Formalized meta-theory of sequent calculi for linear logics

Kaustuv Chaudhuri, Leonardo Lima, Giselle Reis

Formalized Meta-Theory of Sequent Calculi for Linear Logics

Kaustuv Chaudhuri

Inria & LIX/École polytechnique, France

Leonardo Lima

Technische Universität Dresden

Giselle Reis

Carnegie Mellon University in Qatar

Abstract

When studying sequent calculi, proof theorists often have to prove properties about the systems, whether to show that they are "well-behaved", amenable to automated proof search, complete with respect to another system, consistent, among other reasons. These proofs usually involve many very similar cases, which leads to authors rarely writing them in full detail, only pointing to one or two more complicated cases. Moreover, the amount of details makes them more error-prone for humans. Computers, on the other hand, are very good at handling details and repetitiveness.

In this work we have formalized textbook proofs of the meta-theory of sequent calculi for linear logic in Abella. Using the infrastructure developed, the proofs can be easily adapted to other substructural logics. We implemented rules as clauses in an intuitive and straightforward way, similar to logic programming, using operations on multisets for the explicit contexts. Although the proofs are quite big, they use only elementary reasoning principles, which makes the proof techniques fairly portable to other formal reasoning systems.

Keywords: Linear logic; sequent calculus; mechanized meta-theory; logic programming; proof theory

1. Introduction

Sequent calculus proof systems are perhaps the most standard technique used to formulate logics. New logics are nearly always proposed in terms of a

Email addresses: kaustuv.chaudhuri@inria.fr (Kaustuv Chaudhuri), leonardo.alfs@gmail.com (Leonardo Lima), giselle@cmu.edu (Giselle Reis)

sequent calculus. Such proposals are usually also accompanied by certain meta-theorems about the calculi. *Cut-elimination* is usually one of the first things to be established, as it usually entails the system's consistency and makes it suitable for automated proof search. Other meta-theorems include identity reduction, which shows internal completeness of the proof system; rule permutations and inversion lemmas to establish the polarities of connectives; and focusing theorems that establish the existence of normal forms. Although this meta-theory is very important, the proofs are rarely spelled out in detail, let alone formally checked by a proof assistant. One big reason is that these proofs involve a number of cases which is sometimes quadratic in the number of rules in the system, with many of them being very similar. A common approach in publications is to show one or two characteristic cases in detail and then to mention that the rest is "analogous" or "trivial".

Such informal proofs are risky. Girard himself underestimated the difficulty of cut-elimination in linear logic with exponentials. The terminating proof needed a much more involved inductive measure and was detailed later on in [1]. A proof of cut-elimination for full intuitionistic linear logic (FILL) was shown to have a mistake in [2], and the authors of the proof have later published a full corrected version [3]. A proof of cut-elimination for the sequent calculus GLS_V for the provability logic GL was the source of much controversy until this was resolved in [4] and formalized in [5] using Isabelle/HOL. Several sequent calculi proposed for bi-intuitionistic logic were "proved" to enjoy cut-elimination when, in fact, they did not. The mistake is analyzed and fixed in [6]. More recently an error in the cut-elimination proof for modal logic nested systems was corrected in [7].

The repetitive and detail-intensive nature of these proofs makes them good candidates for computerization. However, it is rare to find such proofs formalized along the lines of their informal arguments. Linear logic, in particular, presents a significant challenge because of its treatment of contexts as multisets rather than sets or lists of formulas. Lemmas about multisets are often taken as standard (and invisible) background in an informal proof, but a formal development must explicitly reason about multisets, often inductively.

We have decided to put to test the "folk wisdom" that formal reasoning with multisets is tricky. We formalize the meta-theory for several sequent calculi for various fragments of linear logic. As far as we know, this is the first formalization of its kind. We follow the usual textbook inductive proofs on the rank of the cut formula and/or proof heights, rewriting cuts to smaller cuts. We have proved cut-admissibility, invertibility of inference rules and generalized identity for several systems. Our results show that, with a good encoding of multisets and their properties, the formalization of particular meta-theorems can be completed quickly. Moreover, our formalizations use only elementary theorem proving techniques that can be explained to and carried out by undergraduate students. All that is required is a basic knowledge of proof theory and logic programming.

We have used the proof assistant Abella for this task. Abella's two-level logic approach facilitates the treatment of binders and we avoid having to implement more complicated solutions such as parameterized higher-order abstract

syntax [8].

The implementation can be found online at the following locations.

Browse: `http://abella-prover.org/ll-meta/`
Source: `https://github.com/meta-logic/abella-reasoning`

In this paper we describe relevant parts of the formalization and proofs for the meta-theory of propositional and first-order multiplicative-additive linear logic, and propositional multiplicative-exponential linear logic. Moreover, we explain how the same ideas carry over to formalize the meta-theory of focused and two-sided systems. This work extends [9] by handling first-order and focused calculi. For the most recent developments, please refer to one of the websites above as they will be continuously updated.

2. Background

We use the interactive proof assistant Abella [10] to formalize our different meta-theoretic proofs. The logic behind Abella is a conservative extension of intuitionistic first-order logic; the extensions that are relevant for this paper are:

- The pure simply typed λ-calculus as the term language, together with a primitive equality predicate on such terms that implements $\alpha\beta\eta$-equivalence with all uninterpreted constants treated as *constructors*. The logic is built atop this simply typed term language following the design of Church's simple theory of types. There is a type `prop` dedicated to formulas of the logic, and all logical connectives are implemented as constants of target type `prop`; for example, conjunction $\wedge$ has type $\texttt{prop} \to \texttt{prop} \to \texttt{prop}$ (where $\to$ is the function type), although we write it inline as $A \wedge B$ instead of as $\wedge\ A\ B$.

- Least and greatest fixed point definitions for predicates, *i.e.*, constants of target type `prop`. Such definitions come equipped with corresponding induction and co-induction rules for reasoning about assumptions and conclusions, respectively, and may additionally be *unfolded* to replace any instance of the predicate by its corresponding body.

- Support for extensional universal ($\forall$) quantification. Extensional variables, called *eigenvariables*, are allowed to be *instantiated* by arbitrary terms when reasoning about equations or during case-analysis. This can be highlighted by the formula $\forall x.\,(x = c) \supset p(x) \supset p(c)$ (for some constant c), which is provable because analyzing[1] $x = c$ has the effect of instantiating x with c, reducing the proof obligation to $p(c) \supset p(c)$. However, $\forall x.\, x \neq c$ is not provable[2] – for instance, it would be false in a model where c is the only element. Abella also has existential quantification ($\exists$) and an intensional quantification (∇) that mediates between the two.

[1] More precisely, using the left-introduction rule for equality in the sequent calculus $\mathcal{G}$ [11, 12].
[2] We define $s \neq t$ to be the formula $(s = t) \supset \bot$.

A comprehensive introduction to Abella, including a discussion of its features that are not used in this paper, may be found in the tutorial [10]. The proof theory of $\mathcal{G}$, the logic underlying Abella, is described in [11, 12] and references therefrom.

2.1. Relational Reasoning in Abella

Unlike many other proof assistants in popular use—such as Coq, Agda, Isabelle, etc.—Abella follows a *relational* approach rather than a *functional* approach to specifications. The equational theory on terms in Abella cannot be extended by functional definitions, and hence the term language is just ordinary λ-terms built from variables, constants, λ-abstraction, and application. If we declare a type `nat` of natural numbers with two constants `z` : `nat` and `s` : `nat` → `nat`, then the definition of addition `plus` would be given in terms of a relation of type `nat` → `nat` → `nat` → `prop` that relates the first two arguments to their sum in the third argument. Concretely, this definition would be specified as follows:

```
Define plus : nat -> nat -> nat -> prop by
; plus z X X
; plus (s X) Y (s Z) := plus X Y Z.
```

This definition consists of two *definitional clauses* which are separated from each other by semi-colons, and each clause consists of a *head* and, optionally, a *body* separated by `:=`. Each clause is also implicitly universally (∀) closed over its capitalized identifiers. The first clause above declares that for any `X`, the atom `plus z X X` is `true` (an omitted body in a clause is taken to stand for `true`). The second clause says that for every `X`, `Y`, and `Z`, the atom `plus (s X) Y (s Z)` is true if `plus X Y Z` is true. This predicate is, moreover, given a least fixed point interpretation, which means that there are no other ways of deriving `plus s t u` (for any terms `s`, `t`, and `u`) besides using one of the two definitional clauses. Note that definitional clauses for a predicate do not need to have non-overlapping heads, nor is the body of a clause required to use only subterms of the terms at the head. Thus, iteratively unfolding a relation can be both non-deterministic and non-terminating.

To prove a theorem about such least fixed point definitions, we use the built in `induction` tactic that behaves identically for every definition. As an illustration, consider the following theorem:

```
Theorem plus_z_2 : forall X, plus X z X.
```

To prove this theorem, we need to proceed by induction on the structure of `X` (which is of type `nat`). However, as the only induction principles in Abella apply to least fixed point definitions, we need to reify the structure of `nat`s as such a definition:

```
Define is_nat : nat -> prop by
; is_nat z
; is_nat (s X) := is_nat X.
```

We can then state the theorem as follows:[3]

```
Theorem plus_z_2 : forall X, is_nat X -> plus X z X.
```

Note that the type signature of constants is not itself endowed with any induction principles because the signature is open-ended; it may always be extended with new constants of type `nat`, for instance. However, no such extension can falsify the theorem since the `is_nat` predicate is not extensible.

The proof begins by the tactic invocation `induction on 1`, which indicates induction on the first assumption `is_nat X`, called the *inductive argument*. Since there are two definitional clauses for `is_nat`, there will be two cases to consider. In the first case, we would obtain the equation `X = z` in order to match `is_nat X` against the clause *head* `is_nat z`; this in turn instantiates the (eigen)variable `X` to `z`, leaving us with the obligation `plus z z z`, which is easily proved by the first clause of `plus`. In the second case, we would obtain `X = s X1` where `X1` is a new variable for which we know `is_nat X1`. The goal now is `plus (s X1) z (s X1)`, which we can unfold using the second clause of `plus` to reduce it to `plus X1 z X1`. Thus, we are left with the obligation of proving `plus X1 z X1` from the assumption `is_nat X1`.

To close this loop inductively, Abella reasons by induction on the *size* of the inductive argument, which in this case is `is_nat X`.[4] The initial invocation of the `induction` tactic had produced an *inductive hypothesis* `IH`:

```
IH : forall X, is_nat X * -> plus X z X
```

Here, `is_nat X *` stands for an instance of `is_nat` that is strictly smaller than the initial `is_nat X` in the goal. The goal is, in fact, rewritten to:

```
forall X, is_nat X @ -> plus X z X
```

where the annotation `@` indicates that its size is such that every `*`-annotated instance is strictly smaller. Unfolding the assumption `is_nat X @` reduces its size strictly, so that in the case of its second definitional clause we get a new assumption `is_nat X1 *` (where `X = s X1`). This can now be fed to the `IH` to obtain `plus X1 z X1`, which is what we needed to finish the proof.

Size annotations represent (strong) induction on linearly ordered sizes, but meta-theoretic proofs abound with inductions on more complex orderings, particularly lexicographic orderings. While the logic $\mathcal{G}$ underlying Abella has a general induction rule that can represent any (computable) well-ordering, the implementation of it using size annotations and circular inductive hypotheses

[3] Concrete syntax for $\supset$, $\top$, $\bot$, $\wedge$, $\vee$, $\forall$, $\exists$, and ∇: `->`, `true`, `false`, `/\`, `\/`, `forall`, `exists`, and `nabla`. Note that `->` denotes both the function type in definitions and logical implication in theorems.

[4] Every instance of a least fixed point definition is, intuitively, equivalent to $\bot$ unless it can be shown to be true after unfolding it a finite number of times. Thus, *non-terminating* definitions with clauses such as `p X := p (s X)` would be interpreted as $\bot$. The size of such defined atoms is defined in such a way that it is larger than the number of times it needs to be unfolded to determine that it is true. The precise theory of these size annotations is out of scope for this paper but can be found in [13].

in Abella need to be generalized further for lexicographic induction. This is achieved by means of *size levels* that are created by nested invocations of the `induction` tactic.

Nested inductions are best explained by an example: consider this relational definition of the Ackermann function, where `ack X Y K` stands for $\mathtt{K} = A(\mathtt{X}, \mathtt{Y})$.

```
Define ack : nat -> nat -> nat -> prop by
; ack z X (s X)
; ack (s X) z K := ack X (s z) K
; ack (s X) (s Y) K := exists K1, ack (s X) Y K1 /\ ack X K1 K.
```

We may wish to show that this is a total relation, something that famously cannot be done with induction on natural numbers alone:

```
Theorem ack_total : forall X Y, is_nat X -> is_nat Y ->
  exists K, is_nat K /\ ack X Y K.
```

In this case, we want to induct using the lexicographic ordering of the sizes of `is_nat X` and `is_nat Y`, *i.e.*, the inductive hypothesis may be used if the size of `is_nat X` is strictly smaller, or it stays the same and that of `is_nat Y` is strictly smaller. In Abella we write this using the following invocation:

```
induction on 1. induction on 2.
```

This produces *two* inductive hypotheses and modifies the goal as follows:

```
IH1 : forall X Y, is_nat X * -> is_nat Y -> ...
IH2 : forall X Y, is_nat X @ -> is_nat Y ** -> ...
============================
 forall X Y, is_nat X @ -> is_nat Y @@ -> ...
```

where `...` in each case is `exists K, is_nat K /\ ack X Y K`. The hypothesis `IH1` is familiar from before: it just means that `is_nat X *` is strictly smaller than `is_nat X @`. The hypothesis `IH2`, on the other hand, has `is_nat Y **` which is strictly smaller than `is_nat Y @@`. This hypothesis also has an assumption `is_nat X @` which can only be supplied by the corresponding assumption—unmodified!—from the rewritten goal. Note that the `@` and `@@` annotations have no relation to each other except to denote that the latter was introduced while an induction on the former was in progress. Thus, `is_nat X @` would unfold to produce `*` annotations and `is_nat Y @@` would unfold to produce `**` annotations.

In the rest of this development, we will follow a certain style of specifications where typing predicates such as `is_nat` are not explicitly assumed in proofs but are rather produced by inversion on other predicates. Once again, an example illustrates it best: consider the `plus` relation again, but rewritten so that the following theorem holds of it:

```
Theorem plus_is : forall X Y Z, plus X Y Z ->
  is_nat X /\ is_nat Y /\ is_nat Z.
```

Writing it this way means that we never need to assume both `is_nat X` and `plus X Y Z`, for instance, since the former is derivable from the latter. Here is how we modify the definition of `plus` to guarantee `plus_is`:

```
Define plus : nat -> nat -> nat -> prop by
; plus z X X := is_nat X
; plus (s X) Y (s Z) := plus X Y Z.
```

The proof of `plus_is` is by straightforward induction on `plus X Y Z`. Note that we could have sprinkled more `is_nat` conjuncts in the bodies of the definitional clauses, but the above choice is sufficient. We will opt to alter the natural definitional clauses as minimally as possible to yield the necessary inversion lemmas.

2.2. Two-Level Logic Approach

(This sub-section may be skipped on a first reading as it is only relevant for the first-order object logics to come in section 3.5.)

Among the many inductively definable predicates in Abella is a predicate that encodes the derivability in the sequent calculus for ordinary intuitionistic logic, i.e., LJ. Then, meta-theorems of LJ can be formally established in Abella as inductive theorems. For instance, the fact that weakening and contraction (and, more generally, context-monotonicity) is height-preserving admissible in LJ can be formally established, as can the mother of all substitution-like theorems, cut-elimination.

The *two-level logic approach* [12] to specification and reasoning about computations is a general technique where computations are specified relationally in the *specification logic* in such a way that executions of the computation correspond to *derivations* of the specification logic. The strong reasoning logic presented in the previous section is then used to reason about these computations *in terms of* their specification logic derivations. For Abella, the specification logic is chosen to be the negative fragment of intuitionistic logic, which roughly corresponds to higher-order hereditary Harrop logic (with some restrictions) and can be seen as a fragment of the λProlog logic programming language.

This aspect of Abella is covered in much more detail in the tutorial [10]. Here we illustrate it with the specification version of the same predicate `plus` that we saw at the reasoning logic of the previous section. As before, we define a type signature for the constants and predicates and then give clauses for defining the predicate, but in this case instead of definitional clauses we have *program clauses* that are listed in a *module*.

```
sig nat.

kind nat type.

type z nat.
type s nat -> nat.

type is_nat nat -> o.
type plus nat -> nat -> nat -> o.
```

```
module nat.

is_nat z.
is_nat (s X) :- is_nat X.

plus z X X :- is_nat X.
plus (s X) Y (s Z) :-
  plus X Y Z.
```

This specification can be *loaded* into Abella by means of the `Specification` command. Since the specification logic is a fragment of intuitionistic logic, we

depict *sequents* of the specification logic using the notation `{G |- F}` where `G` is a list of specification logic formulas (which is a term of type `olist`) representing the assumptions and `F` is a specification logic formula (a term of type `o`) representing the conclusion. If the assumption list is empty, we can also abbreviate it as `{|- F}`, or just `{F}` The theorem `plus_z_2` is now written as follows, with an identical proof.

```
Theorem plus_z_2: forall X, { is_nat X } -> { plus X z X }.
```

In this paper, the two-level logic approach aspect of Abella will be used very lightly, and only to avoid a certain kind of bureaucracy with respect to induction for the specific case of reasoning about first-order logics. It turns out that when we are performing induction on a specification logic sequent (i.e., a hypothesis of the form `{G |- F}`), then the inductive measure is preserved if we weaken `G` or instantiate some of the free-variables of the sequent with other terms. This in turn is because weakening and instantiation are height-preserving admissible in the sequent calculus LJ. If we were to use ordinary inductive definitions instead of specification sequents, we would have to make these heights explicit and induct on the heights instead, which would greatly increase the notational bureaucracy.

3. Encoding an Object Language

We use the adjective *object* for entities in the logic and proof systems for which we are formally establishing meta-theorems.

3.1. Encoding Object Formulas

In this paper we will focus on linear logics, both propositional and first order. The propositional systems are encoded completely on the reasoning level. For the first order case, formulas and one predicate over formulas were defined in the specification level to facilitate the proofs.

When defined in Abella, formulas of linear logic are encoded as constants of target type `o`. The definition of linear logic formulas for a one-sided formulation of propositional MALL is given in Figure 1. We define a new basic type `atm` of predicates; since type signatures are open-ended in Abella, our development will be parametric over the inhabitants of this type. From this type, atoms and negated atoms are built using `atom` and `natom` respectively. We keep formulas in negation-normal form in the one-sided formulation, so the only formally negated formulas are atoms. Together with these atoms, we define the predicate `is_fm` for inducting on the structure of formulas.

3.2. Multisets

The crucial ingredient in the representation of a one-sided sequent calculus for MALL is the definition of MALL contexts, which must satisfy the following desiderata.

```
Type atom, natom    atm -> o.
Type tens, par      o -> o -> o.
Type one, bot       o.
Type wth, plus      o -> o -> o.
Type top, zero      o.

Define is_fm : o -> prop by
; is_fm (atom A)
; is_fm (natom A)
; is_fm (tens A B) := is_fm A /\ is_fm B
; is_fm one
; is_fm (par A B) := is_fm A /\ is_fm B
; is_fm bot
; is_fm (wth A B) := is_fm A /\ is_fm B
; is_fm top
; is_fm (plus A B) := is_fm A /\ is_fm B
; is_fm zero.
```

Figure 1: Definitions of formulas in Abella.

1. Given two contexts Γ and Δ, we must be able to tell when they are structurally identical, meaning that they contain the same elements with the same multiplicities.
2. Given contexts Γ and Δ and a formula A, we must be able to recognize when adding A to Γ results in a context that is structurally identical to Δ. This operation is required in order to implement inference rules as it is used to represent adding the principal formula in the conclusion of the rule, and the operands of the principal connective (if relevant) to the premises.
3. Generalizing this further, given three contexts Γ, Δ, and Θ, we must be able to say when adding all the elements of Δ to Γ results in a context that is structurally identical to Θ, *i.e.*, Θ is the *join* or the *multiset union* of Γ and Δ. This operation is not only required for implementing multiplicative rules such as $\otimes$, but also for defining the cut rule(s).

There is a wider than expected design space here. A first attempt might be to simply use `olist` as our representation of contexts, with addition of elements represented by list consing (`::`) and context joining with list append. This makes inductive reasoning on contexts rather straightforward, but, because linear contexts are structurally identical modulo exchange, it requires adding explicit exchange rules to the system, which complicates the meta-theory. An alternative that works is still to use `olist` as our representation, but to relax the notion of structural identity as follows: two lists are structurally identical if one is a permutation of the other. Thus, we need a predicate `perm : olist -> olist -> prop` to recognize list permutations.

To define the addition operation with this modified notion, we could continue to use list cons, but this will still require an explicit exchange rule. Instead, we define a generalized cons operation, called `adj`, that adds an element somewhere in an `olist`, not necessarily at the head. Note that this definition is still

```
Type is_o  o -> prop.

Define is_list : olist -> prop by
; is_list nil
; is_list (A :: L) := is_o A /\ is_list L.

% adj J A K : K is J with A inserted somewhere
Define adj : olist -> o -> olist -> prop by
; adj L A (A :: L) := is_o A /\ is_list L
; adj (B :: K) A (B :: L) := is_o B /\ adj K A L.

% merge J K L : J union K equals L.
Define merge : olist -> olist -> olist -> prop by
; merge nil nil nil
; merge J K L := exists A JJ LL, adj JJ A J /\ adj LL A L /\
    merge JJ K LL
; merge J K L := exists A KK LL, adj KK A K /\ adj LL A L /\
    merge J KK LL.

% perm J K : J and K have the same elements
Define perm : olist -> olist -> prop by
; perm nil nil
; perm K L :=
    exists A KK LL, adj KK A K /\ adj LL A L /\ perm KK LL.
```

Figure 2: Implementation of multisets.

sensitive to the order of elements; for example, `adj [b, c] a [a, b, c]` holds, whereas `adj [b, c] a [a, c, b]` does not. Given this definition, it is a simple matter to define `perm` by induction: two lists are permutatively equal if they are produced by `adj`-ing the same elements. Finally, to define the join of `olist`s up to permutations, we simply iterate this process: an `olist` is the join, written `merge`, of two `olist`s if it is produced by `adj`-ing their elements.

The encoding of multisets we have just described is given in Figure 2. The paramater `is_o` will be instantiated with an inductive definition of formulas, which will then be used to reason by induction on formula ranks. The `adj` definition is the most basic building block, and `perm` and `merge` are defined in terms of `adj`. We may conceivably have taken `perm` as primitive and defined the other operations in its terms, or some other combination, but we found our choice to be rather intuitive.[5] Moreover, Abella's built in `search` tactic, which searches for simple proofs of bounded depth, is often able to automatically derive `perm` and `merge` instances.

[5] One apparently simpler definition of permutations is to define `perm J K` to be just `merge J nil K`. While this would not significantly change the development, it enlarges the search space of Abella's `search` engine because of the overlap in the second and third clauses of `merge`, which has a noticeable impact on performance. Moreover, because `search` never unfolds definitions in the context, having a direct recursion of `perm` to itself simplifies certain proofs. In the future Abella will treat nonrecursive definitions as abbreviations that are (un)folded on the fly, at which point we would be able to commit to this simpler definition of `perm`.

$$\frac{}{\vdash a^{\perp}, a}\ \mathsf{init} \qquad \frac{\vdash \Gamma_1, A \qquad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{}{\vdash 1}\ \mathbf{1} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \perp}\ \perp$$

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

Figure 3: One-sided sequent calculus for MALL

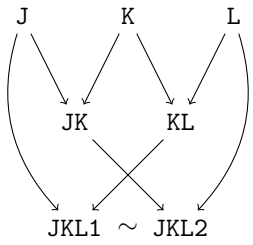
The files `lib/merge.thm` and `lib/perm.thm` contain theorems about multi-sets that are used extensively on the transformations of sequent calculus proofs. These include simple properties, such as `merge`'s stability modulo permutation:

```
Theorem perm_merge_1 : forall J K L JJ,
  merge J K L -> perm J JJ -> merge JJ K L.
```

We also require more complicated lemmas, such as the associativity of `merge`.

```
Theorem merge_assoc : forall J K L JK KL JKL1 JKL2,
  merge J K JK -> merge K L KL ->
  merge J KL JKL1 -> merge JK L JKL2 ->
  perm JKL1 JKL2.
```

This can be depicted more evocatively as follows, where the arrows define the first two arguments to `merge` and $\sim$ denotes `perm`.



Proving this theorem requires establishing that `perm` is an equivalence—specifically that it is transitive—which has a surprisingly unintuitive inductive proof. Such properties are usually taken for granted in informal proofs of cut-elimination. Nevertheless, our experience has been that every property of multisets needed to formalize cut-elimination is proved by straightforward induction.

3.3. One-Sided MALL

We have used the above encoding of multi-sets to define several one and two-sided sequent calculi for fragments of classical linear logic. In this section we give an illustration of the one-sided multiplicative-additive propositional fragment (MALL). Sequents in this fragment are of the form $\vdash \Gamma$, whose inference rules (sans cut) can be found in Fig. 3. The sequent judgment $\vdash \Gamma$ is encoded as the inductively defined atom `mall L`, where `L` represents Γ. The definition of `mall` is given in Fig. 4. Each definitional clause describes, precisely, one of the rules of the sequent calculus, where the head of the clause defines the conclusion of

```
Define mall : olist -> prop by
; mall L := exists A, adj (natom A :: nil) (atom A) L
; mall L := exists A B LL JJ KK J K,
    adj LL (tens A B) L /\ merge JJ KK LL /\
    adj JJ A J /\ mall J /\ adj KK B K /\ mall K
; mall (one :: nil)
; mall L := exists A B LL J K,
    adj LL (par A B) L /\ adj LL A J /\ adj J B K /\ mall K
; mall L := exists LL, adj LL bot L /\ mall LL
; mall L := exists A B LL J K,
    adj LL (wth A B) L /\ adj LL A J /\ mall J /\ adj LL B K /\
        mall K
; mall L := exists LL, adj LL top L
; mall L := exists A B LL J, adj LL (plus A B) L /\ adj LL A J
    /\ mall J
; mall L := exists A B LL J, adj LL (plus A B) L /\ adj LL B J
    /\ mall J.
```

Figure 4: Encoding of one-sided MALL

```
Define dual : o -> o -> prop by
; dual (atom A) (natom A)
; dual (tens A B) (par AA BB) := dual A AA /\ dual B BB
; dual one bot
; dual (plus A B) (wth AA BB) := dual A AA /\ dual B BB
; dual zero top.
```

Figure 5: An asymmetric duality predicate.

the inference rule, and the bodies the premises. The second definitional clause, for instance, first uses `adj` to remove the principal formula `tens A B` from `L`, yielding `LL`; this is then divided into `JJ` and `KK` using `merge`, after which each operand of the $\otimes$ is added to one of the components to build a corresponding `mall` premise.

Note that we do not have an explicit clause for exchange. Nevertheless, we can establish the following theorem by a straightforward induction.

```
Theorem mall_perm : forall K L, mall K -> perm K L -> mall L.
```

Note that the theorem itself says nothing about the sizes of `mall K` and `mall L`, even though it is the case that exchange is height-preserving admissible in the sequent system. The one-sided cut-admissibility theorem will therefore need to be set up in such a way that the size of the result of applying a permutation is not relevant for the inductive hypotheses.

In fact, we will set up the one-sided formulation in such a way that only the height of the derivation containing the positive variant—in the sense of focusing—of the cut formula is relevant. To this end, we define an asymmetric predicate `dual` that relates a positive formula with its dual, depicted in Fig. 5. For the negative formulas, which are the second arguments to `dual`, we will instead prove inversion lemmas by straightforward induction.

```
Theorem bot_inv : forall J L, mall L -> adj J bot L -> mall J.
Theorem par_inv : forall L JJ A B,
  mall L -> adj JJ (par A B) L ->
  exists KK LL, adj JJ A KK /\ adj KK B LL /\ mall LL.
Theorem wth_inv : forall L JJ A B,
  mall L -> adj JJ (wth A B) L ->
  exists KK LL, adj JJ A KK /\ mall KK /\ adj JJ B LL /\ mall LL.
```

These theorems correspond to the following admissible rules.

$$\frac{\vdash \Gamma, \bot}{\vdash \Gamma}\ \bot^{-1} \qquad \frac{\vdash \Gamma, A \parr B}{\vdash \Gamma, A, B}\ \parr^{-1} \qquad \frac{\vdash \Gamma, A \mathbin{\&} B}{\vdash \Gamma, A}\ \&_1^{-1} \qquad \frac{\vdash \Gamma, A \mathbin{\&} B}{\vdash \Gamma, B}\ \&_2^{-1}$$

The cut-admissibility theorem then uses our asymmetric `dual` predicate as follows:

```
Theorem cut : forall A B JJ J KK K LL,
  dual A B ->
  adj JJ A J -> mall J ->
  adj KK B K -> mall K ->
  merge JJ KK LL ->
  mall LL.
```

The proof proceeds by a nested induction on the first and third assumptions. This nesting encodes the following measure for appealing to the inductive hypotheses: either the rank decreases because of case analysis of `dual A B`, or the rank stays the same and the height of `mall J`, which stands for the derivation that contains the positive half of the cut formula pair, decreases.

To illustrate the proof, here is the case where the final rule to be applied in the derivation of `mall J` is $\otimes$. Intuitively, we wish to implement the following reduction:

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\vdash \Gamma_1, A} \qquad \overset{\mathcal{D}_2}{\vdash \Gamma_2, B}}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\otimes \qquad \overset{\mathcal{E}}{\vdash \Gamma_3, A^\perp \parr B^\perp}}{\vdash \Gamma_1, \Gamma_2, \Gamma_3}\ \mathsf{cut} \quad \rightsquigarrow \quad \dfrac{\overset{\mathcal{D}_1}{\vdash \Gamma_1, A} \qquad \dfrac{\overset{\mathcal{D}_2}{\vdash \Gamma_2, B} \qquad \dfrac{\overset{\mathcal{E}}{\vdash \Gamma_3, A^\perp \parr B^\perp}}{\vdash \Gamma_3, A^\perp, B^\perp}\parr^{-1}}{\vdash \Gamma_2, \Gamma_3, A^\perp}\ \mathsf{cut}}{\vdash \Gamma_1, \Gamma_2, \Gamma_3}\ \mathsf{cut}$$

The cut is reduced to two cuts of lower rank, even though the right premise of the cuts can have larger sizes. Note the appeal to the inversion principle for $\parr$.

In the proof, this corresponds to the following proof state[6]:

```
IH : forall A B JJ J KK K LL, dual A B * -> adj JJ A J -> mall J
       -> adj KK B K -> mall K -> merge JJ KK LL -> mall LL
H2 : adj JJ (tens A1 B1) J
H4 : adj KK (par AA BB) K
H5 : mall K
H6 : merge JJ KK LL
H7 : adj LL1 (tens A1 B1) J
H8 : merge JJ1 KK1 LL1
H9 : adj JJ1 A1 J1
```

[6] A proof state is a set of hypotheses and a proof goal.

```
H10 : mall J1 **
H11 : adj KK1 B1 K1
H12 : mall K1 **
H14 : perm JJ LL1
H15 : dual A1 AA *
H16 : dual B1 BB *
============================
 mall LL
```

Applying `par_inv` to `H4` and `H5` yields the new hypotheses:

```
H17 : adj KK AA KK2
H18 : adj KK2 BB LL2
H19 : mall LL2
```

The next step is to create the context $\Gamma_2, \Gamma_3, A^\perp$ which is the conclusion of the first cut on B. In the current proof state, this corresponds to merging `KK1` (Γ_2) and `KK2` ($\Gamma_3, A^\perp$). After merging the contexts, we can apply the inductive hypothesis `IH`, corresponding to the cut on B, which gives us the following new hypotheses:

```
H22 : merge KK1 KK2 L
H23 : mall L
```

Now we need to apply the cut on A, but to build the context of this cut we need to perform a few operations. First, we need to discern `AA` ($A^\perp$) from the context `L` ($\Gamma_2, \Gamma_3, A^\perp$). This is done via the `merge_unadj_2` theorem, which is part of the library of multisets.

```
Theorem merge_unadj_2 : forall J K L KK A,
  merge J K L -> adj KK A K ->
  exists LL, adj LL A L /\ merge J KK LL.
```

After applying this lemma, we will have a variable `LL3` (Γ_2, Γ_3) which we can thus merge with `JJ1` (Γ_1) to get the conclusion of the cut on A. The `IH` can now be applied to obtain the following new hypotheses.

```
H24 : adj LL3 AA L
H25 : merge KK1 KK LL3
H28 : merge JJ1 LL3 L1
H29 : mall L1
```

The case is nearly complete: we just need to show that `L1` is a permutation of `LL`, and then appeal to `mall_perm` on `H29`. Observe that they represent the same context $\Gamma_1, \Gamma_2, \Gamma_3$ but were constructed differently: `LL` is $(\Gamma_1, \Gamma_2), \Gamma_3$ while `L1` is $\Gamma_1, (\Gamma_2, \Gamma_3)$. We can first apply `perm_merge_1` to `H6` and `H14` to obtain:

```
H30 : merge LL1 KK LL
```

Finally, we can apply `merge_assoc` to `H8`, `H25`, `H28`, and `H30` to obtain the required:

```
H31 : perm L1 LL
```

3.4. The Exponential Case

While the MALL cut-elimination proof is long in details, it is not particularly difficult since there is always a single cut to eliminate. The picture gets considerably more complicated with the exponentials, since a single cut rule is no longer sufficient, or, to be more precise, the termination measure for eliminating cuts is much more complex. The main problem is with permuting cuts past contraction rules:

$$\dfrac{\dfrac{\dfrac{\overset{\mathcal{D}}{\vdash ?\Gamma_1, A}}{\vdash ?\Gamma_1, !A}\,!\qquad \dfrac{\overset{\mathcal{E}}{\vdash \Gamma_2, ?A^\perp, ?A^\perp}}{\vdash \Gamma_2, ?A^\perp}\,\mathsf{contr}}{\vdash ?\Gamma_1, \Gamma_2}\,\mathsf{cut}}{} \quad\rightsquigarrow\quad \dfrac{\dfrac{\dfrac{\overset{\mathcal{D}}{\vdash ?\Gamma_1, A}}{\vdash ?\Gamma_1, !A}\,!\qquad \dfrac{\dfrac{\overset{\mathcal{D}}{\vdash ?\Gamma_1, A}}{\vdash ?\Gamma_1, !A}\,!\qquad \overset{\mathcal{E}}{\vdash \Gamma_2, ?A^\perp, ?A^\perp}}{\vdash ?\Gamma_1, \Gamma_2, ?A^\perp}\,\mathsf{cut}}{\vdash ?\Gamma_1, ?\Gamma_1, \Gamma_2}\,\mathsf{cut}^\dagger}{\vdash ?\Gamma_1, \Gamma_2}\,\mathsf{contr}^*$$

The instance of $\mathsf{cut}^\dagger$ is problematic, since neither premise is technically of strictly lower measure: the height and cut-rank is the same in the left-premise, and the right premise is the result of a cut that can be arbitrarily larger.

This problem can be solved in a number of ways, such as by including the number of contractions on the cut-formulas as part of the measure. However, to correctly formulate such a measure, we would need to incorporate multiset orderings, which is not currently supported by Abella's size annotations. We therefore use a different—but still standard—solution of moving to a dyadic sequent calculus[7], with sequents of the form $\vdash \Gamma; \Delta$ where Γ is interpreted as a set— *i.e.*, admitting contraction and weakening—that accumulates the ?-formulas. We use the same type `olist` for Γ. By treating it additively in binary rules and allowing it to be non-empty in axiomatic rules, we have, respectively, contraction and weakening built into the rules. Using this approach, we avoid having to formalize sets as a different type.

Importantly, with this separation of the context into *zones*, we have to increase the number of cut principles to account for occurrences of cut-formulas in both zones. Specifically, the dyadic formulation requires two cuts and a rule of *dereliction* (dl):

$$\dfrac{\vdash \Gamma; \Delta_1, A \qquad \vdash \Gamma; \Delta_2, A^\perp}{\vdash \Gamma; \Delta_1, \Delta_2}\,\mathsf{cut} \qquad \dfrac{\vdash \Gamma; A \qquad \vdash \Gamma, A^\perp; \Delta}{\vdash \Gamma; \Delta}\,\mathsf{ucut} \qquad \dfrac{\vdash \Gamma, A; \Delta, A}{\vdash \Gamma, A; \Delta}\,\mathsf{dl}$$

The conditions for appealing to the inductive hypothesis is now more complicated. An IH can be used if the cut-rank is smaller, or if the derivation with A is of lower height, but in the case where both stay the same we can reduce a cut to a ucut. The issue with contractions above reappears as an issue with dereliction as follows:

$$\dfrac{\overset{\mathcal{D}}{\vdash \Gamma; A} \qquad \dfrac{\overset{\mathcal{E}}{\vdash \Gamma, A^\perp; \Delta, A^\perp}}{\vdash \Gamma, A^\perp; \Delta}\,\mathsf{dl}}{\vdash \Gamma; \Delta}\,\mathsf{ucut} \quad\rightsquigarrow\quad \dfrac{\overset{\mathcal{D}}{\vdash \Gamma; A} \qquad \dfrac{\overset{\mathcal{E}}{\vdash \Gamma, A^\perp; \Delta, A^\perp}}{\vdash \Gamma; \Delta, A^\perp}\,\mathsf{ucut}}{\vdash \Gamma; \Delta}\,\mathsf{cut}$$

[7] See, for example, [14].

However, since a ucut is allowed to justify a cut, there is no termination issue.

To encode the ordering between the two cuts, we need to induct on an additional *weight* parameter to the cut theorem that determines the kind of cut. We encode it in Abella as follows:

```
Kind weight type.

Type heavy, light weight.

Define is_weight : weight -> prop by
; is_weight light
; is_weight heavy := is_weight light.
```

Note that `is_weight heavy` and `is_weight light` are both true, but the former requires strictly more unfolding operations to derive it. This is sufficient to order the two cuts. A more annoying problem comes from the case of permuting a cut past a dereliction. In this case, one of the branches of the derivation, the branch with the ?-formula, has a strictly larger number of non-linear formulas. It turns out that we need to explicitly account for this increased size by means of the following *inclusion* relation:

```
Define incl : olist -> olist -> prop by
  incl G D := exists W, merge G W D.
```

In other words, `incl G D` means that `D` is an extension of `G`. We now have all the ingredients to define the general cut theorem.

```
Theorem cut : forall A B QL J QQL W,
  dual A B -> mell QL J -> is_weight W ->
  incl QL QQL ->
     (W = light /\
      forall JJ KK K LL,
        adj JJ A J ->
        adj KK B K -> mell QQL K ->
        merge JJ KK LL ->
        mell QQL LL)
  \/ (W = heavy /\
      forall QQ K,
        J = A :: nil ->
        adj QQL B QQ -> mell QQ K ->
        mell QQL K).
```

The first disjunct represents cut, while the second is ucut. The proof then begins:

```
induction on 1. induction on 2. induction on 3.
```

which encode the required lexicographic measure. Observe that once the theorem is proved, each disjunct can be individually obtained by instantiating `W` with `light` and `heavy` respectively. This yields the following pair of corollaries, which are precisely the two cuts we are interested in.

```
Theorem cut_admit_linear : forall A B JJ J KK K QL LL,
  dual A B ->
  adj JJ A J -> mell QL J ->
  adj KK B K -> mell QL K ->
  merge JJ KK LL ->
  mell QL LL.
```

```
kind term, atm, fm type.

type atom, natom   atm -> fm.
type tens, par     fm -> fm -> fm.
type one, bot      fm.
type wth, plus     fm -> fm -> fm.
type top, zero     fm.
type all, exs      (term -> fm) -> fm.
```

Figure 6: Definition of formulas in λProlog.

```
Type form fm -> o.

Define is_fm : o -> prop by
; is_fm (form (atom A))
; is_fm (form (natom A))
; is_fm (form (tens A B)) := is_fm (form A) /\ is_fm (form B)
; is_fm (form one)
; is_fm (form (par A B)) := is_fm (form A) /\ is_fm (form B)
; is_fm (form bot)
; is_fm (form (wth A B)) := is_fm (form A) /\ is_fm (form B)
; is_fm (form top)
; is_fm (form (plus A B)) := is_fm (form A) /\ is_fm (form B)
; is_fm (form zero)
; is_fm (form (all A)) := nabla x, is_fm (form (A x))
; is_fm (form (exs A)) := nabla x, is_fm (form (A x)).
```

Figure 7: Predicate for inducting on the formula structure in the two-level implementation.

```
Theorem cut_admit_exponential : forall A B QL QQ K,
  dual A B ->
  mell QL (A :: nil) ->
  adj QL B QQ -> mell QQ K ->
  mell QL K.
```

3.5. The First-Order Case

As mentioned in Section 2.2, we will be using the two-level logic approach to simplify reasoning about quantifiers and instancing. We will move the representation and structure of linear logic formulas from the reasoning level we have seen thus far to the specification logic (λProlog) level of Abella. When defined in λProlog, formulas of linear logic have type `fm` (Figure 6). Since the specification logic is first order, it cannot quantify over terms of type `o` (reserved for predicates). Therefore we need to declare a new type for object formulas. On the reasoning level this type is coerced into `o` by the predicate `form : fm -> o`. In this case, the definition of `is_fm` is slightly altered to the one in Figure 7. In fact, prefixing formulas with `form` is the only major change needed when the definition of formulas is moved to the specification layer.

The quantifiers $\forall$ (`all`) and $\exists$ (`exs`) are defined using λ-tree syntax (sometimes called *higher-order abstract syntax*), meaning that binders in the object language are encoded by means of λ-abstraction in λProlog. Thus, the type of the arguments of these constants is `term -> fm`.

As an example, the formula:

$$\forall x.\, \exists y.\, \big(\mathtt{p}(x)^{\perp} \parr \mathtt{p}(y)\big)$$

where `p` is an atomic predicate, would be represented syntactically with the concrete term:

```
all (x\ exs (y\ (par (natom (p x)) (atom (p y)))))
```

Here, the notation `x\ t` is the concrete syntax for a λ-abstraction where the variable `x` is abstracted in `t`. The abstracting operator `\` is written infix and has the lowest precedence. Note that in this representation we leave the types mostly unstated. In fact, both `x` and `y` would have type `term`, and `p` would have type `term -> atm`. Types are almost always inferred in Abella and so we will omit them in the rest of this paper except when they are needed to disambiguate.

All that remains is to define the `is_fm` predicate on this new encoding of formulas. This is depicted in figure 7. The only significant difference from the version of this predicate in section 3.1 is the use of the intensional `nabla` quantifier in the cases for the quantifiers. For the purposes of this paper, it is sufficient to interpret this quantifier as a `forall` as it does not affect any of the meta-theorems. Since we largely use `dual` instead of `is_fm` to drive our induction in the cut-admissibility theorem, we also move that predicate to the specification logic.

```
type dual fm -> fm -> o.

dual (atom A) (natom A).
dual (tens A B) (par AA BB) :- dual A AA, dual B BB.
dual one bot.
dual (plus A B) (wth AA BB) :- dual A AA, dual B BB.
dual zero top.
dual (exs A) (all B) :- pi x\ dual (A x) (B x).
```

The definition of the calculus is the same as the one in Figure 4, except that all formulas are prefixed by the `form` predicate (for "casting" from type `fm` to `o`).

Existential quantification is encoded using Abella's `exists` quantifier. Universal quantification, on the other hand, uses Abella's intensional quantifier `nabla` (∇) that quantifies over distinct variables. This latter quantifier is used to model the parametric quantifier `pi` of the specification language, used in the final of `dual` above. The difference between `forall` and `nabla` is that the latter encodes a freshness condition based on the quantification order: in the order `forall x, nabla y`, it must be the case that `y` is a variable that cannot occur free in `x`, i.e., `y` is *fresh for* `x`. Thus, for instance, in the last clause below we know that `x` is fresh for `L`, `A`, and `LL`. However, it is *not* fresh for `J`, meaning that `J` is allowed to contain `x`.

```
Define mall : olist -> prop by
```

```
... [propositional rules] ...
; mall L := exists x A LL J,
      adj LL (form (exs A)) L /\ adj LL (form (A x)) J /\ mall J
; mall L := exists A LL,
    adj LL (form (all A)) L /\
    nabla x, exists J, adj LL (form (A x)) J /\ mall J.
```

Using this encoding, the cut-elimination theorem needs to be only slightly altered by adding brackets to the `dual` predicate:

```
Theorem mall_cut : forall A B JJ J KK K LL,
  { dual A B } ->
  adj JJ (form A) J -> mall J ->
  adj KK (form B) K -> mall K ->
  merge JJ KK LL ->
  mall LL.
```

Its proof follows the same double induction as in the case of propositional MALL.

It is worth highlighting why we chose to use two levels instead of using our earlier encodings of formulas purely in the reasoning level. Had we done that, the constants `all` and `exs` representing the quantifiers would have had the type `(term -> o) -> o`. When developing the same proof as before, complications arise on the following cut reduction:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\vdash \Gamma, (A\,t)}}{\vdash \Gamma, \exists x.(A\,x)}\,\exists \qquad \overset{\mathcal{E}}{\vdash \Delta, \forall x.(A\,x)}}{\vdash \Gamma, \Delta}\,\mathsf{cut} \quad \rightsquigarrow \quad \dfrac{\overset{\mathcal{D}}{\vdash \Gamma, (A\,t)} \qquad \dfrac{\overset{\mathcal{E}}{\vdash \Delta, \forall x.(A\,x)}}{\vdash \Delta, (A\,t)}\,\forall^{-1}(t)}{\vdash \Gamma, \Delta}\,\mathsf{cut}$$

(Here, the rule $\forall^{-1}(t)$ stands for the application of the inverse of the $\forall$ rule, which introduces a parameter that is then instantiated by the term t.)

The usual argument is that the induction hypothesis can be applied because the cut on the right derivation is on a smaller formula, but Abella cannot verify this claim. Specifically, we have an inductive argument (i.e., an argument with a `*` annotation) for the formula `all A`, but we need to produce an inductive argument for a term containing `A t`. However, `A t` is not a structural subterm of `all A`, and hence Abella has no way of knowing that it represents a formula of strictly smaller rank. This could have been addressed by making the rank of a formula explicit in, say, the `dual` predicate by an auxiliary `nat` argument. However, we deftly avoid this tedium because, as it turns out, the specification level definition of `dual` tells us that given a hypothesis `{dual (exs P) (all N)}*`, unfolding it yields `nabla x, {dual (P x) (N x)}*`, where the `*` annotation is preserved. Now, by means of the `inst` tactic (which implements height-preserving instantiation of the specification logic), we can get `{dual (P t) (N t)}*` for any term `t`. The formalization of first-order MALL can be found on the website.

3.6. Focused Calculi

Going up further in the level of meta-theoretic complexity, we have also attempted to show cut-admissibility of *focused* proof systems for linear logic [14]. Focused proofs are a natural reorganization of the "small step" style of linear logic

rules into a more "macro" or "synthetic" form where connectives of like polarity are clustered together in maximal clumps, drastically cutting down the noise (i.e., the non-determinism) in sequent proofs without sacrificing completeness or expressivity. The main difficulty in focused proofs is that cuts are not permuted step by step above single connectives but must rather permute past entire "phases". Writing out the details of such a large cut-permutation argument is notoriously tricky, and often involves the use of a number of administrative and auxiliary cuts.

Despite these complications, we were able to complete the cut-elimination argument for focused first-order MALL with relative ease, with the final proof turning out to be even more compact than similar proofs for the unfocused case. This reduction in size came primarily from the use of a single general inversion property that is folded into the inference rules, instead of separate inductively established inversion theorems connective by connective. We are in the process of expanding this formalization to the full focused first-order linear logic (i.e., containing exponentials). The most recent formalization for focused first-order MALL can be found on the website. More detail would be somewhat beyond the scope of this introductory paper and will be left to a followup article that is devoted to specific techniques for representing focused sequent calculi.

3.7. Two-Sided Calculi

We have also implemented the meta-theory of the two-sided sequent calculus for MALL. The big differences between the one-sided and two-sided formulations are that each connective has left and right introduction rules, and that the cut rules apply to formulas on either side of the sequent arrow rather than in terms of duality. Hence, we reason directly on `is_fm` instead of in terms of an asymmetric `dual` predicate, which in turn means that we do an additional nested induction instead of appealing to inversion lemmas. Cuts are now permuted upwards in both premise derivations until they become principal. While the proofs are now longer because of the larger number of inference rules, the ingredients remain largely the same. It is worth noting that the cut permutations proved in the two-sided system can be used to show strong normalization of a cut-elimination strategy for MALL, given an ordering of the cuts.

4. Related work

We are certainly not the first to formalize a cut-elimination proof in a proof assistant. We discuss here a few other projects on this direction and compare them with our approach. This list is far from exhaustive.

Closest to our approach (in the sense that sequents and multisets are encoded) we can cite [5] and [15]. In [5] the authors propose a generic method for formalizing sequent calculi in Isabelle/HOL, making all lemmas and theorems parametric on a set of rules. For the main cut-elimination theorem, weakening must be admissible. They have proved cut-elimination for the sequent calculus GLS_V for provability logic, although in practice they proved the admissibility

of *multi-cut* instead of cut itself (the rules are shown to be equivalent for their system). The use of multi-cut is justified to avoid the complicated cases where the cut-formula is contracted, which is also our approach to exponentials.

A proof of cut-elimination for coalgebraic logics by Pattinson and Schröder was formalized in Coq in [15]. The formalization uncovered a few mistakes in the original proof which were discussed with Pattinson and Schröder and corrected. The author has implemented multisets as setoids in Coq with lists as the underlying type and permutation as the equivalence relation. Our treatment of multisets is largely similar. The author also chose to define a type for heterogeneous lists for lists of a fixed size as part of the encoding. As in [5], the proof is parameterized by a rule set.

To avoid dealing with explicit representations of contexts as multisets, a common approach is to find a different representation for sequent calculus rules which mention explicitly only the principal and auxiliary formulas. This is the path followed in [16], [17], [18] and [19]. In [16] the author annotates sequents of the calculus considered with proof terms, reducing cut-elimination to a type checking problem on those terms. Since the logical framework is intuitionistic, the structural rules of contraction and weakening are forced to be admissible for all such proof calculi. One of the obvious advantages of this approach is avoiding explicit representations of multisets; on the other hand, the adequacy of the term rewriting system to the actual sequent calculus rules is only an informal argument that is not independently verified.

The method developed in [16] was used in [17] for formalizing a proof of completeness of focusing for intuitionistic logic. The author avoids having to show "tedious invertibility lemmas" by using a new proof of completeness that follows from cut-elimination and generalized identity. A number of meta-theoretical properties are proved in this formalization. It would be interesting to see if his proof of completeness of focusing could be formalized in Abella using our results. One commonality between our approach and that of [17] is the use of cut weights to set up a lexicographic measure. Another formal proof (in Coq) of completeness of focusing for several systems was developed in [18], using an algebraic interpretation of the logics which requires some assumptions on the sequent calculus, namely *harmony* (i.e., rules should come in "dual" pairs) and the admissibility of weakening and contraction. It is difficult to see how these ideas can be generalized to the substructural case.

Other linear logic encodings in various proof assistants can be found in [20, 21, 22]. The goal of those works however was to obtain proof search engines for linear logic, so there are no proofs of meta-theoretical properties of the encoded systems. In [20], the author implements linear logic in Isabelle and uses a calculus with exchange rules to avoid having to implement contexts modulo permutation. The same solution is used in [22] for implementing linear logic in Coq, although a later implementation by Cowley[8] uses permutation of contexts. In [21] linear logic is again implemented in Isabelle for proof search, but this time rules are

[8] `https://github.com/acowley/LinearLogic`

encoded using multisets. On top of regular linear logic rules, the authors also add a set of "macro-rules" to the system for facilitating proof search.

Another implementation of linear logic in Coq has recently been developed, this time with the goal of proving meta-theoretical properties [23]. The authors have implemented a multi-set library to encode the contexts, and used parametric higher-order abstract syntax for encoding quantifiers on the first order case. One major difference from our encoding is that many induction measures are stated explicitly as part of the definitions. This development includes proofs of cut-elimination for propositional and first-order LL (one-sided calculi), and a proof of completeness of focusing. The latter is a formalization of Andreoli's proof in [14]. Another big difference is that this proof relies on the *parametric higher-order abstract syntax* approach that necessitates a certain library of axiomatic extensions in order to get a usable induction measure. This is necessitated by the fact that the full (strong) higher-order abstract syntax cannot be supported by a system such as Coq that only allows strictly positive inductive definitions.

5. Conclusion

We have shown an implementation of a "textbook" proof of cut-elimination, using the rewrite rules *à la* Gentzen, for various fragments of linear logic in the proof assistant Abella. This is the first formalization of cut-elimination for linear logic to the best of our knowledge. We have also implemented proofs of other meta-theoretical properties using the same techniques. It required the implementation and proofs of several lemmas about multisets, which we believe can be re-used for meta-theoretical proofs about other calculi. The encoding of sequent calculus rules is quite intuitive and similar to a logic program.

While formalizing this proof we have learned a few interesting things. First of all, it was good to realize that proof assistants are already usable enough to handle such proofs. We were skeptical about this at some points. In fact, we have started translating the Abella code to Coq. The multiset library is fully specified and we have some proofs for meta-theorems of MLL. Because Coq allows for fine programmatic control of proof search, nearly all the required lemmas about the representation of multisets are handled with single invocations of a simplification tactic tailor-made for reasoning about multi-sets (written using Ltac). On the other hand, Coq's induction is more primitive than Abella's and requires making the induction measure explicit, which in turn complicates meta-theoretic proofs, particularly those that rely on lexicographic induction. Of course this might be caused by using an "Abella way of thinking" when implementing the proofs in Coq. We noticed that a familiarity with the proof assistant plays a big role when finding out the lemmas to prove and the proof strategy to follow. Since each proof assistant is unique, reproving something in another software is not so straightforward.

This being said, and despite the fact that we successfully finished several such proofs, we must admit that the amount of boilerplate in the proofs shows us that this approach is not yet ready for general purpose use. The trade-off between having a formalized proof and the time taken to formalize it is still

too big for the average proof theorist. We believe this to be a general problem with proof assistants—not just with Abella—given the related work we have found and our experience in porting the code to Coq. Modularity techniques in proof assistants are already a great help (indeed we have one implementation of multisets which is used by all encodings), but there is still a considerable gap between the kinds of informal meta-theoretic proofs one finds in the average proof theory paper and the formalizations. We are investigating better ways to deal with the tedious and repetitive parts of proofs.

References

[1] V. Danos, Une Application de la Logique Linéaire a l'Etude des Processus de Normalisation (principalement du λ-calcul), Ph.D. thesis, Université Paris (1990).

[2] G. Bierman, A note on full intuitionistic linear logic, Annals of Pure and Applied Logic 79 (3) (1996) 281 – 287. doi:http://dx.doi.org/10.1016/0168-0072(96)00004-8.

[3] T. Braüner, V. de Paiva, Cut-Elimination for Full Intuitionistic Linear Logic, Tech. Rep. BRICS-RS-96-10, BRICS, Aarhus, Danemark, also available as Technical Report 395, Computer Laboratory, University of Cambridge (1996).

[4] R. Goré, R. Ramanayake, Valentini's cut-elimination for provability logic resolved, The Review of Symbolic Logic 5 (2012) 212–238. doi:10.1017/S1755020311000323.

[5] J. E. Dawson, R. Goré, Generic Methods for Formalising Sequent Calculi Applied to Provability Logic, in: Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, 2010. Proceedings, 2010, pp. 263–277. doi:10.1007/978-3-642-16242-8_19.

[6] L. Pinto, T. Uustalu, Proof Search and Counter-Model Construction for Bi-intuitionistic Propositional Logic with Labelled Sequents, in: Automated Reasoning with Analytic Tableaux and Related Methods: 18th International Conference, TABLEAUX 2009. Proceedings, Springer, 2009, pp. 295–309. doi:10.1007/978-3-642-02716-1_22.

[7] S. Marin, L. Straßburger, Label-free Modular Systems for Classical and Intuitionistic Modal Logics, in: Advances in Modal Logic 10, invited and contributed papers from the tenth conference on "Advances in Modal Logic", 2014, pp. 387–406.
URL http://www.aiml.net/volumes/volume10/Marin-Strassburger.pdf

[8] A. Chlipala, Parametric higher-order abstract syntax for mechanized semantics, in: Proceeding of the 13^{th} ACM SIGPLAN international conference on Functional programming, ICFP, 2008, pp. 143–156. doi:10.1145/1411204.1411226.

[9] K. Chaudhuri, L. Lima, G. Reis, Formalized Meta-Theory of Sequent Calculi for Substructural Logics, Electronic Notes in Theoretical Computer Science 332 (2017) 57 – 73, lSFA 2016 - 11th Workshop on Logical and Semantic Frameworks with Applications (LSFA). doi:https://doi.org/10.1016/j.entcs.2017.04.005.

[10] D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, Y. Wang, Abella: A System for Reasoning about Relational Specifications, Journal of Formalized Reasoning 7 (2). doi:10.6092/issn.1972-5787/4650.

[11] A. Gacek, D. Miller, G. Nadathur, Nominal abstraction, Information and Computation 209 (1) (2011) 48–73. doi:10.1016/j.ic.2010.09.004.

[12] A. Gacek, D. Miller, G. Nadathur, A two-level logic approach to reasoning about computations, Journal of Automated Reasoning 49 (2) (2012) 241–273. doi:10.1007/s10817-011-9218-1.
URL http://arxiv.org/abs/0911.2993

[13] A. Gacek, A Framework for Specifying, Prototyping, and Reasoning about Computational Systems, Ph.D. thesis, University of Minnesota (2009).

[14] J.-M. Andreoli, Logic Programming with Focusing Proofs in Linear Logic, Journal of Logic and Computation 2 (3) (1992) 297–347.

[15] H. Tews, Formalizing Cut Elimination of Coalgebraic Logics in Coq, in: Automated Reasoning with Analytic Tableaux and Related Methods: 22nd International Conference, TABLEAUX 2013, Proceedings, Springer, 2013, pp. 257–272. doi:10.1007/978-3-642-40537-2_22.

[16] F. Pfenning, Structural Cut Elimination, in: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, LICS '95, IEEE Computer Society, 1995, pp. 156–.

[17] R. J. Simmons, Structural Focalization, ACM Transactions on Computational Logic 15 (3) (2014) 21:1–21:33. doi:10.1145/2629678.

[18] S. Graham-Lengrand, Polarities & Focussing: a journey from Realisability to Automated Reasoning, Habilitation thesis, Université Paris-Sud (2014).
URL http://hal.archives-ouvertes.fr/tel-01094980

[19] C. Urban, B. Zhu, Revisiting Cut-Elimination: One Difficult Proof Is Really a Proof, in: Rewriting Techniques and Applications: 19th International Conference, RTA 2008, Proceedings, Springer, 2008, pp. 409–424. doi:10.1007/978-3-540-70590-1_28.

[20] P. Groote, Linear logic with isabelle: Pruning the proof search tree, in: Theorem Proving with Analytic Tableaux and Related Methods: 4th International Workshop, TABLEAUX, Springer, 1995, pp. 263–277. doi:10.1007/3-540-59338-1_41.

[21] S. Kalvala, V. D. Paiva, Mechanizing linear logic in Isabelle, in: In 10th International Congress of Logic, Philosophy and Methodology of Science, 1995.

[22] J. Power, C. Webster, Working with Linear Logic in Coq, in: 12th International Conference on Theorem Proving in Higher Order Logics, 1999, pp. 1–16.

[23] B. Xavier, C. Olarte, G. Reis, V. Nigam, Mechanizing Linear Logic in Coq, in: Proceedings of the 12^{th} Workshop on Logical and Semantics Frameworks with Applications (LSFA), 2017, pp. 60–77.