

# 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 (Gen69) 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 (MNPS91) and the focusing discipline (And92), 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 (MS07; Gen69). 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 (Gir87) 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:

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

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  $\Xi_1$  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* (Gen69).<sup>1</sup>

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 (Rau74) given by Rauszer was later found to be incorrect (Cro01) 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) (GL90) 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 (NS97; LPF<sup>+</sup>06). 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 (And92) and uniform proofs (MNPS91). 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 on how these two programs can be used to check whether a rule permutes over another and

<sup>1</sup> 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](http://twelf.org/wiki/Admissibility_of_cut).

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 introduce the terminology which will be used to guide our work. In general, a *sequent with contexts* (And92) is a slight generalization used in the context of logic programming and logical frameworks (NM10; CP02) of the usual sequents in sequent calculus (Gen69). 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 (Gen69)) 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.,  $\Gamma, \Delta, \Theta$ , 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:

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

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 (NM10; CP02), 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. 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  $\Gamma$  of LJ's  $\wedge_R$  rule.

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  $\Gamma_i$  is an unbounded context. Let  $\Gamma'_i \supseteq \Gamma_i$  be a superset of  $\Gamma_i$  for every  $i$ . If the sequent  $\mathcal{S}$  is provable in  $\mathcal{P}$ , then the sequent  $\mathcal{S}'$  obtained from  $\mathcal{S}$  by replacing  $\Gamma_i$  by  $\Gamma'_i$  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 (PS09). 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 (GL90), 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 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

$$a \leftarrow not\ b \quad b \leftarrow not\ a \quad c \leftarrow a$$

has two answer-sets  $\{a, c\}$  and  $\{b\}$ . Finally, one can also specify a constraint in ASP by using a rule whose head is the falsity, denoted by the symbol  $\perp$ . For example, the rule specifies the constraint that  $b$  cannot be true:

$$\perp \leftarrow b$$

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

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) $\perp \leftarrow in(F_1, \Gamma), unitctx(F, \Gamma), F_1 \neq F$ .
$emp(\Gamma)$	$\Gamma = \emptyset$	(r3) $\perp \leftarrow in(F, \Gamma), emp(\Gamma)$ .
$union(\Gamma^1, \Gamma^2, \Gamma)$	$\Gamma = \Gamma^1 \cup \Gamma^2$	(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) $in(F, \Gamma^2) \leftarrow not\ in(F, \Gamma^1), in(F, \Gamma), union(\Gamma^1, \Gamma^2, \Gamma)$ .

### 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 as usual.

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,  $\Gamma$ , 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  $\Gamma$ .

The atomic formula  $unitctx(F, \Gamma)$  specifies that the context  $\Gamma$  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  $\Gamma$ .

In some situations, for instance, when specifying the linear logic initial rule (Gir87), 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<sup>2</sup> of a formula that is in  $\Gamma^i$  is also in  $\Gamma$ . The rule (r6) specifies that if both  $\Gamma^1$  and  $\Gamma^2$  are empty then so is  $\Gamma$ . 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  $\Gamma$  either comes from  $\Gamma^1$  or from  $\Gamma^2$ . Notice how we use the default negation to generate accordingly the splitting behaviour of bounded contexts.

*Auxiliary Context Variables and Macros* From these rules, we can specify some more elaborate conditions on context variables and formula variables. For this we may need fresh auxiliary

<sup>2</sup> Recall that as discussed in Section 2, we keep track of the occurrences of formulas.

context variables, written  $\Gamma_{aux}^j$  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  $\Gamma = \Gamma_1 \cup \dots \cup \Gamma_n$  by using auxiliary context variables as follows:

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

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  $Equal(\Gamma, \{A_1, \dots, A_n\})$ :

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

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

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 (NPR11). 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

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

that is, with  $n + m$  contexts.

In Definition 2,  $ctx_R(i, p)$  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  $\Xi$  is the inference rule with conclusion  $\Gamma_{0,1} \mid \dots \mid \Gamma_{0,n} \vdash \Gamma_{0,n+1} \mid \dots \mid \Gamma_{0,n+m}$ , and  $k$  premises  $\Gamma_{j,1} \mid \dots \mid \Gamma_{j,n} \vdash \Gamma_{j,(n+1)} \mid \dots \mid \Gamma_{j,(n+m)}$ , 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  $ctx_R(i, p)$  has no context nor formula variable, then  $emp(\Gamma_{i,p}) \in \mathcal{B}$ ;
- (Single Context) If  $ctx_R(i, p)$  is a single context variable  $\Gamma$ , then  $EqualCtx(\Gamma_{i,p}, \Gamma) \subseteq \mathcal{B}$ ;
- (Single Formula) If  $ctx_R(i, p)$  is of the form  $A$ , then  $unitctx(A, \Gamma_{i,p}) \in \mathcal{B}$ ;
- (More than one Context and/or Formula) If  $ctx_R(i, p)$  is of the form  $\Gamma_1, \dots, \Gamma_n, A_1, \dots, A_m$ , then  $Equal(\Gamma_{aux}, \{A_1, \dots, A_m\}), Unions(\Gamma_{i,p}, [\Gamma_1, \dots, \Gamma_n, \Gamma_{aux}]) \subseteq \mathcal{B}$ .

where the auxiliary context variable  $\Gamma_{aux}$  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:

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

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

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

It is easy to check that the 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  $\Gamma, \Gamma'$  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,  $\Xi$ , 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  $\mathcal{D} = \langle \Xi_{\mathcal{D}}, \mathcal{B}_{\mathcal{D}} \rangle$  be a derivation skeleton with open premise  $P$  of the form  $\Gamma_{p,1} \mid \dots \mid \Gamma_{p,n} \vdash \Gamma_{p,n+1} \mid \dots \mid \Gamma_{p,m+n}$ . Let  $\mathcal{I} = \langle \Xi_{\mathcal{I}}, \mathcal{B}_{\mathcal{I}} \rangle$  be an inference skeleton, introducing a sequent of the form  $\Gamma_{l,1} \mid \dots \mid \Gamma_{l,n} \vdash \Gamma_{l,n+1} \mid \dots \mid \Gamma_{l,m+n}$ , where we assume that all context variables in  $\Xi_{\mathcal{I}}$  do not appear in  $\Xi_{\mathcal{D}}$ . Then  $\langle \Xi, \mathcal{B} \rangle$  is a derivation skeleton, where  $\Xi$  is obtained by replacing  $P$  by  $\Xi_{\mathcal{I}}$  and where  $\mathcal{B} = \mathcal{B}_{\mathcal{D}} \cup \mathcal{B}_{\mathcal{I}} \cup \mathcal{B}_{Join}$ , where  $\mathcal{B}_{Join}$  is the set of formulas:

$$\text{EqualCtx}(\Gamma_{p,1}, \Gamma_{l,1}), \dots, \text{EqualCtx}(\Gamma_{p,n+m}, \Gamma_{l,n+m}).$$

The interesting bit is the set  $\mathcal{B} = \mathcal{B}_{\mathcal{D}} \cup \mathcal{B}_{\mathcal{I}} \cup \mathcal{B}_{Join}$  specifying the contents of the contexts in  $\Xi$ . In particular, it contains the same specification of  $\mathcal{B}_{\mathcal{D}}$  and the specification of how formulas are moved in the inference rule ( $\mathcal{B}_{\mathcal{I}}$ ). The set  $\mathcal{B}_{Join}$  specifies that the premise  $P$  of  $\Xi_{\mathcal{D}}$  and the conclusion of  $\Xi_{\mathcal{I}}$  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. Assume that  $\langle \Xi'_{\otimes}, \mathcal{B}'_{\otimes} \rangle$  specifies the introduction of the formula  $C \otimes D$  on the right-hand-side and mentions the contexts names  $\Gamma_{j,1}, \Gamma_{j,2}$  for  $3 \leq j \leq 5$ . We obtain a derivation skeleton  $\langle \Xi, \mathcal{B} \rangle$ .  $\Xi$  is as follows:

$$\frac{\frac{\Gamma_{4,1} \vdash \Gamma_{4,2} \quad \Gamma_{5,1} \vdash \Gamma_{5,2}}{\Gamma_{3,1} \vdash \Gamma_{3,2}} \quad \Gamma_{2,1} \vdash \Gamma_{2,2}}{\Gamma_{0,1} \vdash \Gamma_{0,2}}$$

Moreover,  $\mathcal{B} = \mathcal{B}_{\otimes} \cup \mathcal{B}'_{\otimes} \cup \mathcal{B}_{Join}$ . 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.

*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 an answer-set  $a \in \mathcal{A}$  is a *derivation instance* of  $\langle \Xi, \mathcal{B} \rangle$ .

*Theorem 5 (Soundness and Completeness)*

Let  $\mathcal{A} = \{a_1, \dots, a_n\}$  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  $\Xi_1$  and  $\Xi_2$ , are the premises of  $\Xi_2$  provable given proofs of the premises of  $\Xi_1$ ? This amounts to check whether for every open leaf  $\mathcal{S}_2$  of  $\Xi_2$  there exists an open leaf  $\mathcal{S}_1$  such that the provability of  $\mathcal{S}_1$  implies the provability of  $\mathcal{S}_2$ . So this problem can be reduced to what we call the *Provability Implication* problem, namely: does the proof of a sequent  $\mathcal{S}_1$  imply the proof of another sequent  $\mathcal{S}_2$ ? Henceforth we denote this problem by  $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ .

Notice that this problem is in general undecidable. Consider for instance the proof system LK for first-order classical logic. The provability problem will amount to proving a first-order logic (FOL) formula, namely to prove that the formula denoted by  $\mathcal{S}_1$  implies the formula denoted by  $\mathcal{S}_2$ . Since FOL provability is undecidable in general, the provability implication is also undecidable in general. Therefore, there is no hope for a method that is complete if we wish to make it general enough to work for a broad set of logics.

However, many proof transformations, such as permutation lemmas, do not need very complex arguments. Consider the following permutation:

$$\frac{\frac{\frac{\Xi}{\Gamma, A, B \vdash \Delta, C}}{\Gamma, A \wedge B \vdash \Delta, C} [\wedge_I] \quad \frac{\Xi'}{\Gamma', C \vdash \Delta'}}{\Gamma, \Gamma', A \wedge B \vdash \Delta, \Delta'} [cut] \quad \rightsquigarrow \quad \frac{\frac{\frac{\Xi}{\Gamma, A, B \vdash \Delta, C} \quad \frac{\Xi'}{\Gamma', C \vdash \Delta'}}{\Gamma, \Gamma', A, B \vdash \Delta, \Delta'} [cut]}{\Gamma, \Gamma', A \wedge B \vdash \Delta, \Delta'} [\wedge_I]}$$

The premises of the derivation to the left ( $\Xi_1$ ) are the same as the premises of the derivation to the right ( $\Xi_2$ ). Therefore, one can simply re-use the proofs  $\Xi$  and  $\Xi'$ . In fact, as we show in this section, one can perform these checks in an automated fashion. This is again done by means of ASP, and for that 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 a bit 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.

Note that these conditions are sufficient but not necessary. There might be cases that  $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$  holds, but our method cannot identify it. Thus it is sound, but not complete.

Given that there is a way to identify when it is not possible to guarantee whether  $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$ ,



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

Predicate	Meaning
$inSequent(\Gamma, \mathcal{S})$	A context variable $\Gamma$ 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 $\Gamma$ 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  $\Xi_2$  are provable given the proofs of the open leaves of a derivation  $\Xi_1$ . Here we consider that  $s_1, \dots, s_n$  are the open leaves of  $\Xi_2$ .*

(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)	$ok \leftarrow proveIf(s_1, \cdot), \dots, proveIf(s_n, \cdot).$
(c5)	$\perp \leftarrow not ok.$

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  $\Xi_2$  is provable given that another sequent  $\mathcal{S}_1$  from derivation  $\Xi_1$  is provable.

From rules (c1), (c2), and (c3), we can decide  $\mathcal{S}_1 \Rightarrow \mathcal{S}_2$  for some cases. In order to prove Obligation 2, we need to check if all the premises of  $\Xi_2$  are provable from proofs of the premises of  $\Xi_1$ . 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  $\Xi_2$  is proved from *some* premise of  $\Xi_1$ . 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  $\Xi_2$  does not follow from any premise of  $\Xi_1$ .

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  $\Xi_i$ . 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.

#### Theorem 6 (Soundness)

Let  $\mathcal{D}_1 = \langle \Xi_1, a_2 \rangle$  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  $\mathcal{D}_1$ .

#### Proof sketch

The predicate  $notProveIf(\mathcal{S}_2, \mathcal{S}_1)$  is true if there is a possibility that a proof of  $\mathcal{S}_1$  cannot be transformed into a proof of  $\mathcal{S}_2$ . If it is false, then we are sure that a proof of  $\mathcal{S}_1$  can be transformed into a proof of  $\mathcal{S}_2$  (either because these sequents have the same formulas in the same contexts, or

because  $\mathcal{S}_2$  has extra formulas that can be weakened). This is encoded by the clause (c3), which also guarantees that  $\mathcal{S}_1$  is from derivation  $\Xi_1$  and  $\mathcal{S}_2$  is from derivation  $\Xi_2$ . We can conclude that if  $\text{proveIf}(\mathcal{S}_2, \mathcal{S}_1)$  is true, then it is certainly the case that a proof of  $\mathcal{S}_2$  follows from a proof of  $\mathcal{S}_1$ .

Given clauses (c4) and (c5), the program will only return a non-empty set if  $ok$  holds. According to clause (c4),  $ok$  will hold when  $\text{proveIf}(s_i, \_)$  is true for every open leaf  $s_i$  of  $\Xi_2$ , which means that proofs of  $s_i$  will follow from proofs of open leaves of  $\Xi_1$ .  $\square$

## 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  $\alpha$  and  $\beta$  two inference rules of some proof system  $\mathcal{P}$ . Let  $\mathcal{D}_1$  be the set of derivations obtained by applying  $\alpha$  and then  $\beta$  (bottom up) to  $\mathcal{S}$ , and  $\mathcal{D}_2$  the set of derivations obtained by applying  $\beta$  and then  $\alpha$ . We say that  $\alpha$  *permutes over*  $\beta$  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  $\alpha$  and  $\beta$  and the description of an initial sequent, i.e., the principal formulas to which  $\alpha$  and  $\beta$  are applied and where these formulas are in the sequent (described by the predicate  $\text{in}(F, \Gamma)$ ).

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

To check if  $\alpha$  permutes over  $\beta$ , 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 (MP13) it was shown how linear logic can be successfully used as a framework for the specification of sequent calculi. Later, in (NPR11) 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 (Neg05) for modal logics and  $G_K$  (AKZ13) 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,  $\alpha$  and  $\beta$ , and the system automatically checks if  $\alpha$  permutes over  $\beta$ . 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:  $\wp/\wp$ ,  $\wp/\&$ ,  $\&/\wp$ ,  $\&/\&$ ,  $\otimes/\otimes$ ,  $\otimes/\oplus$ ,  $\oplus/\otimes$ ,  $\oplus/\oplus$ ,  $\oplus/\wp$ ,  $\otimes/\wp$ ,  $\oplus/\&$ ,  $\otimes/\&$ ; 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 (Kle52) already investigated this problem for LK and LJ. Some have also investigated more systematically how to determine whether a permutation is valid. For instance, (GP94) and more recently (LH13) 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.

There are several directions to follow from here. Currently, our system provides a yes/no answer (and until the conference, it will also draw the transformations). It would be desirable to output a machine checkable proof object. We are currently investigating how to output, in particular, a proof assistant code demonstrating some permutation, that could be checked by a proof assistant, for example, Twelf (Sch00) or Abella (Gac09).

Since the completeness proof of the focusing strategy heavily relies on permutation lemmas (MS07), we are currently investigating whether one can automate the proposal of focused proof systems which are complete to their un-focused version.

## References

- Arnon Avron, Beata Konikowska, and Anna Zamansky. Cut-free sequent calculi for c-systems with generalized finite-valued semantics. *J. Log. Comput.*, 23(3):517–540, 2013.

- Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
- Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information & Computation*, 179(1):19–75, November 2002.
- Tristan Crolard. Subtractive logic. *Theor. Comput. Sci.*, 254(1-2):151–185, 2001.
- Andrew Gacek. The Abella system and homepage. <http://abella.cs.umn.edu/>, 2009.
- Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.
- Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In *ICLP*, pages 579–597, 1990.
- Didier Galmiche and Guy Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135:135–1, 1994.
- Stephen Cole Kleene. Permutabilities of inferences in Gentzen’s calculi LK and LJ. *Memoirs of the American Mathematical Society*, 10:1–26, 1952.
- Tatjana Lutovac and James Harland. A contribution to automated-oriented reasoning about permutability of sequent calculi rules. Submitted to *Computer Science and Information Systems*, 2013.
- Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7:499–562, July 2006.
- Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
- Dale Miller and Elaine Pimentel. A formal framework for specifying sequent calculus proof systems. To appear in *TCS.*, 2013.
- Dale Miller and Alexis Saurin. From proofs to focused proofs: a modular proof of focalization in linear logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 405–419. Springer, 2007.
- Sara Negri. Proof analysis in modal logic. *J. Philosophical Logic*, 34(5-6):507–544, 2005.
- Vivek Nigam and Dale Miller. A framework for proof systems. *J. Autom. Reasoning*, 45(2):157–188, 2010.
- Vivek Nigam, Elaine Pimentel, and Giselle Reis. Specifying proof systems in linear logic with subexponentials. *Electr. Notes Theor. Comput. Sci.*, 269:109–123, 2011.
- Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *LPNMR*, pages 421–430, 1997.
- Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 101–110. IEEE Computer Society, 2009.
- C. Rauszer. A formalization of the propositional calculus h-b logic. *Studia Logica*, 33:23–34, 1974.
- Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, October 2000. CMU-CS-00-146.