Checking Proof Transformations with ASP

VIVEK NIGAM and LEONARDO LIMA
Universidade Federal da Paraíba, Brazil
(*e-mail:* {vivek.nigam, leonardo.alfs}@gmail.com)

GISELLE REIS
Vienna University of Technology, Austria
(*e-mail:* giselle@logic.at)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Proof transformation is an important proof theoretic technique that has been used for showing a number of foundational results about proof systems. For instance, it is used for showing the admissibility of the cut-rule and the completeness of proof search strategies, such as uniform provability and the focusing discipline. However, in order to check the validity of a proof transformation, such as when one inference rule permutes over another, one needs to consider the combination of how inference rules may be applied. Therefore, checking the correctness of proof transformations is prone to human error. This paper offers the means to automatize the check of such transformations by using Answer Set Programming (ASP).

KEYWORDS: Answer Set Programming, Automatic Verification, Logical Frameworks, Proof Theory

1 Introduction

Proof transformation is a powerful technique used in proving many foundational results about proof systems. For instance, one demonstrates the admissibility of the cut-rule (Gentzen 1969) by showing how to transform a proof with cuts into a proof without cuts. Similarly, in order to show the completeness of a proof search strategy, such as uniform provability (Miller et al. 1991) and the focusing discipline (Andreoli 1992), one demonstrates how to transform an arbitrary (cut-free) proof into another (cut-free) proof that follows the given proof search strategy.

However, it is often a tedious task to verify whether a proof transformation is valid, specially when there is a great number of cases to consider. For example, in the proof of completeness of the focusing discipline and of the admissibility of the cut rule, one needs to show that some rules permute over other rules (Miller and Saurin 2007; Gentzen 1969). These results are called *permutation lemmas* and they are obtained by showing that any proof where two rules are applied in a determined order can be transformed into another proof where these rules are permuted. For example: a linear logic (Girard 1987) proof ending with the derivation to the left, where $\otimes_R$ is applied last, can (always) be transformed into another linear logic proof ending with the derivation to the right, where the $\&_R$ is applied last:

$$\dfrac{\overset{\Xi_1}{\Gamma \vdash \Delta, A} \quad \dfrac{\overset{\Xi_2}{\Gamma' \vdash \Delta', B, C} \quad \overset{\Xi_3}{\Gamma' \vdash \Delta', B, D}}{\Gamma' \vdash \Delta', B, C\& D}\,[\&_R]}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \otimes B, C\& D}\,[\otimes_R] \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi_1}{\Gamma \vdash \Delta, A} \quad \overset{\Xi_2}{\Gamma' \vdash \Delta', B, C}}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \otimes B, C}\,[\otimes_R] \quad \dfrac{\overset{\Xi_1}{\Gamma \vdash \Delta, A} \quad \overset{\Xi_3}{\Gamma' \vdash \Delta', B, D}}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \otimes B, D}\,[\otimes_R]}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \otimes B, C\& D}\,[\&_R]$$

The proof transformation above is one of the many cases required in showing that any instance of a $\otimes_R$ rule can permute over any instance of a $\&_R$. In general, checking the correctness of such transformations involves checking that: (**Obligation 1**) all rules are correctly applied; and that (**Obligation 2**) the premises of the latter derivation can be proved using the proofs introducing the premises of the former derivation. For instance, in the case above, the proof Ξ_1 introducing the sequent $\Gamma \vdash \Delta, A$ in the former derivation can be used twice in the latter derivation. Similar permutation lemmas also appear in the proof of cut-elimination, in particular, when transforming a proof with cuts into a proof with *principal cuts* (Gentzen 1969).[1]

Although one can check by hand the validity of such proof transformations, this procedure is prone to human error as they have a *combinatorial* nature and therefore one can easily miss a case or another. For instance, the cut-elimination result for Bi-Intuitionistic logic (Rauszer 1974) given by Rauszer was later found to be incorrect (Crolard 2001) exactly because one of the permutation lemmas was in fact not true. A much better approach, therefore, is to automate the check of proof transformations.

As we demonstrate in this paper, Answer Set Programming (ASP) (Gelfond and Lifschitz 1990) can be easily used for checking the correctness of wide range of proof transformations. In our initial efforts, we tried to use functional programming to simulate all cases and check whether the Proof Obligations 1 and 2 are satisfied. However, as illustrated above, these problems require a combinatorial solution, for which our solution using ASP turned out to be a much simpler. In particular, we construct two Answer Set Programs (henceforth referred to as LP):

• The first program, $\mathcal{T}$, is constructed for a given inference rule, r, and specifies the set of its valid instances. We show that $\mathcal{T}$ is sound and complete in the sense that its set of *answer-sets* corresponds exactly to the set of all possible valid instances of the rule r, solving Obligation 1.

• For two given sequents, $\mathcal{S}_1$ and $\mathcal{S}_2$, the second program, $\mathcal{P}$, checks whether the sequent $\mathcal{S}_2$ is provable, when assuming that $\mathcal{S}_1$ is also provable. In particular, we show that if the program $\mathcal{P}$ has at least one answer-set, then it is possible to derive a proof of $\mathcal{S}_2$ from a proof of $\mathcal{S}_1$. This is enough to solve Obligation 2. As this problem is undecidable in general, we show the soundness of this program.

Another main advantage of using ASP is that it enables the use of powerful off-the-shelf provers (Niemelä and Simons 1997; Leone et al. 2006). We implemented a tool that takes the specification of a proof system and checks automatically which inference rules of the object-system permute over another rule. Whenever the tool can find a valid permutation it outputs the corresponding proof transformation, and whenever it cannot show that a rule permutes over another, it can output the cases that it failed to find a valid permutation. We used this tool to show a number of proof transformations. For instance, our tool checks all cases of the key permutation lemmas needed for showing the completeness of the focusing discipline (Andreoli 1992) and uniform proofs (Miller et al. 1991). Up to the best of our knowledge there is no such tool yet available.

After reviewing in Section 2 the proof theoretic and logic programming machinery needed in this paper, we propose in Section 3 an answer set program that specifies the derivations that are available from the proof system, proving the correctness and completeness of the specification. Section 4 contains the answer set program that checks whether a derivation is provable assuming the provability of another derivation. We also show that this check is sound. Section 5 elaborates

[1] In fact, permutation lemmas compose a great part of the cut-elimination proof, roughly half of the proof. See for example the proof in `http://twelf.org/wiki/Admissibility_of_cut`.

on how these two programs can be used to check whether a rule permutes over another and discusses our implementation and experimental results. Finally in Section 6, we discuss our tool and experimental results and conclude by pointing out related and future work.

2 Preliminaries

2.1 Sequents with Contexts and Inference Rules

We assume that reader is familiar with the basic terminology of proof theory and only introduce the terminology which will be used to guide our work. In general, a *sequent with contexts* (Andreoli 1992) is a generalization used in the context of logic programming and logical frameworks (Nigam and Miller 2010; Cervesato and Pfenning 2002) of the usual sequents in sequent calculus (Gentzen 1969). They are of the form

$$\Gamma_1 \mid \Gamma_2 \mid \cdots \mid \Gamma_n \vdash \Gamma_{n+1} \mid \Gamma_{n+2} \mid \cdots \mid \Gamma_{n+m}$$

containing $n + m$ *sequent contexts*. For example, in intuitionistic logic (LJ (Gentzen 1969)) sequents are of the form $\Gamma \vdash A$ with 2 contexts, one context to the left of the turnstyle and another to the right.

An *inference rule* is composed by one conclusion sequent and zero or more premises. These are normally written using *context variables*, *e.g.*, Γ, Δ, Θ, which may be instantiated with collections of formulas, and *formula variables*, *e.g.*, $A \wedge B$, $\Box A$, whose schema variables, A and B, can be instantiated with any formula.

For instance, the inference rules for $\wedge_R$ of intuitionistic logic and $\Box_r$ of the modal logic S4 are shown below:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ [\wedge_R] \qquad\qquad \frac{\Box\Gamma \vdash A, \diamond\Delta}{\Box\Gamma, \Gamma' \vdash \Box A, \diamond\Delta, \Delta'}\ [\Box_r]$$

The $\Box_r$ rule contains one premise, namely the sequent $\Box\Gamma \vdash A, \diamond\Delta$, and a conclusion sequent $\Box\Gamma, \Gamma' \vdash \Box A, \diamond\Delta, \Delta'$. It introduces the principal formula $\Box A$; A is an active formula. Any instance of the context variables $\Gamma, \Gamma', \Delta, \Delta'$ and the formula schema A in the rule above will correspond to a *valid* instance of this rule. Similarly for the $\wedge_R$ rule. In the case of the S4 system, we consider each sequent having *four* contexts: boxed formulas on the left, other formulas on the left, diamond formulas on the right and other formulas on the right. Whereas for intuitionistic logic sequents, only *two* contexts are needed: left and right.

A sequent calculus *proof system* $\mathcal{P}$ for a logic L is a set of inference rules such that the formulas derived by the application of these rules are exactly the formulas valid in L. We say that a sequent $\mathcal{S}$ is *provable or derivable* in $\mathcal{P}$ if there exists a derivation of $\mathcal{S}$ using the inference rules of $\mathcal{P}$ such that all leaves are *closed*, i.e., their topmost rule has no premises. A leaf (or premise) is said to be *open* if it is not closed.

Following the terminology normally used in the logical framework literature (Nigam and Miller 2010; Cervesato and Pfenning 2002), we classify sequent contexts in two ways: *bounded* and *unbounded*. Context variables appearing in a *bounded* context are instantiated by a *multiset* of formulas, which cannot be contracted nor weakened. Thus, when a multiple premise rule is applied, these formulas are split among the premises. This is the case for the contexts in rule $\otimes_R$ of linear logic, shown in the Introduction. Context variables appearing in an *unbounded* context are instantiated by *sets* of formulas, which can be contracted or weakened as much as needed. Consequently, they are copied among the premises of a rule, which is the case for Γ in the $\wedge_R$ rule above.

In order to distinguish different occurrences of a formula, we associate to each formula occurrence A a number. That is, two different occurrences of the same formula A are represented by $(A, i), (A, j)$, where $i \neq j$. For simplicity, we assume implicitly that different occurrences of a formula are distinguished this way.

We assume that the shape of sequents, *i.e.*, the number of contexts a sequent has and their classification as bounded and unbounded are given as part of the input. This classification will guide the specification of the Answer Set Programs we use to check more proof transformations (Section 4). For example, if a context is unbounded, then the following lemma is provable:

Lemma 1
Let $\mathcal{P}$ be a proof system, whose sequents are of the form $\mathcal{S} = \Gamma_1 \mid \Gamma_2 \mid \cdots \mid \Gamma_i \mid \cdots \mid \Gamma_n \vdash \Delta_1 \mid \Delta_2 \mid \cdots \mid \Delta_m$, where Γ_i is an unbounded context. Let $\Gamma'_i \supseteq \Gamma_i$ be a superset of Γ_i for every i. If the sequent $\mathcal{S}$ is provable in $\mathcal{P}$, then the sequent $\mathcal{S}'$ obtained from $\mathcal{S}$ by replacing Γ_i by Γ'_i is also provable in $\mathcal{P}$.

The proof is straightforward, as one simply needs to weaken the formulas in $\Gamma'_i \setminus \Gamma_i$ in $\mathcal{S}'$ and recover the sequent $\mathcal{S}$, which is assumed to be provable in $\mathcal{P}$. Such lemmas enhance the checking of proof transformation, in particular for checking Obligation 2, detailed above.

Finally, we notice that there are proof systems that have other types of contexts, such as contexts that behave as lists of formulas, that is, that the exchange rule is not always applicable (Pfenning and Simmons 2009). There are not many of these types of proof systems. We believe, however, that our machinery for checking proof transformations can be extended for them, but this is left out of the scope of this paper.

2.2 Answer Set Programming

Although we assume that the reader is familiar with Answer Set Programming (Gelfond and Lifschitz 1990), we review some of its basic definitions. 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 and $Body(r)$ is a finite set of literals. A rule with $Head = L$ and $Body(r) = \{L_1, \dots, L_n\}$ is written $L \leftarrow L_1, \dots, L_n$. An *interpretation* M of $\mathcal{K}$ is a subset of $\mathcal{K}$. An atomic formula, A, is true in M, written $M \models A$, if $A \in M$, otherwise false. A literal *not* A is true in M, written $M \models not\ A$, if $A \notin M$, otherwise false. An *Answer Set Program* (LP) is a set of rules. An interpretation M is an *answer set* of a LP P if $M' = least(P \cup \{not\_A \mid A \notin M\})$, where $M' = M \cup \{not\_A \mid A \notin M\}$ and *least* is the least 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 specifies the constraint that b cannot be true:

$$\bot \leftarrow b$$

Table 1. *List of atomic formulas used together with their denotations and their logical axiomatization $\mathcal{T}$. Following usual logic programming conventions, all non-predicate term symbols are assumed to be universally quantified, and we use commas, ",", for conjunctions and "$\leftarrow$" for the reverse implication.*

Alphabet	Denotation	Logic Specification
$in(F, \Gamma)$	$F \in \Gamma$	No theory.
$unitctx(F, \Gamma)$	$\Gamma = \{F\}$	(r1) $in(F, \Gamma) \leftarrow unitctx(F, \Gamma)$. (r2) $\bot \leftarrow in(F_1, \Gamma), unitctx(F, \Gamma), F_1 \neq F$.
$emp(\Gamma)$	$\Gamma = \emptyset$	(r3) $\bot \leftarrow in(F, \Gamma), emp(\Gamma)$.
$union(\Gamma^1, \Gamma^2, \Gamma)$	$\Gamma = \Gamma^1 \cup \Gamma^2$	(r4) $in(F, \Gamma) \leftarrow in(F, \Gamma^1), union(\Gamma^1, \Gamma^2, \Gamma)$. (r5) $in(F, \Gamma) \leftarrow in(F, \Gamma^2), union(\Gamma^1, \Gamma^2, \Gamma)$. (r6) $emp(\Gamma) \leftarrow emp(\Gamma^1), emp(\Gamma^2), union(\Gamma^1, \Gamma^2, \Gamma)$. (r7) $in(F, \Gamma^1) \leftarrow not\ in(F, \Gamma^2), in(F, \Gamma), union(\Gamma^1, \Gamma^2, \Gamma)$. (r8) $in(F, \Gamma^2) \leftarrow not\ in(F, \Gamma^1), in(F, \Gamma), union(\Gamma^1, \Gamma^2, \Gamma)$.

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

3 Checking the Validity of Derivations

Assume given a proof system, with inference rules detailed as before. For each rule in the proof system, we can construct a LP that specifies all its possible instances. The idea is that for each inference rule in the proof system, we associate an *inference skeleton* and a *set of atomic facts*. The skeleton specifies the tree structure of the inference rule, while the set of facts specifies how the formulas are moved along a derivation. From these, we can construct larger derivations.

The alphabet and the theory used to specify these derivations are shown in Table 1. The logic program that we need is very simple, with only eight rules: (r1), (r2), ..., (r8). These rules and the predicates in Table 1 specify in a declarative fashion the content of context variables, Γ, in a derivation. The encoding is all based on atomic formulas of the form $in(F, \Gamma)$, which specify that the formula F is in the context Γ.

The atomic formula $unitctx(F, \Gamma)$ specifies that the context Γ has a single formula F. The first rule (r1) specifies that $in(F, \Gamma)$, while the second rule (r2) is a constraint rule specifying that there is no other formula F' different from F in the context Γ.

In some situations, for instance, when specifying the linear logic initial rule (Girard 1987), we need to specify that some contexts are empty, which is done by using the atomic formula $emp(\Gamma)$. Rule (r3) is a constraint that specifies that no formula can be in an empty context.

The most elaborate specification are the rules (r4) – (r8), which specify the atomic formula $union(\Gamma^1, \Gamma^2, \Gamma)$, i.e. $\Gamma = \Gamma^1 \cup \Gamma^2$. The rules (r4) and (r5) specify that $\Gamma^1 \subseteq \Gamma$ and $\Gamma^2 \subseteq \Gamma$, that is, the occurrence[2] of a formula that is in Γ^i is also in Γ. The rule (r6) specifies that if both Γ^1 and Γ^2 are empty then so is Γ. The rules (r7) and (r8) specify that these contexts are bounded (see Section 2.1), that is, the union $\Gamma = \Gamma^1 \cup \Gamma^2$ is a multiset union. An occurrence of a formula in Γ either comes from Γ^1 or from Γ^2. Notice how we use the default negation to generate accordingly the splitting behavior of bounded contexts.

[2] Recall that as discussed in Section 2, we keep track of the occurrences of formulas.

Auxiliary Context Variables and Macros From these rules, we can specify more elaborate conditions on context variables and formula variables. For this we may need fresh auxiliary context variables, written Γ^j_{aux} with an *aux* subscript. Freshness is guaranteed, as usual, by using a counter that is incremented whenever one needs a new auxiliary context.

For example, the theory below, written $Unions(\Gamma, [\Gamma_1, \dots, \Gamma_n])$, specifies that $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_n$ by using auxiliary context variables as follows:

$$union(\Gamma_1, \Gamma_2, \Gamma^1_{aux}), union(\Gamma^1_{aux}, \Gamma_3, \Gamma^2_{aux}), \dots, union(\Gamma^{n-3}_{aux}, \Gamma_{n-1}, \Gamma^{n-2}_{aux}), union(\Gamma^{n-2}_{aux}, \Gamma_n, \Gamma)$$

It is easy to check from the specification of $union(\cdot, \cdot, \cdot)$ that indeed whenever $in(F, \Gamma_i)$ is true in an answer-set then so is $in(F, \Gamma)$, and conversely that if $in(F, \Gamma)$ is true in an answer-set then there is one $1 \leq i \leq n$ such that $in(F, \Gamma_i)$ is also true.

Similarly, we can define when a context variable $\Gamma = \{A_1, \dots, A_n\}$, is a collection of formulas. This is specified by the theory below, written $Equal(\Gamma, \{A_1, \dots, A_n\})$:

$$unitctx(A_1, \Gamma^1_{aux}), unitctx(A_2, \Gamma^2_{aux}), \dots, unitctx(A_n, \Gamma^n_{aux}), Unions(\Gamma, [\Gamma^1_{aux}, \dots, \Gamma^n_{aux}])$$

While $unitctx(A_i, \Gamma^i_{aux})$ specifies that the auxiliary context variable contains only the formula A_i, the specification $Unions(\Gamma, [\Gamma^1_{aux}, \dots, \Gamma^n_{aux}])$ specifies that $\Gamma = \Gamma^1_{aux} \cup \cdots \Gamma^n_{aux}$. Notice that the auxiliary variables are fresh, so the auxiliary variables needed in $Unions(\Gamma, [\Gamma^1_{aux}, \dots, \Gamma^n_{aux}])$ are different from those in $\{\Gamma^1_{aux}, \dots, \Gamma^n_{aux}\}$.

Finally, another specification that we will need is when two context variables have the same formulas, that is, $\Gamma = \Gamma'$. This is specified by the theory $EqualCtx(\Gamma, \Gamma')$, specified below:

$$emp(\Gamma_{aux}), union(\Gamma, \Gamma_{aux}, \Gamma')$$

The atom $emp(\Gamma_{aux})$ specifies that $\Gamma_{aux} = \emptyset$, while $union(\Gamma, \Gamma_{aux}, \Gamma')$ specifies that $\Gamma = \Gamma_{aux} \cup \Gamma'$.

We are aware that these operations might not cover all possible context transformations of sequent calculus rules, but, as we discuss in more details in Section 5, we use a linear logical framework for specifying proof systems (Nigam et al. 2011). In this language, the theory in Table 1 contains the necessary constructs to check the proof transformations proposed.

Derivation Skeletons Derivation skeletons are defined inductively by using introduction rules of a given proof system. Assume that the sequent contexts of this proof system are of the form

$$\Gamma_1 \mid \cdots \mid \Gamma_n \vdash \Gamma_{n+1} \mid \cdots \mid \Gamma_{n+m}$$

that is, with $n + m$ contexts. In Definition 2, $\text{ctx}_R(i, p)$ denotes the i^{th} context of premise p, if $p \neq 0$ or of the conclusion if $p = 0$.

Definition 2 (*Inference skeleton*)
Let R be an inference rule in the proof system. The *inference skeleton* for R is a pair $\langle \Xi, \mathcal{B} \rangle$, where Ξ is the inference rule with conclusion $\Gamma_{0,1} \mid \cdots \mid \Gamma_{0,n} \vdash \Gamma_{0,n+1} \mid \cdots \mid \Gamma_{0,n+m}$, and k premises $\Gamma_{j,1} \mid \cdots \mid \Gamma_{j,n} \vdash \Gamma_{j,(n+1)} \mid \cdots \mid \Gamma_{j,(n+m)}$, with $1 \leq j \leq k$, where all context variables have different names, *i.e.*, $\Gamma_{i,k} = \Gamma_{j,l}$ only if $i = j$ and $k = l$. $\mathcal{B}$ is a set of formulas constructed by checking how the contexts 1 to $n + m$ are represented in the conclusion and premises of R. $\mathcal{B}$ is the smallest set such that

- (No Context nor Formula) If $\text{ctx}_R(i, p)$ has no context nor formula variable, then $emp(\Gamma_{i,p}) \in \mathcal{B}$;
- (Single Context) If $\text{ctx}_R(i, p)$ is a single context variable Γ, then $EqualCtx(\Gamma_{i,p}, \Gamma) \subseteq \mathcal{B}$;
- (Single Formula) If $\text{ctx}_R(i, p)$ is of the form A, then $unitctx(A, \Gamma_{i,p}) \in \mathcal{B}$;

• (More than one Context and/or Formula) If $\mathrm{ctx}_R(i, p)$ is of the form $\Gamma_1, \ldots, \Gamma_n, A_1, \ldots, A_m$, then $Equal(\Gamma_{aux}, \{A_1, \ldots, A_m\})$, $Unions(\Gamma_{i,p}, [\Gamma_1, \ldots, \Gamma_n, \Gamma_{aux}]) \subseteq \mathcal{B}$.
where the auxiliary context variable Γ_{aux} is a fresh context name.

Example: Consider the $\otimes_R$ rule shown to the left. The *inference skeleton* for it is the pair $\langle \Xi_\otimes, \mathcal{B}_\otimes \rangle$ obtained as described in Definition 2, where $\Xi_\otimes$ is the derivation shown to the right:

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma' \vdash \Delta', B}{\Gamma, \Gamma' \vdash \Delta, \Delta', A \otimes B}\ [\otimes_R] \qquad \frac{\Gamma_{1,1} \vdash \Gamma_{1,2} \quad \Gamma_{2,1} \vdash \Gamma_{2,2}}{\Gamma_{0,1} \vdash \Gamma_{0,2}}$$

Moreover, $\mathcal{B}_\otimes$ is the set:

$$\mathcal{B}_\otimes = \left\{ \begin{array}{c} Equal(\Gamma^1_{aux}, \{A \otimes B\}), Unions(\Gamma_{0,2}, [\Delta, \Delta', \Gamma^1_{aux}]) \\ Equal(\Gamma^2_{aux}, \{A\}), Unions(\Gamma_{1,2}, [\Delta, \Gamma^2_{aux}]) \\ Equal(\Gamma^3_{aux}, \{B\}), Unions(\Gamma_{2,2}, [\Delta, \Gamma^3_{aux}]) \\ Unions(\Gamma_{0,1}, [\Gamma, \Gamma']), EqualCtx(\Gamma_{1,1}, \Gamma), EqualCtx(\Gamma_{2,1}, \Gamma') \end{array} \right\}$$

It is easy to check that the Logic Program (LP) $\mathcal{B}_\otimes \cup \mathcal{T}$ has a single answer set, containing the formulas $in(A \otimes B, \Gamma_{0,2})$, $in(A, \Gamma_{1,2})$ and $in(B, \Gamma_{2,2})$.

Now, consider the following case, where we know that Γ, Γ' has the formula C, but we do not specify to which premise it was moved. This can be specified by adding the formula $in(C, \Gamma_{0,1})$ to the set $\mathcal{B}_\otimes$. This addition will cause the resulting program to have two answer-sets. One where $in(C, \Gamma_{1,1})$, that is C is moved to the first premise, and another answer-set where $in(C, \Gamma_{2,1})$, that is, where C is moved to the second premise. Thus, we only need to build a generic derivation, Ξ, and specify declaratively the contents of its contexts.

The following definition specifies the set of derivation skeletons, which are obtained by using inference skeletons specified in Definition 2.

Definition 3 (*Derivation skeleton*)
The set of *derivation skeletons* is defined inductively. Every inference skeleton is a derivation skeleton. Let $\mathcal{D} = \langle \Xi_D, \mathcal{B}_D \rangle$ be a derivation skeleton with open premise P of the form $\Gamma_{p,1} \mid \cdots \mid \Gamma_{p,n} \vdash \Gamma_{p,n+1} \mid \cdots \mid \Gamma_{p,m+n}$. Let $\mathcal{I} = \langle \Xi_I, \mathcal{B}_I \rangle$ be an inference skeleton, introducing a sequent of the form $\Gamma_{I,1} \mid \cdots \mid \Gamma_{I,n} \vdash \Gamma_{I,n+1} \mid \cdots \mid \Gamma_{I,m+n}$, where we assume that all context variables in Ξ_I do not appear in Ξ_D. Then $\langle \Xi, \mathcal{B} \rangle$ is a derivation skeleton, where Ξ is obtained by replacing P by Ξ_I and where $\mathcal{B} = \mathcal{B}_D \cup \mathcal{B}_I \cup \mathcal{B}_{Join}$, where $\mathcal{B}_{Join}$ is the set of formulas:

$$EqualCtx(\Gamma_{p,1}, \Gamma_{I,1}), \ldots, EqualCtx(\Gamma_{p,n+m}, \Gamma_{I,n+m}).$$

The interesting bit is the set $\mathcal{B} = \mathcal{B}_D \cup \mathcal{B}_I \cup \mathcal{B}_{Join}$ specifying the contents of the contexts in Ξ. In particular, it contains the same specification of $\mathcal{B}_D$ and the specification of how formulas are moved in the inference rule ($\mathcal{B}_I$). The set $\mathcal{B}_{Join}$ specifies that the premise P of Ξ_D and the conclusion of Ξ_I are the same.

Example Continuing with the example above, consider the *derivation skeleton* obtained by applying another inference skeleton for the $\otimes_R$ rule, $\langle \Xi'_\otimes, \mathcal{B}'_\otimes \rangle$ to the left-premise of the inference skeleton built above, where $\langle \Xi'_\otimes, \mathcal{B}'_\otimes \rangle$ specifies the introduction of $C \otimes D$ on the right-hand-side and $in(C \otimes D, \Gamma_{0,2}) \in \mathcal{B}_\otimes$, that is, $C \otimes D$ appears in the conclusion sequent. Moreover, assume that $\langle \Xi'_\otimes, \mathcal{B}'_\otimes \rangle$ mentions the contexts names $\Gamma_{j,1}, \Gamma_{j,2}$ for $3 \leq j \leq 5$. We obtain the derivation skeleton $\langle \Xi, \mathcal{B} \rangle$, where Ξ is as follows:

$$\Xi \quad = \quad \frac{\dfrac{\Gamma_{4,1} \vdash \Gamma_{4,2} \quad \Gamma_{5,1} \vdash \Gamma_{5,2}}{\Gamma_{3,1} \vdash \Gamma_{3,2}} \qquad \Gamma_{2,1} \vdash \Gamma_{2,2}}{\Gamma_{0,1} \vdash \Gamma_{0,2}}$$

and $\mathcal{B} = \mathcal{B}_\otimes \cup \mathcal{B}'_\otimes \cup \mathcal{B}_{Join}$. It is easy to check that the LP $\mathcal{B} \cup \mathcal{T}$ has two answer-sets. One answer-set has $in(A, \Gamma_{4,2})$, that is, the formula A is moved to the left-premise. The second answer set has $in(A, \Gamma_{5,2})$, that is, the formula A is moved to the right-premise. In this way, we can construct a single derivation skeleton, while the answer sets of the LP program associated to it specifies the concrete instance derivations, if there are any.

Recovering Derivations We use the following rewrite system to recover a derivation of the proof system from an answer-set $a \in \mathcal{A}$ of a derivation skeleton $\langle \Xi, \mathcal{B} \rangle$. The rewrite system has two phases:

Phase 1:	$unitctx(F, \Gamma) : \Gamma \rightarrow F$	$emp(\Gamma) : \quad \Gamma \rightarrow \cdot$	$union(\Gamma', \Gamma'', \Gamma) : \quad \Gamma \rightarrow \Gamma', \Gamma''$
Phase 2:		$in(F, \Gamma) : \quad \Gamma \rightarrow \Gamma, F$	

In the first phase, for every applicable constraint, emp, $unitctx$, and $union$ in a, and applicable context variable, Γ, we apply the corresponding rule *exactly once*. Then in the second phase, for every in constraint in a and applicable context variable, Γ, we apply the corresponding rule again exactly once. We can show that this rewrite system is *strongly confluent*.

Definition 4 (*Derivation Instance*)
Let $\langle \Xi, \mathcal{B} \rangle$ be a derivation skeleton and $\mathbb{P}_1 = \mathcal{B} \cup \mathcal{T}$. Let $\mathcal{A}$ be the answer set of $\mathbb{P}_1$. Then the derivation $\langle \Xi, a \rangle$ obtained by using the rewrite system above and an answer-set $a \in \mathcal{A}$ is a *derivation instance* of $\langle \Xi, \mathcal{B} \rangle$.

The following theorem is proved by induction on the height of derivations and by using the rewrite system above.

Theorem 5 (*Soundness and Completeness*)
Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be the answer set of the derivation skeleton $\langle \Xi, \mathcal{B} \rangle$ obtained by applying some inference skeletons for a given proof system $\mathcal{P}$. Then $\langle \Xi, a_i \rangle$ is a derivation instance iff it is a derivation that can be obtained by applying the corresponding inference rules of $\mathcal{P}$ in the same order and on the same premises.

4 Checking Provability Implication of Derivations

We now are interested in solving Obligation 2: given two derivation Ξ_1 and Ξ_2, are the premises of Ξ_2 provable given proofs of the premises of Ξ_1? This problem is undecidable in general, as provability is undecidable already for first-order logic. So there is no hope to build a general tool that can automatically can answer this question. However, many proof transformations, such as permutation lemmas, do not need very complex arguments, though still combinatorial. Consider the following permutation:

$$\dfrac{\dfrac{\overset{\Xi}{\Gamma, A, B \vdash \Delta, C}}{\Gamma, A \wedge B \vdash \Delta, C}\,[\wedge_l] \quad \overset{\Xi'}{\Gamma', C \vdash \Delta'}}{\Gamma, \Gamma', A \wedge B \vdash \Delta, \Delta'}\,[cut] \quad \rightsquigarrow \quad \dfrac{\dfrac{\overset{\Xi}{\Gamma, A, B \vdash \Delta, C} \quad \overset{\Xi'}{\Gamma', C \vdash \Delta'}}{\Gamma, \Gamma', A, B \vdash \Delta, \Delta'}\,[cut]}{\Gamma, \Gamma', A \wedge B \vdash \Delta, \Delta'}\,[\wedge_l]$$

The premises of the derivation to the left (Ξ_1) are the same as the premises of the derivation to the right (Ξ_2). Therefore, one can simply re-use the proofs Ξ and Ξ'. In fact, as we show in this section, one can perform these checks in an automated fashion. We call this problem the

Table 2. *Predicates used to reason whether $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$.*

Predicate	**Meaning**
$inSequent(\Gamma, \mathcal{S})$	A context variable Γ is in a sequent called $\mathcal{S}$.
$inDer(\mathcal{S}, D)$	The sequent $\mathcal{S}$ belongs to the derivation D, where $D \in \{\Xi_1, \Xi_2\}$.
$bounded(\Gamma)$	Context Γ is bounded.
$proveIf(\mathcal{S}_2, \mathcal{S}_1)$	$\mathcal{S}_2$ is derivable if $\mathcal{S}_1$ is derivable.
$notProveIf(\mathcal{S}_2, \mathcal{S}_1)$	It is not possible to affirm that there is a derivation of $\mathcal{S}_2$ from a derivation of $\mathcal{S}_1$.

Table 3. *Theory $\mathcal{T}_d$ used to reason whether the open leaves of a derivation Ξ_2 are provable given the proofs of the open leaves of a derivation Ξ_1. Here we consider that $s_1, \ldots, s_n$ are the open leaves of Ξ_2.*

(c1)	$notProveIf(\mathcal{S}_2, \mathcal{S}_1) \leftarrow in(F, \Gamma), inSequent(\Gamma, \mathcal{S}_1), inDer(\mathcal{S}_1, \Xi_1),$ $not\ in(F, \Gamma'), inSequent(\Gamma', \mathcal{S}_2), inDer(\mathcal{S}_2, \Xi_2).$
(c2)	$notProveIf(\mathcal{S}_2, \mathcal{S}_1) \leftarrow in(F, \Gamma), inSequent(\Gamma, \mathcal{S}_2), inDer(\mathcal{S}_2, \Xi_2), bounded(\Gamma),$ $not\ in(F, \Gamma'), inSequent(\Gamma', \mathcal{S}_1), inDer(\mathcal{S}_1, \Xi_1).$
(c3)	$proveIf(\mathcal{S}_2, \mathcal{S}_1) \leftarrow not\ notProveIf(\mathcal{S}_2, \mathcal{S}_1), inDer(\mathcal{S}_1, \Xi_1), inDer(\mathcal{S}_2, \Xi_2).$
(c4)	$ok \leftarrow proveIf(s_1, \_), \ldots, proveIf(s_{n,})\,.$
(c5)	$\bot \leftarrow not\ ok.$

Provability Implication: does the proof of a sequent $\mathcal{S}_1$ imply the proof of another sequent $\mathcal{S}_2$? Although still undecidable, this problem is simpler as it allows us to consider the premises of the derivations individually, as done above. Henceforth we denote this problem by $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$.

We show that we can use ASP to solve automatically the Provability Implication problem for a wide range of proof systems. For this, we add to the language the predicates in Table 2 and the rules in Table 3.

The predicate $notProveIf(\mathcal{S}_2, \mathcal{S}_1)$ specifies the cases in which we *cannot* guarantee that $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$. They are: (1) $\mathcal{S}_1$ contains a formula that is not in $\mathcal{S}_2$. If this formula is used in the proof of $\mathcal{S}_1$, we cannot transform it into a proof of $\mathcal{S}_2$. This is specified by rule (c1). And (2) $\mathcal{S}_2$ contains a formula in a *bounded* context (see Section 2.1) that is not in $\mathcal{S}_1$ This is specified by rule (c2). This rule increases the power of our method, by using Lemma 1: if the sequent $\mathcal{S}_1$ is provable and if $\mathcal{S}_2$ is obtained by adding a formula to an unbounded context, then $\mathcal{S}_2$ is provable.

Notice that we are being overcautious, as there are cases in which conditions (c1) and/or (c2) are satisfied, and $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$. However, but there are no conditions for which (c1) and (c2) are not satisfied and it is not the case that $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$. Thus it is sound, but not complete.

Given that there is a way to identify when it is not possible to guarantee whether $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$, we use "double negation" to specify when it is the case that $\mathcal{S}_2$ is actually provable (predicate $proveIf(\mathcal{S}_2, \mathcal{S}_1)$). This is specified by rule (c3), which decides whether a sequent $\mathcal{S}_2$ from derivation Ξ_2 is provable given that another sequent $\mathcal{S}_1$ from derivation Ξ_1 is provable.

From rules (c1), (c2), and (c3), we can check for many cases whether $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$. In order to prove Obligation 2, we need to check if all the premises of Ξ_2 are provable from proofs of the premises of Ξ_1. Since these premises are sequents themselves, we can use the $proveIf(\mathcal{S}_2, \mathcal{S}_1)$ predicate to reason about all the of them. In particular, we need to make sure that *every* premise of Ξ_2 is proved from *some* premise of Ξ_1. This is specified by clause (c4).

Finally, we are interested only in whether there is a transformation or not, so we add the rule (c5) to make sure that no models will be generated if some premise of Ξ_2 does not follow from any premise of Ξ_1.

Let $\langle \Xi_1, a_1 \rangle$ and $\langle \Xi_2, a_2 \rangle$ be derivation instances. Furthermore, let $\mathcal{L}_i$ be the set of predicates $inSequent(\Gamma, \mathcal{S})$, $inDer(\mathcal{S}, D)$ and $bounded(\Gamma)$ that describes the open leaves of Ξ_i. We define $\mathbb{P}_2 = \mathcal{T}_d \cup a_1 \cup a_2 \cup \mathcal{L}_1 \cup \mathcal{L}_2$. It is required in $\mathbb{P}_2$ that each derivation, sequent and context have unique names.

The following theorem is proved by constructing the proof of the open premises of $\mathcal{D}_2$ from the proofs introducing the open-premises of $\mathcal{D}_1$ by weakening possible extra formulas in the open premises of $\mathcal{D}_2$, as specified by any answer-set of $\mathbb{P}_2$.

Theorem 6 (*Soundness*)
Let $\mathcal{D}_1 = \langle \Xi_1, a_1 \rangle$ and $\mathcal{D}_2 = \langle \Xi_2, a_2 \rangle$ be derivation instances. If $\mathbb{P}_2$ returns a non-empty answer set, then all open leaves of $\mathcal{D}_2$ are provable given proofs of open leaves of $\mathcal{D}_1$.

5 Putting all Together: Tool and Experimental Results

In Section 3, $\mathbb{P}_1$ was defined to obtain the possible derivations of a sequence of rule applications of a proof system. In Section 4, $\mathbb{P}_2$ was presented to determine if a derivation instance $\mathcal{D}_2$ follows from another derivation instance $\mathcal{D}_1$. Using these two programs, it is possible to automate the checking of a certain type of proof transformations. In this Section we define the problem of rule permutation and how this is solved combining $\mathbb{P}_1$ and $\mathbb{P}_2$.

Definition 7 (*Rule Permutation*)
Let $\mathcal{S}$ be a sequent and α and β two inference rules of some proof system $\mathcal{P}$. Let $\mathcal{D}_1$ be the set of derivations obtained by applying α and then β (bottom up) to $\mathcal{S}$, and $\mathcal{D}_2$ the set of derivations obtained by applying β and then α. We say that *α permutes over β* if for all $d_1 \in \mathcal{D}_1$, there exists $d_2 \in \mathcal{D}_2$ such that the provability of d_1 implies the provability of d_2.

The input of our algorithm to check permutations is: a proof system $\mathcal{P}$ (inference rules), the format of a sequent in $\mathcal{P}$ (bounded and unbounded, left and right contexts), two inference rules α and β and the description of an initial sequent, i.e., the principal formulas to which α and β are applied and where these formulas are in the sequent (described by the predicate $in(F, \Gamma)$).

Using $\mathbb{P}_1$ and the initial constraints, one can generate the *derivation instances* $\mathcal{D}_1$ from the application of rules α/β. Similarly, the derivation instances $\mathcal{D}_2$ are obtained from the application of β/α. Since $\mathbb{P}_1$ is sound and complete, the sets in $\mathcal{D}_1$ and $\mathcal{D}_2$ correspond to all possible derivations of α/β and β/α respectively.

To check if α permutes over β, we need to check that all $d_1 \in \mathcal{D}_1$ have a corresponding $d_2 \in \mathcal{D}_2$ such that the provability of d_1 implies the provability of d_2. This provability check is done by $\mathbb{P}_2$. By executing the program a finite number of times, since the sets $\mathcal{D}_1$ and $\mathcal{D}_2$ are finite, one can check the permutation condition. Since $\mathbb{P}_2$ is sound, it will only fail when indeed d_2 is not provable from d_1.

Tool and Experimental Results In (Miller and Pimentel 2013) it was shown how linear logic can be successfully used as a framework for the specification of sequent calculi. Later, in (Nigam et al. 2011) it was shown how linear logic with subexponentials can capture a wider range of proof systems with rules that have more refined structural restrictions. Using the latter logic it is possible to specify the well known sequent calculi LK and LJ, but also more involved calculi such as S4 and G3K (Negri 2005) for modal logics and G_K (Avron et al. 2013) for paraconsistent logics. This framework, and the specification of several calculi, can be found at `http://code.google.com/p/sellf`. This is part of a bigger project on reasoning about sequent calculus systems, and the advantage of using this framework is that we can check proof transformations for different calculi that are available.

For this reason we chose to implement the methods of Sections 3 and 4 for the focused sequent calculus for linear logic. This fact has also influenced the decision for choosing the alphabet in Table 1. Using these methods, we implemented a function that checks permutation lemmas and made it available for the users. Given the specification of a logic in this framework, the user can choose two inference rules, say, α and β, and the system automatically checks if α permutes over β. Some examples of specifications are already available in the system, but the user is free to write their own.

Currently, the system outputs only whether the rules permute or not (the negative answer corresponding to "don't know"), but in principle it could also show the permutation cases. We expect to add this functionality until before the conference.

We tested the implementation using the rules for linear logic and intuitionistic logic, in which the permutation cases are well known and essential for the completeness of the proof search disciplines focusing and uniform proofs, respectively. Our system identified correctly all the permutation cases for LL: $\parr/\parr$, $\parr/\&$, $\&/\parr$, $\&/\&$, $\otimes/\otimes$, $\otimes/\oplus$, $\oplus/\otimes$, $\oplus/\oplus$, $\oplus/\parr$, $\otimes/\parr$, $\oplus/\&$, $\otimes/\&$; and the all the permutations for LJ: $\supset_l$ / $\supset_r$, $\supset_l$ /$\wedge_r$, $\supset_l$ /$\wedge_l$.

6 Related and Future Work

This paper contributes to automating the check of important proof theoretic properties. In particular, we showed how ASP is a suitable programming paradigm for checking proof transformations. We proposed two programs for doing so. The first checks the validity of a derivation given where the formulas can possibly occur in a sequent. The second program is used to check whether the proof of one sequent follows from the proof of another sequent. This is part of an effort to build tools to help proof theorists to design proof systems.

The problem of checking the validity of proof transformations and in particular, for the proof of permutation lemmas is as old as sequent calculus systems. Kleene in (Kleene 1952) already investigated this problem for LK and LJ. Some have also investigated more systematically how to determine whether a permutation is valid. For instance, (Galmiche and Perrier 1994) and more recently (Lutovac and Harland 2013) have proposed some vocabulary based on the role of formulas in an instance of an inference rule. They were able to prove when a permutation is possible by using this vocabulary. However, they do not provide the means to automatize this check as one still needs to enumerate all possible instances of rule applications to check whether all instances can permute. This paper gives a solution for this problem by using ASP. We believe that the these two lines of work complement each other and could be combined. But this is left as future work. In fact, up to the best of our knowledge, this is the first result on automating the check of proof transformations.

Similarly, in machine proofs for cut-elimination, such as those in Twelf (Schürmann 2000) or Abella (Gacek 2009), one proves permutation lemmas. Again, one needs to prove each case by hand. Although the system will complain when one case is forgotten, still the permutation lemmas compose roughly half of the proof. One future direction is to integrate our method, which only provides yes/no answers, with these systems. This would mean that our system would also output a proof object, which could be checked by Twelf or Abella. Moreover, it would also remove the burden from the user to write down many (if not all) permutation lemmas by hand.

Finally, since the completeness proof of the focusing strategy heavily relies on permutation lemmas (Miller and Saurin 2007), we are currently investigating whether one can automate the proposal of focused proof systems which are complete to their un-focused version.

Acknowledgements We thank Dale Miller, Elaine Pimentel, and Bruno W. Paleo for fruitful discussions. Nigam was supported by the CNPq.

References

Andreoli, J.-M. 1992. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation 2,* 3, 297–347.

Avron, A., Konikowska, B., and Zamansky, A. 2013. Cut-free sequent calculi for c-systems with generalized finite-valued semantics. *J. Log. Comput. 23,* 3, 517–540.

Cervesato, I. and Pfenning, F. 2002. A Linear Logical Framework. *Information & Computation 179,* 1 (Nov.), 19–75.

Crolard, T. 2001. Subtractive logic. *Theor. Comput. Sci. 254,* 1-2, 151–185.

Gacek, A. 2009. The Abella system and homepage. `http://abella.cs.umn.edu/`.

Galmiche, D. and Perrier, G. 1994. On proof normalization in linear logic. *Theoretical Computer Science 135*, 135–1.

Gelfond, M. and Lifschitz, V. 1990. Logic programs with classical negation. In *ICLP*. 579–597.

Gentzen, G. 1969. Investigations into logical deductions. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed. North-Holland, Amsterdam, 68–131.

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

Kleene, S. C. 1952. Permutabilities of inferences in Gentzen's calculi LK and LJ. *Memoirs of the American Mathematical Society 10*, 1–26.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic 7*, 499–562.

Lutovac, T. and Harland, J. 2013. A contribution to automated-oriented reasoning about permutability of sequent calculi rules. Submitted to Computer Science and Information Systems.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51*, 125–157.

Miller, D. and Pimentel, E. 2013. A formal framework for specifying sequent calculus proof systems. TCS, 474: 98–116.

Miller, D. and Saurin, A. 2007. From proofs to focused proofs: a modular proof of focalization in linear logic. In CSL, 2007.

Negri, S. 2005. Proof analysis in modal logic. *J. Philosophical Logic 34,* 5-6, 507–544.

Niemelä, I. and Simons, P. 1997. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *LPNMR*. 421–430.

Nigam, V. and Miller, D. 2010. A framework for proof systems. *J. Autom. Reasoning 45,* 2, 157–188.

Nigam, V., Pimentel, E., and Reis, G. 2011. Specifying proof systems in linear logic with subexponentials. *Electr. Notes Theor. Comput. Sci. 269*, 109–123.

Pfenning, F. and Simmons, R. J. 2009. Substructural operational semantics as ordered logic programming. In LICS, 2009.

Rauszer, C. 1974. A formalization of the propositional calculus h-b logic. *Studia Logica 33*, 23–34.

Schürmann, C. 2000. Automating the meta theory of deductive systems. Ph.D. thesis, Carnegie Mellon University. CMU-CS-00-146.