Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 338 (2018) 219–236

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Mechanizing Focused Linear Logic in Coq

Bruno Xavier and Carlos Olarte[1]

Universidade Federal do Rio Grande do Norte (UFRN), Natal, Brazil

Giselle Reis[2]

Carnegie Mellon University, Doha, Qatar

Vivek Nigam

Centro de Informática, Universidade Federal da Paraíba João Pessoa, Brazil, and Fortiss, Munich, Germany

Abstract

Linear logic has been used as a foundation (and inspiration) for the development of programming languages, logical frameworks and models for concurrency. Linear logic's cut-elimination and the completeness of focusing are two of its fundamental properties that have been exploited in such applications. Cut-elimination guarantees that linear logic is consistent and has the so-called sub-formula property. Focusing is a discipline for proof search that was introduced to reduce the search space, but has proved to have more value, as it allows one to specify the shapes of proofs available. This paper formalizes first-order linear logic in Coq and mechanizes the proof of cut-elimination and the completeness of focusing. Moreover, the implemented logic is used to encode an object logic, such as in a linear logical framework, and prove adequacy.

Keywords: linear logic, focusing, Coq

1 Introduction

Linear Logic was proposed by Girard [12] more than 30 years ago, but it still inspires computer scientists and proof theorists alike, being used as a foundation for programming languages, models of concurrency [18,8,19] and logical frameworks [16]. Such developments rely on two fundamental properties of linear logic: cut-elimination [11] and the completeness of focusing [1].

[1] The work of Carlos Olarte was supported by CNPq, the STIC AmSud - Capes project "EPIC: EPistemic Interactive Concurrency" (Proc. No 88881.117603/2016-01) and the project FWF START Y544-N23.

[2] Giselle Reis was funded by grant NPRP 7-988-1-178 from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

https://doi.org/10.1016/j.entcs.2018.10.014

Cut-elimination states that any proof with instances of the cut-rule:

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^{\perp}}{\vdash \Gamma, \Delta}$$

can be transformed into a proof of the same formula without any instance of the cut-rule. The two main consequences of this theorem are: (1) the system's consistency, *i.e.*, it is not possible to prove both $\vdash A$ and $\vdash A^{\perp}$; and (2) all proofs satisfy the sub-formula property, *i.e.*, a proof of a formula A, contains only sub-formulas of A. *Focusing* is a proof search discipline proposed by Andreoli [1] which constraints proofs by enforcing that rules sharing some structural property, like invertibility, are grouped together. The completeness of focusing states that if a formula has a proof, then it has a focused proof.

The combination of cut-elimination and focusing allows for the construction of powerful linear logical frameworks. By relying on these two properties, proof search is considerably improved. Moreover, these properties can be used to engineer the types of proofs, thus allowing the specification/encoding of a number of different proof systems (e.g., sequent calculus, natural deduction and tableaux systems) for different logics [16,17].

This paper presents a formalization of first-order classical linear logic (LL) in Coq, including proofs for cut-elimination and completeness of focusing. Up to the best of our knowledge, this is the first formalization of the meta-theory of first-order linear logic in Coq (see Section 6). Our main contributions are listed below:

(1) Quantifiers: Most of the formalizations of cut-elimination in the literature deal with propositional systems (see Section 6). The first-order quantifiers of LL are fundamental for the adequacy results in Section 5. We use the technique of Parametric HOAS [4] (*i.e.*, dependent types in Coq) to encode the LL quantifiers in the meta-logic. This alleviates the burden of specifying substitution and freshness conditions. However, it comes at a price, as substitution lemmas and structural preservation under substitutions need to be assumed as axioms (see details in Section 2).

(2) Focusing completeness: While cut-elimination theorems for a number of proof systems have been formalized, including propositional linear logic [3], this is, as far as we know, the first formalization of first-order LL's cut-elimination and LL's focused proof system completeness. The proof formalized is exactly the one presented by Andreoli [1]. It involves a number of non-trivial proof transformations.

(3) Encoding proof systems: By relying on linear logic's cut-elimination and focusing property, it is possible to encode a number of proof systems [17,16]. Focusing is used to achieve the highest level of adequacy (*i.e.*, from rules to partial derivations). This is done by engineering derivations available in the LL framework to match derivations of the encoded proof system. We build a tactic that automatically handles the whole negative phase of the proof, making proofs shorter and simpler for the specifier/programmer. The mechanized proofs are similar to the paper proofs and quite direct. We demonstrate this by encoding the system LJ for intuitionistic propositional logic into LL.

(4) Treatment of contexts: A main challenge in LL arises from the fact that contexts are treated as multisets of formulas and not as sets (due to the lack of weakening and contraction) . We show that the exchange rule is admissible in LL, i.e., if $\vdash \Gamma$ is provable in LL and Γ' is a permutation of Γ then $\vdash \Gamma'$ is provable in LL. We use the library `Morphisms` of Coq to easily substitute, in any proof, equivalent multisets. For doing that, we extended the standard library for multisets in Coq with additional theorems. Moreover, we developed tactics that handle (automatically) most of the proofs of multisets in our formalization.

(5) Induction measures: Although more relaxed measures for the height of derivations can be used as induction measures (*e.g.*, the axiom rule can have height n for any n), we decided to follow carefully the induction measures used in the proof of cut-elimination and the completeness of focusing in, *e.g.*, [25] and [1]. Hence, our proofs reflect exactly the procedures described in the literature.

Our formalization is available at `https://github.com/meta-logic/coq-ll` and the organization of the paper follows closely the structure of the files. Section 2 deals with the syntax of LL and the snippets of code are from `Syntax.v`. Section 3 deals with different sequent calculi for LL and a focused system for it (`SequentCalculi.v`). Section 4 presents several results about LL: structural properties (`SequentCalculiBasicTheory.v`), cut-elimination (`Cut_Elim.v`) and completeness of focusing (`Completeness.v`). Section 5 shows the application of our formalization to prove correct the encoding of LJ into LL (`LJLL.v`). Section 6 discusses related and future work. We present here some of the most important definitions and key cases in the proofs of the theorems. It is important to note that, for the sake of presentation, we omit some cases (e.g., in inductive definitions) and we also change marginally the notation to improve readability. Also, in theorems' statements, all the variables are implicitly universally quantified. The reader may always consult the complete definitions and proofs in the source files.

2 Linear Logic Syntax

Linear logic (LL) [12] is a resource conscious logic, in the sense that formulas are consumed when used during proofs, unless they are marked with the exponential ? (whose dual is !), in which case, they behave *classically*. LL connectives include the additive conjunction & and disjunction $\oplus$ and their multiplicative versions $\otimes$ and $\parr$, together with their units and the first-order quantifiers:

$$\begin{array}{rcl} A, B, \ldots & ::= & \underbrace{a}_{\text{LITERAL}} \mid \underbrace{A \otimes B \mid 1}_{\text{MULTIPLICATIVE}} \mid \underbrace{A \oplus B \mid 0}_{\text{ADDITIVE}} \mid \underbrace{\exists x.A}_{\text{QUANTIFIED}} \mid \underbrace{!A}_{\text{EXPONENTIAL}} \\ & \mid & \underbrace{a^{\perp}}_{\text{LITERAL}} \mid \underbrace{A \parr B \mid \perp}_{\text{MULTIPLICATIVE}} \mid \underbrace{A \& B \mid \top}_{\text{ADDITIVE}} \mid \underbrace{\forall x.A}_{\text{QUANTIFIED}} \mid \underbrace{?A}_{\text{EXPONENTIAL}} \end{array} \tag{1}$$

The main issue when formalizing first-order logics in proof assistants is how to encode quantifiers. At first glance, one might consider the following naive signature for the constructors `fx` and `ex` of universally and existentially quantified formulas, respectively:

```
| fx : (var → formula) → formula        | ex : (var → formula) → formula
```

In order to define substitutions of variables for terms on such formulas, it is necessary to define a `term` type as a union of, *e.g.*, `var`s and `function`s; and also to implement substitution from scratch. This means dealing with variable capture and equality of terms.

It is possible to avoid this unnecessary bureaucracy if substitution is handled by the meta-level β-reduction. This means that a quantified formula $Qx.F$ ($Q \in \{\forall, \exists\}$) is represented as $Q(\lambda x.F)$, where λ is a meta-level binder. In this case, we have:

```
| fx : (term → formula) → formula        | ex : (term → formula) → formula
```

This approach is called *higher-order abstract syntax* (HOAS) or λ-tree syntax [21,15]. In a functional framework, the type `(term → formula)` ranges over all functions of this type. This is not desirable as it allows functions, called *exotic terms* [6], to pattern match on the term and return a structurally (or logically) different formula for each case. Note that, in a relational framework, this is not a problem, since all definitions must have the type of propositions on the meta-level (different from the type of formulas of the object logic).

A solution for this problem is either to require that `term` is an uninhabited type or to quantifying over all types as below:

```
| fx : forall T, (T → formula) → formula        | ex : forall T, (T → formula) → formula
```

However, in both specifications, it is impossible to write a function that computes the size of a formula (a good description of these problems can be found at http://adam.chlipala.net/cpdt/html/Hoas.html). A solution proposed in [4] consists of parametrizing the type `T` for quantified variables not in quantifiers' constructors, but outside the whole specification. This approach is called *parametric* HOAS. Using this technique, linear logic's syntax is formalized as follows.

```
Section Sec_lExp.
  Variable T: Type.              (* Parameter for building variables *)
  Inductive term :=              (* Terms *)
   | var (t: T)  | cte (e: A)    (* variables and constants *)
   | fc1 (n: nat) (t: term)   | fc2 (n: nat) (t1 t2: term). (* family of functions *)

  Inductive aprop :=                 (* Atomic propositions *)
   | a0: nat → aprop                 (* 0-ary predicates *)
   | a1: nat → term → aprop | a2: nat → term → term → aprop. (* family of predicates *)

  Inductive lexp := (* Formulas *)
   | atom (a :aprop) ) | perp (a: aprop)    (* positive/negated atoms *)
   | top | bot | zero | one                 (* units *)
   | tensor (F G: lexp)  | par (F G: lexp)  (* multiplicative *)
   | plus (F G: lexp)  | witH (F G: lexp)   (* additive *)
   | bang (F: lexp) | quest (F: lexp)       (* exponentials *)
   | ex (f: T → lexp) | fx (f: T → lexp).   (* quantifiers *)
End Sec_lExp.
```

The type `A` of constant terms is a global parameter. The signature of first-order terms includes a family of functions `fc1` and `fc2` of one and two arguments respectively. In each case, the first parameter (`nat`) is the identifier, *i.e.*, the name of the function. Atomic propositions can be 0-ary (`a0`), unary (`a1`) or binary (`a2`) predicates. As in the case of functions, a natural number acts as the identifier. The rest of the code should be self-explanatory. We note that more general constructors

for functions and predicate using a list (of arbitrary length) of parameters could have been defined. However, the current signature is general enough for our encodings and it greatly simplifies the notation.

All types defined in `Section Sec_lExp` are parametrized by the type `T`. Therefore, the type of `top`, for example, is not `lexp`, but `forall T: Type, lexp T`. Hence, for any type `T`, the expression `top T` is of type `lexp T` (*e.g.*, `top nat: lexp nat`). Clearly, the definition of linear logic formulas must be independent of `T`. In particular, it should not allow pattern matching on terms of type `T`. Therefore, the type `Lexp` of linear logic formulas is defined over all types `T` (*i.e.*, it is a dependent type). The same holds for atoms and terms:

```
Definition Term := forall T: Type, term T.    (* type for terms *)
Definition AProp := forall T: Type, aprop T. (* type for atomic propositions *)
Definition Lexp := forall T: Type, lexp T.    (* Type for formulas *)
```

Hence, connectives must be functions on `T`, e.g.,

```
Definition Top: Lexp := fun T ⇒ top. (* formula ⊤*)
Definition Atom (P: AProp): Lexp := fun T ⇒ atom (P T). (* building atomic propositions *)
Definition Tensor (F G: Lexp): Lexp := fun T ⇒ tensor (F T) (G T). (* formula F ⊗ G *)
```

Since `T` is never destructed, all its occurrences in the above code can be replaced by "`_`" (meaning "*irrelevant*"). This means that an arbitrary type `T` is passed as a parameter, and it does not interfere with the structure of formulas.

Some of the forthcoming proofs proceed by structural induction on a LL formula `F: Lexp`. However, since `Lexp` is a (polymorphic) function, not an inductive type, Coq's usual `destruct`, `induction` or `inversion` tactics do not work. Following [4], the solution is to define inductively the notion of *closed formulas* as follows:

```
Inductive ClosedT: Term → Prop :=
  | cl_cte: forall C, ClosedT (Cte C)
  | cl_fc1: forall n t1, ClosedT t1 → ClosedT (FC1 n t1)
  | cl_fc2: forall n t1 t2, ClosedT t1 → ClosedT t2 → ClosedT (FC2 n t1 t2).

Inductive ClosedA : AProp → Prop :=
  | cl_a0: forall n, ClosedA (A0 n)
  | cl_a1: forall n t, ClosedT t → ClosedA (fun _ ⇒ a1 n (t _)).
  | cl_a2: forall n t t',  ClosedT t → ClosedT t' → ClosedA (fun _ ⇒ a2 n (t _) (t' _)).

Inductive Closed : Lexp → Prop :=
  | cl_atom: forall A, ClosedA A → Closed (Atom A )
  | cl_perp: forall A, ClosedA A → Closed (Perp A )
  | cl_one: Closed One
  | cl_tensor: forall F G, Closed F → Closed G → Closed (Tensor F G)
  | cl_fx: forall FX, Closed (Fx FX)
  [...]
```

Such definitions rule out the occurrences of the *open* term `var x` in the type `Term` and, consequently, in atomic propositions and formulas. Now we need the axioms that only *closed* structures can be built:

```
Axiom ax_closedT: forall X: Term, ClosedT X.
Axiom ax_closedA: forall A: AProp, ClosedA A.
Axiom ax_closed : forall F: Lexp, Closed F.
```

The statements above cannot be proved in Coq, mainly because it is not possible to induct on function types (such as `Term`). This would require, *e.g.*, a meta-model of the Calculus of Inductive Constructions [2] itself inside Coq. Let us

give some intuition of why those axioms are consistent with the theory of Coq. This will clarify the closeness condition we impose on `Lexp`. If 1 is the identifier for a predicate P and `c` a constant of type `A`, the formula $P(c)$ can be defined as `Definition Pc: Lexp := fun T: Type ⇒ atom (a1 1 (cte c)).` However, the same exercise does not work for $P(x)$ when x is a free variable. If we were to write `Definition Px: Lexp := fun T: Type ⇒ atom (a1 1 (var ??)).`, then "??" must be an inhabitant of `T`. Since we do not know anything about `T`, we cannot name an element of it and `Px` will never type check. Hence, both terms and formulas are necessarily closed (without free variable).

Another consequence of the functional representation of terms is that we require the axiom of Functional Extensionality of the standard library of Coq to check whether two terms/formulas are the same:

```
Axiom functional_extensionality_dep : forall {A} {B : A → Type},
  forall (f g : forall x : A, B x), (forall x, f x = g x) → f = g.
```

Roughly, given two function f and g, we conclude $f = g$ if $f(x) = g(x)$ for all x.

Substitutions. According to the definition of quantifiers, substitutions should be performed on formulas of type `T → lexp`, where `T` is a parameter. Following the same idea as before (and as in [4]), we define a type `Subs` for such formulas as a (closed) dependent type: `Definition Subs := forall T: Type, T → lexp T.` By quantifying over all `T`s, we prevent functions that destruct the term and change the structure of the formula.

Now we need to define a wrapper for substitutions. This substitution function will take as parameters `S: Subs` and `X: Term`, and return a `Lexp`. The first step is to apply Coq's β-reduction to `S`, the type of `X` (instantiating the `forall`) and `X` (first argument). The result of this reduction is of type `lexp (term T)`, which is different from `Lexp`, defined as `forall T: lexp T.` A `lexp T` is constructed by the function `flatten: lexp (term T) → lexp T`, which is defined in a section parametrized by `T`.

```
Definition Subst (S: Subs) (X: Term) : Lexp := fun T: Type  ⇒ flatten (S (term T) (X T)).
```

As an example, the steps below apply $\lambda x.\top \otimes Q(f(x), c)$, to the constant d. The resulting inhabitant of `Lexp` is commented out.

```
Definition S: Subs := fun (T: Type) (x: T) ⇒ tensor top (atom (a2 1 (fc1 1 (var x)) (cte c))).
Definition t1: Term := fun T ⇒ (cte d).
Eval compute in Subst S t1.
(* Result: fun T:Type ⇒ tensor one (atom (a2 4 (fc1 1 (cte d)) (cte c))) *)
```

Formula complexity. Even if it is now possible to reason on the structure of the formula via the `Closed` definition, some of our proofs proceed by induction on the weight of a formula. Our definition follows the standard one (*i.e.*, $W(\top) = 0, W(F \otimes G) = 1 + W(F) + W(G)$, etc). It should be the case that $W(F[t/x]) = W(F[t'/x])$ for any two terms t and t'. This is true since substitutions cannot perform "case analysis" on the term t to return a different formula. This simple fact requires extra work in Coq. First, we define when two formulas are equivalent modulo renaming of bound variables (some cases are omitted):

```
Inductive xVariantT: term T → term T'→ Prop :=
  | xvt_var: forall x y, xVariantT (var x) (var y)
```

$$\frac{}{\vdash a^{\perp}, a}\ I \qquad \frac{\vdash \Gamma_1, A \quad \vdash \Gamma_2, B}{\vdash \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{}{\vdash 1}\ 1 \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \parr B}\ \parr \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \bot}\ \bot \qquad \frac{\vdash \Gamma, A[x/e]}{\vdash \Gamma, \forall x.F}\ \forall \qquad \frac{\vdash \Gamma, A[x/t]}{\vdash \Gamma, \exists x.F}\ \forall_d$$

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B}\ \& \qquad \frac{}{\vdash \Gamma, \top}\ \top \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\ \oplus_1 \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\ \oplus_2 \qquad \frac{\vdash ?A_1, \ldots, ?A_n, A}{\vdash ?A_1, \ldots, ?A_n, !A}\ !$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\ ? \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\ W \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\ C$$

Fig. 1. Linear logic introduction rules: e is fresh, *i.e.*, does not appear in Γ.

```
  | xvt_cte: forall c, xVariantT (cte c) (cte c)
  [...]

Inductive xVariantA: aprop T → aprop T'→ Prop :=
  | xva_eq: forall n, xVariantA (a0 n) (a0 n)
  | xva_a1: forall n t t', xVariantT t t' → xVariantA (a1 n t) (a1 n t')
  [...]

Inductive EqualUptoAtoms: lexp T → lexp T' → Prop :=
  | eq_atom: forall A A', xVariantA A A' → EqualUptoAtoms (atom A) (atom A')
  | eq_ex: forall FX FX', (forall t t', EqualUptoAtoms (FX t) (FX' t')) →
                                        EqualUptoAtoms (ex (FX )) (ex (FX'))
  [...]
```

For similar arguments as the ones given above for *closeness*, we need to add as axioms that inhabitants of `Subs` and `Lexp` cannot make choices based on its arguments:

```
Axiom ax_subs_uptoAtoms: forall (T T': Type) (t: T) (t': T') (FX: Subs),
                                  EqualUptoAtoms (FX T t) (FX T' t').
Axiom ax_lexp_uptoAtoms: forall (T T': Type) (F: Lexp), EqualUptoAtoms (F T) (F T').
```

The needed results for the weight function (defined as `Exp_weight`) can thus be proved:

```
Theorem subs_weight: forall (FX: Subs) x y, Exp_weight(Subst FX x) = Exp_weight(Subst FX y).
```

We also formalized propositional linear logic, including all the theorems in this section as well as those in Sections 3 and 4. In the propositional case, the type of formulas is a standard inductive type and there is no need for parametric (polymorphic) definitions. Hence, none of the axioms above were needed. In the first-order case, removing those axioms will amount to, e.g., formalize in a finer level the binders and substitutions. This is definitely not an easy task (see e.g., [9,10]) and we would not get for free all the benefits inherited from the PHOAS approach (e.g., substitutions are, by definition, capture avoiding).

3 Sequent Calculi

The proof system for one-sided (classical) first-order linear logic is depicted in Figure 1. A sequent has the form $\vdash \Gamma$ where Γ is a multiset of formulas (*i.e.*, exchange is implicit). While this system is the one normally used in the literature, LL's focused proof system is equipped with some more structure. As shown by Andreoli [1], it is possible to incorporate contraction (C) and weakening (W) into the introduction rules. The key observation is that formulas of the form $?F$ can be contracted and

weakened. This means that such formulas can be treated as in classical logic, while the remaining formulas are treated linearly. This is reflected into the syntax in the so called *dyadic sequents* which have two contexts:

$$\frac{\vdash \Theta, F : \Gamma}{\vdash \Theta : \Gamma, ?F}\ ? \qquad \frac{\vdash \Theta, F : \Gamma, F}{\vdash \Theta, F : \Gamma}\ copy \qquad \frac{}{\vdash \Theta : a^{\perp}, a}\ I$$

$$\frac{\vdash \Theta : \Gamma_1, A \quad \vdash \Theta : \Gamma_2, B}{\vdash \Theta : \Gamma_1, \Gamma_2, A \otimes B}\ \otimes \qquad \frac{\vdash \Theta : \Gamma, A \quad \vdash \Theta : \Gamma, B}{\vdash \Theta : \Gamma, A \& B}\ \&$$

Here Θ is a set of formulas and Γ a multiset of formulas. The sequent $\vdash \Theta : \Gamma$ is interpreted as the linear logic sequent $\vdash ?\Theta, \Gamma$ where $?\Theta = \{?A \mid A \in \Theta\}$. It is then possible to define a proof system for LL without explicit weakening (implicit in rule I above) and contraction (implicit in e.g., *copy* and $\otimes$ above). Notice that only the linear context Γ is split among the premises in the $\otimes$ rule. The complete proof system can be found in [1].

Rules of the dyadic system are specified as inductively defined predicates. The following code is an excerpt of the definition of the dyadic system (called `sig2` in our files):

```
Inductive sig2: list Lexp → list Lexp → Prop :=
 | sig2_init : forall B L A, L =mul= (A ⁺) :: [A ⁻] →⊢ B ; L
 | sig2_bang: forall B F L , L =mul= [! F] →⊢ B ; [F] →⊢ B ; L
 | sig2_ex : forall B L FX M t, L =mul= E{FX} :: M →⊢ B ; (Subst FX t) :: M →⊢ B ; L
 | sig2_fx : forall B L FX M, L =mul= (F{FX}) :: M → (forall x, ⊢ B ; [Subst FX x] ++ M) →⊢ B ; L
[...]
```

Given an atomic proposition `A:Aprop`, `A⁺` and `A⁻` stand, respectively, for `Atom(A)` (A) and `Perp(A)` $(A^{\perp})$. Given a substitution `FX:Subs`, the LL quantifiers are represented as `E{FX}` and `F{FX}`. The rule for the universal quantifier relies on Coq's (dependent type constructor) `forall` that takes care of generating a fresh variable. Finally, ++ is list concatenation.

Given two multisets of formulas `M` and `N`, `M=mul=N` denotes that `M` is multiset equivalent to `N`. Coq's library `Coq.Sets.Multiset` defines multisets as bags (of type `A → nat`), thus specifying the number of occurrences for each element of a given type A. Reasoning about such bags is hard and automation becomes trickier. Using lists as representation of multisets seems to be a better (and cleaner) choice. In fact, the library CoLoR (`http://color.inria.fr/`) follows that direction. CoLoR is quite general and formalizes, among several other theorems, properties about data structures such as relations, finite sets, vectors, etc. CoLoR transforms the lists M and N to `Coq.Sets.Multiset` and uses the multiplicity definition of Coq in order to define multiset equivalence. Our `Multisets` module does not use this transformation. Instead, it uses directly `Coq.Lists` that features mechanisms to count the number of occurrences of a given element. We also added some useful theorems for our formalization and implemented automatic techniques (e.g., `solve_permutation`) that discharge most of the proofs about multisets we needed in our developments.

Inductive Measures. In our proofs, we usually require measures for the height of

the derivation as well as for the number (and complexity) of the cuts used. For this reason, we also specified other variants of the sequent rules where such measures are explicit. For instance, the proof of cut-elimination was performed on the following system:

```
Inductive sig3: nat → nat → list lexp → list lexp → Prop :=
  | sig3_init : forall (B L: list lexp) A, L =mul= (A ⁺) :: [A ⁻] → 0 ⊢ 0 ; B ; L
  | sig3_CUT : forall (B L: list lexp) n c w h, sig3_cut_general w h n c B L → S n ⊢ S c ; B ; L
  [...]

with sig3_cut_general : nat → nat → nat → nat → list lexp → list lexp → Prop :=
 | sig3_cut : forall ...   → m ⊢ c1;B ; F ::  M → n ⊢ c2 ; B ; F ::  N → sig3_cut_general ...
 | sig3_ccut : forall ...   → m ⊢ c1;B ; (! F) :: M → n ⊢ c2 ; F°:: B ; N → sig3_cut_general ...
```

Note that there are two cut-rules: Cut (`sig3_cut`) and $Cut!$ (`sig3_ccut`). The second rule is needed in the proof of cut-elimination as shown in Section 4.1. We later show that the system with only the Cut rule (`sig2`) is equivalent to `sig3`. Sequents in `sig3` take the form $n \vdash c; B; L$ where n is the height of the derivation, c the number of times the cut-rule was used and B and L the classical and linear contexts respectively. Definition `sig3_cut_general` makes also explicit the following measures : w (the complexity of the cut formula) and $h = m + n$ (the cut-height, including the height of the two premises of the cut-rule). Such measures will be useful in Section 4.1.

3.1 Focused System

Focusing was first proposed by Andreoli [1] as a discipline on proofs in order to reduce non-determinism. Proofs are organized in two alternating phases: the negative phase contains only invertible rules, and the positive phase contains only non-invertible rules. The connectives $\parr, \bot, \&, \top, ?, \forall$ have invertible introduction rules and are thus classified as *negative*. The remaining connectives $\otimes, 1, \oplus, !, \exists$ are *positive*. Formulas inherit their polarity according to their main connective, *e.g.*, $A \otimes B$ is positive and $A \parr B$ is negative.

In LL's focused proof system LLF (also called triadic system), there are two types of sequents where Θ is a set of formulas, Γ a multiset of formulas, and L a list of formulas:

- $\vdash \Theta : \Gamma \Uparrow L$ belongs to the negative phase. During this phase, all negative formulas in L are introduced and all positive formulas and atoms are moved to Γ.
- $\vdash \Theta : \Gamma \Downarrow A$ belongs to the positive phase. During this phase, all positive connectives at the root of A are introduced.

Let us present some rules.

$$\frac{\vdash \Theta : \Gamma \Uparrow A, L \quad \vdash \Theta : \Gamma \Uparrow B, L}{\vdash \Theta : \Gamma \Uparrow A \& B, L} \qquad \frac{\vdash \Theta : \Gamma_1 \Downarrow A \quad \vdash \Theta : \Gamma_2 \Downarrow B}{\vdash \Theta : \Gamma_1, \Gamma_2 \Downarrow A \otimes B}$$

$$\frac{\vdash \Theta : \Gamma \Downarrow P}{\vdash \Theta : \Gamma, P \Uparrow} D_1 \qquad \frac{\vdash \Theta, P : \Gamma \Downarrow P}{\vdash \Theta, P : \Gamma \Uparrow} D_2 \qquad \frac{\vdash \Theta : \Gamma \Uparrow N}{\vdash \Theta : \Gamma \Downarrow N} R$$

Notice that focusing ($\Downarrow$) persists on the premises of the $\otimes$ rule. The negative phase ends when the list of formulas L is empty. Then, the decision rules D_1 (linear) and D_2 (classical) are used to start a new positive phase. Finally, the release rule switches to a negative phase when the current focused formula is negative. This restriction on proofs has two main applications: it considerably reduces proof search space and it allows specifiers to engineer proofs as we illustrate in Section 5.

The rules of the focused system (`TriSystem`) were formalized as the previous ones. We used `⊢B ; M ; UP L` to denote the negative phase and `⊢B ; M ;DW F` for a positive phase focused on F. Similar to the (unfocus) sequent systems `sig2`, we also defined for the triadic system a version with explicit height of derivation (`TriSystemh`) that we later show to be equivalent.

An interesting feature of `TriSysem` is that we can define automatic tactics to handle the negative phase. For instance the formula $p^- \parr q^- \parr \bot \parr (?p^+) \parr (?q^+)$ is proved as follows:

```
Example sequent: ⊢ [] ; [] ; UP( [ (p⁻&q⁻) ⅋ ⊥ ⅋ ?p⁺ ⅋ ?q⁺ ] ).
Proof with unfold p;unfold q;InvTac.
  NegPhase. (* Negative phase *)
  eapply tri_dec2 with (F:= p⁺) ... (* apply the decision rule on the classical context *)
  eapply tri_dec2 with (F:= q⁺) ...
Qed.
```

We first decompose all the negative connectives and store the atoms (`NegPhase`). Then, we have to prove two sequents (due to the & rule). For proving those sequents, we only need to decide to focus on the formulas p^+ and q^+ respectively. The "..." notation in Coq applies the tactic `InvTac` that solves all the needed intermediary results (e.g., checking the polarities of the atoms and applying the initial rules when needed).

3.2 Structural Properties

Using strong induction on the height of the derivation, we show several structural properties for the above systems. For instance, we proved that equivalent multisets prove the same formulas (preserving the height of the derivation):

```
Theorem sig2h_exchange: B1 =mul= B2 → L1 =mul= L2 → n ⊢ B1; L1 → n ⊢ B2; L2.
```

Moreover, using the library `Morphisms`, we are able to easily substitute equivalent multisets during proofs (using the tactic `rewrite`). Moreover, we proved height preserved weakening and contraction for the classical context:

```
Theorem weakening_sig2h: n ⊢ B; L → n ⊢ B ++ D; L.

Theorem contraction_sig2h: n ⊢ F :: F :: B; L → n ⊢ F :: B; L.
```

Similar properties were proved for the triadic system. The only interesting case is exchange for the focused context (needed in the proof of completeness):

```
Theorem EquivUpArrow : n ⊢ B ; M ; UP L → L =mul= L' → exists m, m ⊢ B ; M ; UP L'.
```

In this case, the height of the derivation is not preserved since the last context is a list and not a multiset. The proof of such theorem required some lemmata showing the invertibility of the negative connectives. In particular, in a sequent

$\vdash$`B ; M ; UP L ++ [a] ++ L'`, during the negative phase it is possible to work on the formula a any time during the proof. Those results correspond to the following theorems (all the variables are universally quantified):

```
Theorem EquivAuxTop : ⊢ B ; M ; UP (L++[⊤]++L').

Theorem EquivAuxBot : ⊢ B ; M ; UP (L++L') → ⊢ B ; M ; UP (L++[⊥]++L').

Theorem EquivAuxWith : ⊢ B ; M ; UP (L++[F]++L') → ⊢ B ;M ; UP (L++[G]++L') →
                                                    ⊢ B; M ;UP (L++[F&G]++L').

Theorem EquivAuxPar : n ⊢ B ; M ; UP (L++[F ; G]++L') → ⊢ B ; M ; UP (L++[F ⅋ G]++L').

Theorem EquivAuxSync : ~ Asynchronous F → ⊢ B ; M++[F] ; UP (L++L') → ⊢ B ; M ; UP (L++[F]++L').

Theorem EquivAuxForAll : (forall x, ⊢ B ; M; UP (L++[Subst FX x]++L')) →
                                    ⊢ B ; M ; UP (L++[F{FX}]++L').

Theorem EquivAuxQuest : n ⊢ B++[F] ; M ; UP (L++L') → ⊢ B ; M ; UP (L++[? F]++L').
```

The proofs of these lemmas proceed by induction on the sum of the complexity of the formulas in L (i.e., summing up the complexities of the formulas in L). The `Asynchronous F` predicate asserts that F is a negative formula.

4 Meta-Theory

This section presents the main results formalized in our system: cut-elimination (for the dyadic system) and completeness of the focused system. Hence, as a corollary, we show the equivalence of all these systems (dyadic, triadic, with/without cut rules and with/without measures) and prove the consistency of LL. We show the key cases and relevant strategies to complete the proofs of the main theorems. For that, we shall use the following notation. We start listing, using "H" ids, the relevant hypotheses of the theorem and the current Goal (using "G" ids). Then, we write the relevant steps (using Coq's comments) to generate new hypotheses or transform the current goal. For instance:

```
HI : forall m <=n, m ⊢ B ; M ; UP L → exists x, x ⊢ B ; M ++ L (* inductive hypothesis *)
H1: n ⊢ B ; M ; UP L
G:  ⊢ B ; M ++ L (* current goal *)
(* apply HI in H1 to conclude H2 *)          H2 : exists x, x ⊢ B ; M ++ L
(* using inversion in H2 we show H2' *)      H2' : x ⊢ B ; M ++ L
(* conclude G by using adequacy (with/without measures) in H2' *)
```

In proofs involving the triadic system (Section 4.2), the focusing discipline determines easily the next step/goal in the proof. In those cases, we do not use comments to explain the proof but we use directly Coq's tactics. Roughly, those tactics correspond to one application of a logical rule of the triadic system (`TriSystem`). We note that some of the proofs in Section 4.1 correspond to standard sequent transformations that can be found, e.g., in [25].

4.1 Cut Elimination

The proof is structured as follows. The main theorem consists in showing that a proof with one cut can be replaced with a proof with zero cuts. Recall that the measure of the number of cuts is explicit (as c) in the system `sig3`:

```
Lemma cut_elimination_base: n ⊢ 1 ; B ; L → exists m, m ⊢ 0 ; B ; L.
```

This lemma represents the case where we are eliminating the upper-most cut in a derivation tree. The proof, proceeds by double induction on the complexity of the cut-formula (w) and the cut-height (h), i.e., the sum of the premises' heights of the cut-rule. The proof of this lemma requires several additional lemmas/cases:
(i) The base case corresponds to $w = h = 0$. The cut-formula is one of the units or an atomic proposition. Moreover, since $h = 0$, both premises are either the initial rule or $\top$. We grouped those cases in the following theorem.

```
Theorem cut_aux: L =mul= M1 ++ M2 → 0 ⊢ 0 ; B ; F ::  M1 → 0 ⊢ 0 ; B ; F° ::  M2 → exists m, m ⊢ 0 ; B ; L.
```

(ii) In the cases where the formula is not principal, one has to permute the cut and reduce the inductive measure h. For instance, the case where rule $\oplus$ is used in one of the cut-premises is proved as follows:

```
H :  n1 ⊢ 0; B; a ::  F ::  T (* 0 cuts, height n1. "a" is the cut-formula *)
Hn1 : S n1 ⊢ 0; B; a ::  F ⊕ G ::  T
Hn2 : n2 ⊢ 0; B; a° ::  M2  (* 0 cuts, height n2 *)
HI : forall h<= n1 + n2 → inductive hypothesis on the cut−height
G: exists m : nat, m ⊢ 0; B; F ⊕ G ::  T ++ M2
(* apply cut on H and Hn2 to conclude Hc*)              Hc: S(max n1 n2) ⊢ 1 ; B; F ::  T ++ M2
(* use HI to produce a cut-free proof Hc' *)            Hc': x ⊢ 0 ; B; F ::  T ++ M2
(* using exists with t:= S x the new goal is G' *)      G' : S x  ⊢ 0; B; F ⊕ G ::  T++ M2
(*  conclude G' from Hc' and the rule ⊕ *)
```

(iii) When the cut formula is principal in both premises, we perform (possible several) cuts with simpler formulas. For instance, the case of $\otimes$ is:

```
H1 : S (max n m) ⊢ 0; B; F ⊗ G ::  (M1 ++ M2) (* cut formula is F * G *)
H2 : S n0 ⊢ 0; B; F° ⅋ G° ::  D
H3 : m ⊢ 0; B; F ::  M1
H4 : n ⊢ 0; B; G ::  M2
H5 : n0 ⊢ 0; B; G° ::  F° ::  D
HI : forall w<= w(F ⊗ G) → inductive hypothesis on the weight
G: exists m0 : nat, m0 |− 0; B; M1 ++ M2 ++ D
(* apply cut on H4 and H5 to conclude H6 *)             H6 : S(max n n0) ⊢ 1 ; B; F° ::  (M2 ++ D)
(* use HI on H6 to produce a cut-free proof H6' *)      H6' : x ⊢ 0 ; B; F° ::  (M2 ++ D)
(* apply cut on H3 and H6 to conclude H7 *)             H7 : S(max m x) ⊢ 1 ; M1 + + M2 + + D
(* use HI on H7 to produce a cut-free proof H7' *)      H7' : y ⊢ 0 ; M1 + + M2 + + D
(* conclude G from H7' *)
```

The case when ! is the cut-formula and principal requires and additional rule $Cut!$

$$\frac{\vdash \Theta : \Delta, !F \quad \vdash \Theta, F^{\perp} : \Gamma}{\vdash \Theta : \Delta, \Gamma}$$

that we show to be admissible (Theorem `sig2_iff_sig3` below). This rule is encoded in the `sig3` system (constructor `sig3_ccut`). We then transform an application of Cut into an application of $Cut!$, reducing the cut-height:

```
H1 : S n |− 0; B; [!  F]
H2 : S n0 |− 0; B; ? F° ::  L
H3 : n |− 0; B; [F]
H4 : n0 |− 0; F° ::  B; L
HI : forall h <= S (n + n0) → inductive hypothesis on cut−height
G:  exists m0 : nat, m0 ⊢ 0; B; L
(* apply ccut on H1 and H4 to conclude H5*)      H5 : S(max (S n) n0) ⊢ 1; B ; L
(* use HI to obtain the cut-free proof H5' *)    H5' : x ⊢ 0; B ; L
(* conclude G from H5' *)
```

(iv) The cases for eliminating an application of $Cut!$ are similar.

Using all the lemmas above, the proof of cut-elimination considers all the cases (including the symmetric ones) generated by Coq. The final step is to show that a proof with an arbitrary number of cuts can be transformed into a proof without cuts. This can be easily done by induction on the number of cuts and using the previous results:

```
Theorem cut_elimination : forall B L n c, n |− c ; B ; L → exists m, m |− 0 ; B ; L.
```

As a corollary, we can show the consistency of LL:

```
Theorem consistency : ∼ sig3 n c [] []    ∧    ∼ sig3 n c [] [0]    ∧ ∼ sig3 n c [] [⊥].
```

Now we can prove that the $Cut!$ rule is admissible and hence, the systems with (`sig3`) and without (`sig2`) this rule are equivalent:

```
Theorem sig2_iff_sig3: sig2 B; L ↔ sig3 B; L.
```

4.2 Completeness of Focusing

The following theorem shows that focused proofs can be mimicked by the dyadic system:

```
Theorem Soundness : LexpPos M → n |−F− B ; M ; A → |−− B ; M ++ (Arrow2LL A).
```

where the predicate `LexpPos` states that all the formulas in the list/multiset M are all positive and the function `Arrow2LL` simply transforms "$\Uparrow L$" into L and "$\Downarrow F$" into the list $[F]$. The proof is easy by induction on the height of the derivation n: we just need to apply exactly the same rule used in the focused proof.

The proof of the inverse theorem, i.e., completeness, is of course more involved. First, we proved the invertibility theorems in Section 3.1 (for the negative phase). Then we proved that applications of positive rules can be switched:

```
Theorem InvCopy : |−F− B ++ [F] ;M ; UP (F :: L) → LexpPos M → |−F− B ++ [F] ; M ; UP L .

Theorem InvEx : |−F− B ;M ; UP (Subst FX t :: L) → LexpPos M → |−F− B ; M ++ [E{ FX}] ; UP L .

Theorem InvPlus : |−F− B ;M ; UP (F :: L) → LexpPos M → |−F− B ; M ++ [F ⊕ G] ; UP L .

Theorem InvTensor : LexpPos (M ++ M') → |−F− B ;M ; UP (F :: L) → |−F− B; M'; UP (G :: L') →
                    |−F− B ; M ++ M' ++ [F ⊗ G] ; UP (L ++ L') .
```

In [1] Andreoli detailed the proof of the case for $\otimes$ (`InvTensor`). Let us explain the case for $\oplus$. First we define the following predicates:

```
Definition RUp (n:nat) := forall B L M F G, LexpPos M →
      n |−F− B ;M; UP (L ++ [F]) → |−F− B ; M ++ [F ⊕ G]; UP L.
Definition RDown (n:nat) := forall B M H F G, LexpPos M → PosOrNegAtom F →
      n |−F− B ; M ++ [F] ; DW H → |−F− B ; M ++ [F ⊕ G] ; DW H.
Definition RInd (n:nat) := RUp n ∧ RDown (n −1).
```

The predicate `RUp` determines how $\oplus$ permutes with the negative connectives. We proceed by induction on n. In the inductive cases, we consider two cases, namely, when L is empty or not. Let us explain the first case. We consider the last rule applied. The cases of the negative connectives are easy, e.g., the case $\bot$ is as follows:

```
Hyp1:   S n |−F− B; M1; UP [ ⊥]
G1: |−F− B; M1  ++ [⊥ ⊕ G];  UP [ ]    eapply tri_dec1 with (F:= ⊥ ⊕ G) ...
G2: |−F− B; M1 ++ [];  DW (⊥ ⊕ G)     eapply tri_plus1 ...
G3: |−F− B; M1 ++ [];  DW ⊥           eapply tri_rel ...
G4: |−F− B; M1 (2++·) []; UP⊥        (* conclude by using AdequacyTri1 and Hyp1 *)
```

The last step (`AdequacyTri1`) uses the adequacy result relating the system with measures (sequent in `Hyp1`) and the current goal.

The interesting case is the store case, i.e., when the the formula F is an atom or a positive formula. Then we have the following situation:

```
HDown : RDown (n −1)
Hyp1 :  n |−F− B; M1 ++ [F];  UP [ ]
```

Since the sequent in `Hyp1` is provable, due to focusing, we know that it is provable by using a decision rule. We then need to consider three cases, namely, focusing on either `F`, on a formula in `M1` or on a formula in `B`. Let us consider the second case. We have the following situation (after some substitutions):

```
HDown : RDown (n')
Hyp1' :    n' |−F− B; M1' ++ [F];  DW F'
HML :   M1 =mul= F' :: M1'
G1: |−F− B; (F' ::  M1')  ++ [F ⊕ G];  UP []        eapply tri_dec1 with (F:=F0) ...
G2: |−F− B; M1' ++  [F ⊕ G];  DW F'
```

The proof of `G2` ends by applying the inductive hypotheses `HDown` in `Hyp1'`.

The second predicate (`RDown`) allows us to permute two positive phases in the proof. We proceed by induction on n and consider all the cases for H in `DW H`. Let us show the case when $H = \exists x.H'$. The main hypotheses and goal are:

```
HDown : RDown n'
H : n' |−F− B; M ++ [F];  DW (Subst FX t)
G1 : |−F− B; M ++ [F ⊕ G];  DW (E{FX})     apply tri_ex with (t:=t) ...
G2: |−F− B; M ++ [F ⊕ G];  DW (Subst FX t)
```

The proof ends by applying the inductive hypothesis `HDown` in `H`.

5 Applications

In [16], LL was used as the logical framework for specifying a number of logical and computational systems. The idea is to use two meta-level predicates $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ for identifying objects that appear on the left or on the right side of the sequents in the object logic. Hence, for instance, object-level sequents of the form $B_1, \ldots, B_n \vdash C_1, \ldots, C_m$ (where $n, m \geq 0$) are specified as the multiset $\lfloor B_1\rfloor, \ldots, \lfloor B_n\rfloor, \lceil C_1\rceil, \ldots, \lceil C_m\rceil$. Here, as an application of our developments, we specify an encoding of intuitionistic propositional logic (LJ) into LL and prove the adequacy of the encoding. The machinery we develop is general enough to mechanize the proof of other adequacy theorems for logical systems [16] and also for concurrent computational systems [18,19].

We first specify the syntax and the logical rules of LJ in the usual way:

```
Inductive LForm :Set :=
  | bot (* false *) | atom : nat → LForm (* atomic propositions *)
  | conj : LForm → LForm → LForm (* conjunction *)
```

```
  | disj : LForm → LForm → LForm (* disjunction *)
  | impl : LForm → LForm → LForm. (* intuitionistic implication *)
Inductive sq : list LForm → nat → LForm → Prop :=
  | init : forall (L L' :list LForm) a, L =mul= atom a :: L' → L ; 0 |−P− atom a
  | cR : forall L F G n m , L ; n |−P− F → L ; m |−P− G → L ; S (max n m) |−P− conj F G
  | cL : forall L G G' F L' n, L =mul= (conj G G') :: L' → G :: G' :: L' ; n |−P− F → L ; S n |−P− F
  [...]
```

We name the two meta-level predicates $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ as 1 and 3 respectively, and we also name the (meta-level) functional symbols that represent each of the connectives in LJ:

```
Definition rg := 1. (* UP predicate *)   Definition lf := 3. (* DOWN predicate *)
Definition bt := 0. (* bottom *)         Definition pr := 1. (* atoms / propositions *)
Definition cj := 2. (* conjunction *)    Definition dj := 3. (* disjunction *)
Definition im := 4. (* implication *)
```

We note that even (resp. odd) predicates are assumed to be positive (resp. negative). Hence, rg and lf are negative atoms and it is not possible to focus on them.

Now we are able to encode LJ's logical rules as bipoles [16]. Roughly, a bipole is a positive formula in which no positive connective can be in the scope of a negative one. Focusing on such a formula will produce a single positive and a single negative phase. This two-phase decomposition enables the adequate capturing of the application of an object-level inference rule by the meta-level logic. For instance, LJ initial rule is encoded as the following LL formula:

```
Definition INIT :Lexp := Ex (fun _ x ⇒ tensor
        (tensor  (perp (a1 rg (fc1 pr (var x))))  (perp (a1 lf (fc1 pr (var x)))))  top).
```

If we decide to focus on INIT, then, the rule $\exists$ requires to chose a value for x and the focus continues on the formula $(\lceil(A1(pr(x)))^{\perp}\rceil \otimes \lfloor(A1(pr(x)))^{\perp}\rfloor) \otimes \top$. Hence, the two atoms $\lceil(A1(pr(x)))\rceil$ and $\lfloor(A1(pr(x)))\rfloor$ must be already in the context (since focusing cannot be lost). The proof finishes by loosing the focusing on $\top$ and then, in the negative phase, by using the rule $\top$. In other words, in one change of polarity, we check that a given atom a is on the right and on the left, thus finishing the proof.

The other rules follow a similar pattern. For instance, $\wedge_R$ is specified as:

```
Definition CRIGHT :Lexp := Ex (fun _ x ⇒ ex( fun y ⇒
                    tensor (perp (a1 rg (fc2 cj (var x) (var y))))
                      ( withH (atom (a1 rg (var x)))  (atom (a1 rg (var y)))  ) )).
```

Again, in a positive phase, $\exists$ and $\otimes$ are applied and the atom $\lceil A1(cj(x,y))\rceil$ is consumed. Focusing is lost on $\&$ and we obtain two LL sequents. In each of them, the atoms $\lceil A1(x)\rceil$ and $\lceil A1(y)\rceil$ are stored. What we observe is exactly the behavior of $\wedge_R$.

Thanks to the automatic tactics developed, the proof of soundness and completeness are relatively simple. Let us explain the main arguments. In the case of soundness, we have:

```
Definition encodeSequent (L: list PL.LForm) (F: PL.LForm) :=
                        |−F− Theory ; (encodeFR F) :: encodeList L ; UP [].

Theorem Soundness: L |−P− n ; F → ( encodeSequent L F ).
```

where the function `encodeFR` (resp. `encodeList`) simply generate the needed $\lceil\cdot\rceil$ atoms (resp. list of $\lfloor\cdot\rfloor$ atoms). Moreover, `Theory` is a list with the encoding of all the LJ rules.

We proceed by induction on n. Let us consider the case of conjunction right. It suffices to build the focused proof, starting by focusing on the rule `CRIGHT`:

```
HI: (* Inductive hypothesis  x ≤ max(n,m)  *)
HF: L |−P− n; F
HG: L |−P− m; G
G1: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; UP []                  eapply tri_dec2 with (F:= CRIGHT) ...
G2: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; DW CRIGHT              eapply tri_ex with (t:= encodeTerm F).
G3: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; DW E{ ...}  ...        eapply tri_ex with (t:= encodeTerm G).
G4: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; DW  AtR ⊗ (AtF&AtG)    eapply tri_tensor ...
G5: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; DW  AtF&AtG            apply tri_rel ...
G6: |−F− Theory ; ⌈cj(F,G)⌉ ::encodeList L; UP  AtF&AtG            apply tri_with; apply tri_store ...
(* Branch F *) apply HI in HF ...
(* Branch G *) apply HI in HG ...
```

Recall that the "..." solves the easy cases (see Section 3.1). On the other side, completeness is stated as follows:

```
Theorem Completeness : ( encodeSequent L F ) → exists n, L |−P− n ; F.
```

We proceed by induction on the height of the (LL) derivation. The interesting part of this proof is that, thanks to focusing, we only need to use the `inversion` tactic on the hypotheses until the goal is proved. For instance, in the inductive case ($n > 0$) the most relevant hypothesis and goal are:

```
H: S n |−F− Theory; encodeFR F :: encodeList L; UP []
G: exists n : nat, L |−P− n; F
```

Focusing tells us that the proof in H must proceed by deciding either to focus on a formula in the classical context (`Theory`) or in the linear context (`encodeFR F :: encodeList L`). However, atoms in the linear context are negative and we cannot focus on them. Therefore, the only alternative is to focus on one of the formulas in `Theory`. This makes the proof rather simple. For instance, consider the case `INIT`:

```
H' : n |−F− Theory; encodeFR F :: encodeList L; DW INIT
```

We continue applying inversion on `H'` to "generate" the consequences of having the proof `H'`. The main consequence, in this case, is

```
H'' : exists L' a,  L =mul= (F :: L') ∧ F = PL.atom a
```

This means that F is necessarily an atom and such atom is in the list of (PL) formulas L. Hence, the proof concludes easily by applying the initial rule of LJ.

6 Related and Future Work

Intuitionistic propositional linear logic was implemented in Coq [22] and Isabelle [14], but the main goal of these works was to provide proof search, and thus no meta-theorems were proved. Cut-elimination and invertibility lemmas were proved for a formalization of several linear logic calculi in Abella [3]. Even though the paper presents only the propositional part, the proofs for first-order and focused fragments

were later completed. The first-order implementation requires the use of the two-level logic approach, particular to this tool. The reader may also refer to YALLA (https://perso.ens-lyon.fr/olivier.laurent/yalla/), and embedding of propositional LL in Coq.

A generic method for formalizing sequent calculi in Isabelle/HOL is proposed in [5]. The meta-theorems are parametrized by the set of rules and for cut-elimination, weakening must be admissible. The authors have applied the method to provability logics.

Completeness of focusing was proved for intuitionistic propositional logic in Twelf and Agda [24]. The proof follows the technique developed in [20] where sequents are annotated with terms and the problem is reduced to type-checking. Focusing follows as a corollary of two other properties proved about the calculus. In their approach, context management is handled in the meta-level, so much of the bureaucracy in handling multisets is avoided. The method in [24] could be used to prove completeness of focusing of LL as well. A possible candidate implementation is the one from [23, Chapter 6] on a modified version of Twelf. Unfortunately this development was never carried out. Another approach for proving completeness of focusing, formalized in Coq, uses an algebraic implementation of the calculus [13]. To use this solution, the calculus must have "dual" rules for all connectives (harmony) and admissibility of contraction and weakening.

In the short term, we plan to use the machinery in Section 5 in order to mechanize the proofs of adequacy of other formalisms into Linear Logic, e.g., Hybrid Linear Logic [7], and Concurrent Constraint Process Calculi (CCP) [19]. In order to prove the adequacy of CCP calculi featuring modalities, we have to specify also the so-called subexponentials [19] (roughly, exponentials decorated with indexes). Finally, a more interesting outcome will be formalizing the theorems in [16]. This should allow us to use the meta-theory of linear logic to prove meta-theorems (e.g., cut-elimination) of other logics encoded into LL.

References

[1] J.-M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 1992.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, EATCS Series, 2004.

[3] K. Chaudhuri, L. Lima, and G. Reis. Formalized meta-theory of sequent calculi for substructural logics. *Electr. Notes Theor. Comput. Sci.*, 332:57–73, 2017.

[4] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP*, 2008.

[5] J. E. Dawson and 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.

[6] J. Despeyroux, A. P. Felty, and A. Hirschowitz. Higher-Order Abstract Syntax in Coq. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA*, 1995.

[7] J. Despeyroux, C. Olarte, and E. Pimentel. Hybrid and Subexponential Linear Logics. *To appear in Electronic Notes in Theoretical Computer Science*, 2017.

[8] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic (CSL'12)*, 2012.

[9] A. P. Felty and A. Momigliano. Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax. *Journal of Automated Reasoning*, 2012.

[10] A. P. Felty, A. Momigliano, and B. Pientka. The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey. *Journal of Automated Reasoning*, 2015.

[11] G. Gentzen. Investigations into Logical Deductions. In *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

[12] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 1987.

[13] S. Graham-Lengrand. *Polarities & Focussing: a journey from Realisability to Automated Reasoning*. Habilitation thesis, Université Paris-Sud, 2014.

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

[15] D. Miller and C. Palamidessi. Foundational Aspects of Syntax. *ACM Computing Surveys*, 1999.

[16] D. Miller and E. Pimentel. A formal framework for specifying sequent calculus proof systems. *Theoretical Computer Science*, 2013.

[17] V. Nigam and D. Miller. A Framework for Proof Systems. *Journal of Automated Reasoning*, 2010.

[18] V. Nigam, C. Olarte, and E. Pimentel. A General Proof System for Modalities in Concurrent Constraint Programming. In *Concurrency Theory (CONCUR)*, 2013.

[19] C. Olarte, E. Pimentel, and V. Nigam. Subexponential concurrent constraint programming. *Theoretical Computer Science*, 2015.

[20] F. Pfenning. Structural Cut Elimination. *Information and Computation*, 2000.

[21] F. Pfenning and C. Elliott. Higher-order Abstract Syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[22] J. Power and C. Webster. Working with linear logic in coq. In *12th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16, 1999.

[23] J. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

[24] R. J. Simmons. Structural focalization. *CoRR*, abs/1109.6273, 2011.

[25] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2000.