}], [\lfloor A \rfloor]) \qquad \Gamma_l^{11} \to ([\Gamma_l^{14}], [\lfloor B \rfloor])$$
$$\Gamma_l^{9} \to ([\Gamma_l^{14}], [\cdot])$$

After all, let's assume we have finished processing the constraints. We need to create a copy of the proof tree, considering only subexponentials declared as we demonstrated in the beginning of this section. Notice that in our example, we will consider only l and r. Next, we traverse the whole tree again, going through all subexponentials and replacing them for their associated content in the hashtable. Finally, the only thing left to do is to collapse some sequents in order to consider only sequents where inference rules were applied and, obviously, the conclusion. Ultimately, the following figure is Quati's output of our example (the context variables are normalized after the rewriting step):

$$\dfrac{\dfrac{\Gamma_l^1 \vdash c \quad \Gamma_l^2, d, b, a \vdash C}{\Gamma_l^1, \Gamma_l^2, imp(c)(d), b, a \vdash C}\, imp_l}{\Gamma_l^1, \Gamma_l^2, and(a)(b), imp(c)(d) \vdash C}\, and_l \quad \rightsquigarrow \quad \dfrac{\Gamma_l^1 \vdash c \quad \dfrac{\Gamma_l^2, b, a, d \vdash C}{\Gamma_l^2, and(a)(b), d \vdash C}\, and_l}{\Gamma_l^1, \Gamma_l^2, imp(c)(d), and(a)(b) \vdash C}\, imp_l$$

6 Implementation

This method of finding permutations of inference rules is implemented in the system SELLF[7], an implementation of linear logic with subexponentials in OCaml. This system was implemented with the goal of being a framework for reasoning about sequent calculi given their linear logic encoding. The cut elimination check, described in [23], is also implemented in SELLF. It contains the encoding of several sequent calculi for different logics.

Quati can be used in a more user-friendly way through its web-interface[8]. There are some preloaded examples but the user is free to modify them or type their own. Once the rules are generated, the user can check for the permutation

[7] https://github.com/meta-logic/sellf

[8] http://quati.gisellereis.com/

of any two of them. If they are curious to see what is under the hood, they can see the bipole derivations with the set of constraints describing each rule. All the generated figures on the website can be downloaded as a LaTeX file.

The command line interface, available in SELLF, offers a little more functionality, such as the generation of all bipoles or all possible permutations at once. The user can also generate a permutability graph, which indicates which rule permutes over which (in beta).

7 Related Work

Proof transformations and permutability of inferences in particular have been addressed by several other authors, either as the means to an end or as the end itself. The first work we could find is that of Kleene [11], which investigates the permutability of inferences in Getnzen's sequent calculi for classical and intuitionistic logic. He lists all possible cases (not the transformations though), giving counter-examples and discussing those that do not work.

Such lemmas became more relevant in recent times with the automation of proof search. By identifying rules that can be eagerly applied or avoiding different orders which are shown to be equivalent, one can really reduce the proof search space. Along these lines we can find the work of Galmiche and Perrier [6]. The authors have analyzed the permutation of all rules in linear logic, which is nicely summarized in a table on [6, page 9].

The previously cited works only look at one calculus though. A more universal approach is that of Lutovac and Harland [13]. The authors define a general format for sequents and generic conditions for permutability based on the components of inference rules. The attempt to capture all, or at least many, sequent calculi is in line with our ideas. In fact, we may be able to use some of their ideas to bridge the remaining gap in our framework: the encoding of sequent rules as bipoles. Their investigations are purely theoretical though, with no accompanying implementation.

Automated proofs of cut-elimination often have to use permutation lemmas, but each implementation and each system is different, and there seem to be no general way of proving those lemmas for every calculi. The approach followed by Twelf and the LF framework family is worth noting due to its simplicity [25]. This comes from the fact that the context is left implicit and handled by the framework itself. Unfortunately, this makes it very hard to generalize the method for other calculi where the context has different structural properties. In order to have a similarly elegant solution for linear logic, for example, one has to move to a linear logical framework. Also worth mentioning are the implementations of cut-elimination proofs in Abella [2]. To the best of our knowledge, this property is proved for at least three different systems. The first one, for intuitionistic logic, follows closely the approach of [25] and thus suffers from the same problem. The second one, for Harrop formulas, uses an explicit context and the permutations are done in the cut-elimination proof directly. The third one, for linear logic, was recently developed by two authors of this paper and contains many inversion lemmas. All the permutations performed are within the proofs themselves, and not as lemmas. It was not clear how to either transform these to lemmas or generalize to arbitrary calculi.

8 Conclusion

We have described a systematic method for finding out whether two inference rules in sequent calculus permute given their encoding as a linear logic bipole (possibly with subexponentials). This method is sound, but not complete, due to the inherent undecidability of whether a proof of a sequent $\mathcal{S}$ implies the existence of a proof of another sequent $\mathcal{S}'$. Nevertheless we believe that the automation of this otherwise repetitive and combinatorially intense task is helpful for proof theorists. One can concentrate on the permutations that failed and verify whether this is because the transformation relies on an implicit lemma or if the rules actually do not permute.

The method consists of using the bipoles encoding the rules to construct a schema derivation associated with a set of constraints, imposed by the application of linear logic rules. The models of this set of constraints in a logic program, together with the schema derivation, describe each possible result of the rule application on the object level. By repeating this procedure we can get a derivation of two bipoles (in linear logic) or two rules (in the object logic). In order to decide if the provability of one derivation implies the provability of the other, we again resort to sets of constraints and a logic program. This time the model does not matter, and we are only interested if there is a solution or not. At each step, we can take a schema derivation its model and reconstruct the corresponding derivation on the object logic using a rewriting algorithm. This algorithm is used to show the permutations between the rules and also the inference rules of the object logic given the encoding (a good sanity check for the encoding itself).

The whole procedure can be done with the click of a button, and using the rewriting algorithm the linear logic encoding could be completely transparent if it weren't for the initial encoding of the sequent calculus. We have plans to automate this encoding, and completely hide framework from the user, as we realise that learning a new logic to use the system is an annoying overhead. In spite of our investigations, it is still not yet clear what is an elegant way of doing this.

During our investigations of permutation lemmas, we have realised that many transformations rely on invertibility lemmas. For example, $\&_r$ permutes up $⅋_r$ only because $⅋_r$ is invertible. As of now, Quati will not identify this permutation, failing at the case where $⅋_r$ is applied to only one premise of $\&_r$. Technically this is correct, but in practice we use the fact that $⅋_r$ is invertible to argue that this case can be transformed into the others. The identification of invertible rules will not only improve the permutability check, but it is also interesting by itself.

As of now, our method is not able to identify permutations of the cut rule or quantifiers, but we are working to include support for them in the future. Permutations using cut seem to be a simply a matter of adapting our way of handling encodings. We need to treat if differently because the cut encoding has no "head". Quantifier permutations are more tricky because of eigenvariable violations. As we do not have an explicit term signature, quati would not be able to tell when a violation might occur.

Lastly we would like to extract proof objects from the permutations found so that it can be formally checked by a theorem prover. We recently encoded a cut-elimination proof in the interactive theorem prover Abella [3] and a lot of time was spend in proving permutations and invertibility lemmas, until converging to a reasonable encoding of multi-set operations. We can use this multi-set library to try to generate the proofs automatically given the permutations found in Quati.

References

1. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation **2**(3), 297–347 (1992)
2. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: A system for reasoning about relational specifications. Journal of Formalized Reasoning **7**(2), 1–89 (2014). DOI 10.6092/issn.1972-5787/4650
3. Chaudhuri, K., Lima, L., Reis, G.: Formalized meta-theory of sequent calculi for substructural logics. In: Workshop on Logical and Semantic Frameworks, with Applications (LSFA) (2016)
4. Crolard, T.: Subtractive logic. Theor. Comput. Sci. **254**(1-2), 151–185 (2001)
5. Danos, V., Joinet, J.B., Schellinx, H.: The structure of exponentials: Uncovering the dynamics of linear logic proofs. In: G. Gottlob, A. Leitsch, D. Mundici (eds.) Kurt Gödel Colloquium, *LNCS*, vol. 713, pp. 159–171. Springer (1993)
6. Galmiche, D., Perrier, G.: On proof normalization in linear logic. Theor. Comput. Sci. **135**(1), 67–110 (1994)
7. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: ICLP, pp. 579–597 (1990)
8. Gentzen, G.: Investigations into logical deductions. In: M.E. Szabo (ed.) The Collected Papers of Gerhard Gentzen, pp. 68–131. North-Holland, Amsterdam (1969)
9. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**, 1–102 (1987)
10. Herbrand, J.: Recherches sur la théorie de la Démonstration. Ph.D. thesis, University of Paris (1930)
11. Kleene, S.C.: Permutabilities of inferences in Gentzen's calculi LK and LJ. Memoirs of the American Mathematical Society **10**, 1–26 (1952)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**, 499–562 (2006). DOI http://doi.acm.org/10.1145/1149114.1149117
13. Lutovac, T., Harland, J.: A contribution to automated-oriented reasoning about permutability of sequent calculi rules (2013). Submitted to Computer Science and Information Systems
14. Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems (2013). To appear in TCS.
15. Miller, D., Saurin, A.: From proofs to focused proofs: a modular proof of focalization in linear logic. In: J. Duparc, T.A. Henzinger (eds.) CSL 2007: Computer Science Logic, *LNCS*, vol. 4646, pp. 405–419. Springer (2007)
16. Niemelä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal lp. In: LPNMR, pp. 421–430 (1997)
17. Nigam, V.: Exploiting non-canonicity in the sequent calculus. Ph.D. thesis, Ecole Polytechnique (2009)
18. Nigam, V.: On the complexity of linear authorization logics. In: LICS, pp. 511–520. IEEE (2012)
19. Nigam, V.: A framework for linear authorization logics. Theoretical Computer Science **536**(0), 21 – 41 (2014). DOI http://dx.doi.org/10.1016/j.tcs.2014.02.018
20. Nigam, V., Miller, D.: Algorithmic specifications in linear logic with subexponentials. In: ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), pp. 129–140 (2009)
21. Nigam, V., Miller, D.: A framework for proof systems. J. Autom. Reasoning **45**(2), 157–188 (2010)
22. Nigam, V., Olarte, C., Pimentel, E.: A general proof system for modalities in concurrent constraint programming. In: P.R. D'Argenio, H.C. Melgratti (eds.) CONCUR, *LNCS*, vol. 8052, pp. 410–424. Springer (2013)
23. Nigam, V., Pimentel, E., Reis, G.: An extended framework for specifying and reasoning about proof systems. Journal of Logic and Computation (2014). DOI http://dx.doi.org/10.1093/logcom/exu029. Published online. Special issue in honor of Roy Dyckhoff.
24. Olarte, C., Pimentel, E., Nigam, V.: Subexponential concurrent constraint programming. Theor. Comput. Sci. **606**, 98–120 (2015)
25. Pfenning, F.: Structural cut elimination. In: Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, pp. 156–166. IEEE Computer Society Press, San Diego, California (1995)
26. Rauszer, C.: A formalization of the propositional calculus h-b logic. Studia Logica **33**, 23–34 (1974)
27. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Cambridge Univ. Press (1996)