École Normale Supérieure de Lyon

M2 : Informatique Fondamentale

Internship

# Formalized proofs of soundness and completeness of a hypersequent calculus for lattice-ordered Abelian groups

*Author:*
Christophe Lucas

*Supervisor:*
Dr. Matteo Mio

February 2018 - June 2018

## Contents

## 1 Introduction

In the last decades, many applications for mathematical logic have been found, notably regarding the formal verification of hardware and software systems: logics are used to express and verify properties of programs. By using tools for the verification of properties of complex systems based on logical methods, it is possible to find and correct bugs which could otherwise be very costly or even be dangerous to human lives. As a concrete example, in [LFZdG13] the authors used logical methods to verify some correctness properties of the software regulating air-traffic.

Among these logics, probabilistic logics are specialized for expressing properties of systems using probabilistic features such as random bit generation and randomization. Since those features are essential in many field - including cryptography, communication protocols, neural networks and machine learning - some recent work has focused on those probabilistic logics. One approach to the design of probabilistic logic is based on quantitative semantics. With this approach, a formula $F$ is not evaluated to a Boolean value - *true* or $false$ - but to a real number expressing some quantitative property of the program. For instance, in these logics, it is possible to write a formula whose semantic is the probability (a real number between 0 and 1) of reaching a terminal state.

The quantitative logic of [MFM17], in particular, has connectives corresponding to the basic operations on real numbers of sum, max and min. The structure $(\mathbb{R}, +, 0, \max, \min)$ is well known in mathematics: it is a lattice-ordered Abelian group because it is both an Abelian group $(\mathbb{R}, +, 0, -)$ and a lattice $(\mathbb{R}, \max, \min)$.

Abelian logic is the equational theory of lattice-ordered Abelian groups: from the basic axioms of lattice-ordered Abelian groups (see Figure 4) it is possible to derive, using the standard inference rules of equational logics (i.e., substituting equals for equals, see Figure 3 for details) other valid equations. For example, $- - 0 = 0$ is not an axiom but can be derived from the axioms. Finding proofs of equalities in Abelian logic is difficult because the crucial transitivity rule of equational logic (see Figure 3 ) requires to correctly guess the right interpolant $t_2$ among infinitely many possible candidates.

For reducing the complexity associated with the transitivity rule to the bare minimum, other better behaved deductive systems can be designed. In the context of Boolean logic, the equational theory of Boolean algebras, a well-behaved system called Sequent calculus has been designed by Gentzen in [Gen35]:

**Axiomatic rules:**

$$\frac{}{\bot \vdash}\ \bot_l \qquad \frac{}{\vdash \top}\ \top_r \qquad \frac{}{A \vdash A}\ \text{ax}$$

**Structural rules:**

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{aff}_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\ \text{aff}_r$$

$$\frac{\Gamma,\ A,\ A \vdash \Delta}{\Gamma,\ A \vdash \Delta}\ \text{contr}_l \qquad \frac{\Gamma \vdash A,\ A,\ \Delta}{\Gamma \vdash A,\ \Delta}\ \text{contr}_r$$

**Logical rules:**

$$\frac{\Gamma,\ A,\ B \vdash \Delta}{\Gamma,\ A \wedge B \vdash \Delta}\ \wedge_l \qquad \frac{\Gamma \vdash A,\ \Delta \quad \Gamma \vdash B,\ \Delta}{\Gamma \vdash A \wedge B,\ \Delta}\ \wedge_r$$

$$\frac{\Gamma,\ A \vdash \Delta \quad \Gamma,\ B \vdash \Delta}{\Gamma,\ A \vee B \vdash \Delta}\ \vee_l \qquad \frac{\Gamma \vdash A,\ B,\ \Delta}{\Gamma \vdash A \vee B,\ \Delta}\ \vee_r$$

$$\frac{\Gamma \vdash A,\ \Delta \quad \Gamma,\ B \vdash \Delta}{\Gamma,\ A \Rightarrow B \vdash \Delta}\ \Rightarrow_l \qquad \frac{\Gamma,\ A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\ \Rightarrow_r$$

$$\frac{\Gamma \vdash A,\ \Delta}{\Gamma,\ \neg A \vdash \Delta}\ \neg_l \qquad \frac{\Gamma,\ A \vdash \Delta}{\Gamma \vdash \neg A,\ \Delta}\ \neg_r$$

**Cut-rule:**

$$\frac{\Gamma \vdash A,\ \Delta \quad \Gamma,\ A \vdash \Delta}{\Gamma \vdash \Delta}\ \text{cut}$$

Figure 1: Inference rules of Gentzen's sequent calculus

In the sequent calculus, the critical rule is the cut-rule which, as the transitivity rule in equational logic, requires to guess a formula $A$. Note indeed that all other rules do not require any such guess as all the formulas in the premises already appear in the conclusion. The main theorem of Gentzen about the sequent calculus is called *cut-elimination* and states that if a proof of a sequent exists then a proof that does not use the cut-rule exists [Gen35]. Thanks to this theorem, Gentzen's sequent calculus is easier to manipulate than equational logic.

For Abelian logic, one interesting system is the hypersequent calculus GA introduced by George Metcalfe, Nicola Olivetti and Dov Gabbay in [MOG05]. The system is equivalent to the Abelian logic in the same way Gentzen's sequent calculus is equivalent to Boolean logic. And also for GA, the authors proved that the critical cut-rule can be eliminated.

The work carried out in this internship consists of the formalization, in the proof assistant Agda, of the main definitions and key results about the system GA from [MOG05]. The benefits of having a computer-verified formalization of the results regarding GA include:

- Proofs regarding deductive systems, and GA in particular, can be very tedious as they involve long arguments by induction having a lot of case analyses. It is therefore easy to make mistakes. Our formalization is computer-verified and therefore does not have mistakes.
- In Agda, every proof corresponds to a computable function. This means, for example, that from the proof of cut-elimination we obtain an algorithm that produces derivations without cuts.
- this formalization can be used as a basis to be extended to other logics with more operators and axioms, such as the probabilistic logic of [MFM17].

This report is structured as follow. In section 2, we detailed the structure of the GitHub repository where all the Agda code (more than 16000 lines) produced during this work is freely available. In section 3, we introduced the proof assistant Agda and the reason why we chose this proof assistant for our formalization. Then, in section 4, we present the basic definitions regarding equational logic and Abelian logic. In section 5, we define the GA system introduced in [MOG05]. Finally, in section 6 we explain how we proved the soundness and the completeness of this system. Some conclusions and directions of future work are presented in section 7.

## 2 Structure of the GitHub repository

The formalization can be found on the git depository `https://github.com/clucas26e4/formalisedGA`.
In total, there are 16663 lines of code.
It is structured as follow:

- module `Nat`: formalization of some useful properties on natural numbers. The natural numbers are defined in the standard library of Agda.
- module `Int`: formalization of properties on integers useful to prove that $\mathbb{Z}$ is a model of Abelian logic. The integers are defined in the standard library of Agda.
- module `Equality`: structural equality as defined in the standard library.
- module `Print`: formalization of some functions to generate latex code to print some of our objects, like preproofs.
- folder `Semantic`: in this folder are collected all objects related to lattice-ordered Abelian groups and Abelian logic.
  - module `Semantic.SemEquality`: formalization of Abelian logic.
    - module `Semantic.SemEquality.Proprieties`: formalization of some basic properties on lattice-ordered Abelian groups useful to prove the soundness of the system.
  - module `Semantic.Interpretation`: formalization of the interpretation of a list of terms, of a sequent and of a hypersequent. In here are functions that take these objects and return a term.
  - module `Semantic.Model`: formalization of a valuation and proof that $\mathbb{Z}$ is a model of Abelian logic.
- folder `Syntax`: in this folder are collected all objects related to the GA system defined in [MOG05].
  - module `Syntax.Term`: formalization of a term.
  - module `Syntax.ListTerm`: formalization of a list of terms.
    - module `Syntax.ListTerm.Proprieties`: some useful properties on lists of terms.
    - module `Syntax.ListTerm.Canonic`: formalization of a “canonic” form for atomic lists of terms. Since variables are formalized by natural numbers and since a atomic list of terms only has variables, the canonical form of an atomic list of terms is where the variables are sorted by increasing order.
  - module `Syntax.Seq`: formalization of a sequent.
    - module `Syntax.Seq.Proprieties`: some useful properties on sequents.
  - module `Syntax.HSeq`: formalization of a hypersequent.
    - module `Syntax.HSeq.Proprieties`: some useful properties on hypersequents.

    * module `Syntax.HSeq.Complexity`: formalization of the complexity of hypersequents, which is a pair of natural numbers. Formalization of a well-founded order on the complexity. This well-founded order will be used to prove that the function that creates a preproof terminates.
    * module `Syntax.HSeq.Ordered`: formalization of a predicate on hypersequents. This predicat states that a hypersequent is sorted, meaning that if $G = H_1|...|H_n$ then the number of operators of $H_i$ is greater than the number of operators of $H_j$ for $j < i$.
    * module `Syntax.HSeq.LinearComb`: formalization of a linear combinaison of the sequents of a hypersequent. A linear combinaison of $\Gamma_1 \vdash \Delta_1|...|\Gamma_n \vdash \Delta_n$ is a $n$-uplet of natural numbers. The idea is to have $\lambda_i$ iterations of the $i$-th sequent.
  - module `Syntax.HSeqList`: formalization of a list of hypersequents.
  - module `Syntax.Proof`: formalization of a cut-free proof of a hypersequent.
    * module `Syntax.Proof.Soundness`: proofs that all rules of the GA system are sound.
    * module `Syntax.Proof.Invertibility`: proofs that all logical rules of the GA system are invertible.
    * module `Syntax.Proof.Completeness`: proofs useful for the completeness of the GA system. The property that states that if an atomic hypersequent is positive then it has the $\lambda$-property is admitted here.
  - module `Syntax.CutProof`: formalization of proof of a hypersequent (with the cut-rule).
    * module `Syntax.CutProof.Soundness`: proofs that all rules of the GA+cut system are sound.
    * module `Syntax.CutProof.Invertibility`: proofs that all logical rules of the GA+cut system are invertible.
    * module `Syntax.CutProof.Completeness`: proof that the GA+cut system is complete.
  - module `Syntax.Preproof`: formalization of a preproof, a proof that only uses logical and exchange rules and that can stop at any point.
    * module `Syntax.Preproof.Create`: formalization of the function that takes a hypersequent and use every possible logical rules until having a list of atomic hypersequents. Proofs of the correctness of this function.
  - module `Syntax.StructuralPreproof`: formalization of structural preproof, a proof that only uses structural and exchange rules and that can stop at any point.

## 3 Agda

Agda is a dependently typed functional programming language first implemented by Ulf Norell during his PhD thesis at Chalmers University of Technology [Nor07]. The language has a structure very similar to the popular functional programming language Haskell and usual programming constructs - data types (or inductive types), pattern matching, records, `let` expressions and modules.

The choice of Agda was also motivated by the experience the present author has accumulated in his previous internship. Its Haskell-like structure makes proving a proposition exactly like coding a function, which allows to immediately have some algorithms - for instance, proving the cut-elimination theorem immediately gives an algorithm that takes a proof using the cut-rule and returns a cut-free proof.

Agda's system is based on a theory very similar to Martin-Löf type theory [Nor07]: propositions are translated into types and a proof of such proposition is a term inhabiting this type. The cases for going from propositions to types that are most used in my work are:

- the implication: $A \Rightarrow B$ is translated into a function from the translation of $A$ to the translation of $B$, noted $A \rightarrow B$ in Agda.
- the universal quantifier: $\forall x \in X, P(x)$ is translated into a dependant function from the type associated with $X$ to the depend type associated with $P(x)$, noted $(x : X) \rightarrow P(x)$ in Agda.
- the existential quantifier: $\exists x \in X, P(x)$ is translated into a dependant pair which first element has type the translation of $X$ and the second element has the depend type $P(x)$, noted $\exists[\ x\ ]\ (P(x))$ in Agda.
- the negation: $\neg A$ is translated into a function from the translation of $A$ to the empty type $\bot$, noted $\neg\ A$ in Agda or $A \rightarrow \bot$.

Agda relies heavily on pattern matching, both for inductive reasoning (by pattern matching on a inductive type) and to eliminate absurd cases.
For instance, with an inductive type $\mathbb{N}$ defined by:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

the plus function can be defined inductively as such:

```
{- _+_ is a infix definition: -}
{- the _ are where the arguments must be put -}
_+_ : ℕ → ℕ → ℕ
zero + m = m
(suc n) + m = suc (n + m)
```

Also in Agda, we defined an equivalence relation (noted ≡) as:

```
data _≡_ : {A : Set} → A → A → Set where
  refl : {A : Set} → {a : A} → a ≡ a
```

So $A \equiv B$ if and only if $A$ and $B$ are the same object. One of the most important theorem regarding this relation is the congruence, meaning that if two objects are equal, they remain equal if we apply a function:

```
cong : {A B : Set} −> (f : A → B) −> {x y : A} −> x ≡ y −> f x ≡ f y
cong f refl = refl
```

Now we can prove that $n + 0 = n$ for all $n$. The proposition is $\forall n \in \mathbb{N}, n + zero \equiv n$ which is translated in $(n : \mathbb{N}) \rightarrow n + \mathsf{zero} \equiv n$. The term is defined by induction (so with pattern matching) by:

```
n+0=n : (n : ℕ) → n + zero ≡ n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

Also, we can defined the relation less or equal by:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : (n : ℕ) → zero ≤ n
  s≤s : (n m : ℕ) → n ≤ m → (suc n) ≤ (suc m)
```

and we can prove that $1 \leq 0$ implies false: we do a pattern matching on the term t of type (suc zero) ≤ zero, Agda seek a constructor that correspond to the type. It finds none, so it answers that this case is not possible by replacing t by ().

```
absurd : (suc zero) ≤ zero → ⊥
absurd ()
```

Everything needed to understand the Agda part of my work is now defined, so we will introduce equational logic (with its Agda definition) and the theory of lattice-ordered Abelian groups.

## 4 Equational Reasoning

### 4.1 Equational Logic

A deductive system can be divided into two parts: the syntax and the semantic. We will first talk about the semantic part and then about the syntax.

**Definition 4.1** (Signature)**.** A *signature* $\Sigma$ is a finite set of symbols, each associated with a natural number named arity. The symbols of arity 0 are called *constants*, the other symbols are called *functions* or *operators*. A symbol $f$ of arity $n$ is noted $f^n$

**Definition 4.2** ($\Sigma$-term)**.** Given an infinite set of variables $\mathcal{V}$, generally noted $a, b, c, ...,$ and a signature $\Sigma = \{f_i^{n_i}\}$, a $\Sigma$-*term* $t$ is defined inductively by:

$$t := a \mid f_i^{n_i}(t_1, ..., t_{n_i})$$

The set of $\Sigma$-terms is noted $\Sigma_{\text{term}}$.

*Example* 4.1. The signature for lattice-ordered Abelian groups is $\{0^0, +^2, -^1, \sqcap^2, \sqcup^2\}$.

**Definition 4.3** (Interpretation)**.** An *interpretation* $\mathcal{M}$ of a signature $\Sigma$ is a set $M$ where every constant $c \in \Sigma$ is associated to an element $c^{\mathcal{M}} \in M$ and every function $f^n \in \Sigma$ to a function $f^{\mathcal{M}} : M^n \rightarrow M$.

*Example* 4.2. The trivial interpretation with only one element is an interpretation of all signatures.

**Definition 4.4** (Valuation)**.** A *valuation* of a signature $\Sigma$ to an interpretation $\mathcal{M}$ is a function $v$ mapping every variable to a element of $M$ and extended to a homomorphism - meaning that for every constant $c \in \Sigma$, $v(c) = c^{\mathcal{M}}$ and that for every function $f^n \in \Sigma$ and every terms $t_1, ..., t_n$, $v(f(t_1, ..., t_n)) = f^{\mathcal{M}}(v(t_1), ..., v(t_n))$.

**Definition 4.5** (Valid equality)**.** Let $\Sigma$ be a signature and $\mathcal{M}$ an interpretation of $\Sigma$. Let $t_1, t_2$ be two $\Sigma$-term. A equation $t_1 = t_2$ is said *valid* in $\mathcal{M}$, noted $\mathcal{M} \vDash t_1 = t_2$, if for every valuation $v$, $v(t_1) = v(t_2)$.

**Definition 4.6** (Axiom)**.** An *axiom* is an equality between two terms.

*Example* 4.3. For instance, the axioms for the lattice-ordered Abelian groups are:

**Abelian group axioms**

$$A + (B + C) = (A + B) + C \qquad A + B = B + A$$
$$A + 0 = A \qquad A - A = 0$$

**Lattice ordered axioms**

$$A \sqcup (B \sqcup C) = (A \sqcup B) \sqcup C \qquad A \sqcup B = B \sqcup A \qquad A \sqcup (A \sqcap B) = A$$
$$A \sqcap (B \sqcap C) = (A \sqcap B) \sqcap C \qquad A \sqcap B = B \sqcap A \qquad A \sqcap (A \sqcup B) = A$$

**Compability axiom**

$$((A \sqcap B) + C) \sqcap (B + C) = (A \sqcap B) + C$$

Figure 2: Axioms of lattice-ordered Abelian groups

**Definition 4.7** (Model)**.** Let $\Sigma$ be a signature and $\mathbb{T}$ a set of axioms. A *model* $\mathcal{M}$ of $\mathbb{T}$ is an interpretation such that for every axiom $A = B \in \mathbb{T}$, $\mathcal{M} \vDash A = B$ holds.

*Example* 4.4. $(\mathbb{Z}, 0, +, \max, \min)$, $(\mathbb{Q}, 0, +, \max, \min)$, $(\mathbb{R}, 0, +, \max, \min)$ are models of lattice-ordered Abelian groups.

Now the syntax part of equational reasoning:

**Definition 4.8** (Context)**.** Given an infinite set of variables and a signature $\Sigma = \{f_i^{n_i}\}$, a *context* $C$ is defined inductively by:

$$C := \bullet \mid a \mid f_i^{n_i}(C_1, ..., C_{n_i})$$

**Definition 4.9** (Equational theory)**.** Let $\Sigma$ be a signature and $\mathbb{T}$ a set of axioms. The $\Sigma$-*equational theory* defined by $\mathbb{T}$ is given by the following inference rules: $\mathbb{T} \vdash t_1 = t_2$ is said *derivable* if there is a derivation tree where the conclusion is

**Equational rules:**

$$\frac{}{\mathbb{T} \vdash t_1 = t_2}\ \text{ax} - (t_1 = t_2) \in \mathbb{T}$$

$$\frac{}{\mathbb{T} \vdash t = t}\ \text{refl} \qquad \frac{\mathbb{T} \vdash t_2 = t_1}{\mathbb{T} \vdash t_1 = t_2}\ \text{sym}$$

$$\frac{\mathbb{T} \vdash t_1 = t_2 \quad \mathbb{T} \vdash t_2 = t_3}{\mathbb{T} \vdash t_1 = t_3}\ \text{trans} \qquad \frac{\mathbb{T} \vdash t_1 = t_2}{\mathbb{T} \vdash C[t_1] = C[t_2]}\ \text{ctxt}$$

$$\frac{\mathbb{T} \vdash f^n(t_1, ..., t_{i-1}, a, t_{i+1}, ..., t_n) = g^{n'}(t'_1, ..., t'_{j-1}, a, t'_{j+1}, ..., t'_{n'})}{\mathbb{T} \vdash f^n(t_1, ..., t_{i-1}, t, t_{i+1}, ..., t_n) = g^{n'}(t'_1, ..., t'_{j-1}, t, t'_{j+1}, ..., t'_{n'})}\ \text{subst}$$

Figure 3: Inference rules of equational reasoning

$\mathbb{T} \vdash t_1 = t_2$.

Here is the formalization of the rules of equational logic - implemented in the module `Semantic.SemEquality`:

```
--Definition of semantic equality
data _≡si_ : For −> For −> Set where
  -- equational rules
  reflsi :
    (A : For) −>
    A ≡si A
  transsi :
    (A B C : For) −>
    (A=B : A ≡si B) −>
    (B=C : B ≡si C) −>
    A ≡si C
  symsi :
    (A B : For) −>
    (A=B : A ≡si B) −>
    B ≡si A
  ctxtsi :
    (A B : For) −>
    (Ctxt : Context) −>
    (A=B : A ≡si B) −>
    (Ctxt [ A ]) ≡si (Ctxt [ B ])
  substsi :
    (A B C : For) −>
    (k : ℕ) −>
    (B=C : B ≡si C) −>
    (A [ B / k ]) ≡si (A [ C / k ])
  -- Axioms
```

This can be applied to any set of axioms $\mathbb{T}$.
Birkhoff has proven that syntax and semantic are directly related by the following theorem [Bir35]:

**Theorem 4.1** (Birkhoff's theorem)**.** *Let $\Sigma$ be a signature and $\mathbb{T}$ a $\Sigma$-theory. $\mathcal{M} \vDash t_1 = t_2$ holds for every model of $\mathbb{T}$ if and only if $\mathbb{T} \vdash t_1 = t_2$ is derivable.*

Let's now see a practical example of such theory with lattice-ordered Abelian groups.

## 4.2 Lattice-ordered Abelian groups $\mathbb{A}$

A lattice-ordered Abelian group is an Abelian group $(G, +, 0)$ with a lattice structure $(G, \sqcup, \sqcap)$ compatible with the $+$ operation. The formalized definition is the following:

**Definition 4.10** (Lattice-ordered Abelian group $\mathbb{A}$)**.** The theory of *lattice-ordered Abelian groups* is given by the signature $\Sigma_{\mathbb{A}} = \{0^0, +^2, -^1, \sqcup^2, \sqcap^2\}$ and the following axioms:

**Abelian group axioms**

$$A + (B + C) = (A + B) + C \qquad A + B = B + A$$

$$A + 0 = A \qquad A - A = 0$$

**Lattice ordered axioms**

$$A \sqcup (B \sqcup C) = (A \sqcup B) \sqcup C \qquad A \sqcup B = B \sqcup A \qquad A \sqcup (A \sqcap B) = A$$

$$A \sqcap (B \sqcap C) = (A \sqcap B) \sqcap C \qquad A \sqcap B = B \sqcap A \qquad A \sqcap (A \sqcup B) = A$$

**Compability axiom**

$$((A \sqcap B) + C) \sqcap (B + C) = (A \sqcap B) + C$$

Figure 4: Axioms of lattice-ordered Abelian groups

The definition in Agda is - formalized in the module `Semantic.SemEquality`:

```
--Definition of semantic equality
data _≡s_ : For −> For −> Set where
  -- equational rules
  refls :
    (A : For) −>
    A ≡s A
  transs :
    (A B C : For) −>
    (A=B : A ≡s B) −>
    (B=C : B ≡s C) −>
    A ≡s C
  syms :
    (A B : For) −>
    (A=B : A ≡s B) −>
    B ≡s A
  ctxts :
    (A B : For) −>
    (Ctxt : Context) −>
    (A=B : A ≡s B) −>
    (Ctxt [ A ]) ≡s (Ctxt [ B ])
  substs :
    (A B C : For) −>
    (k : ℕ) −>
    (B=C : B ≡s C) −>
    (A [ B / k ]) ≡s (A [ C / k ])
  -- Axioms
  -- group axioms
  asso−+S :
    (A B C : For) −>
    (A +S (B +S C)) ≡s ((A +S B) +S C)
  commu−+S :
    (A B : For) −>
    (A +S B) ≡s (B +S A)
  neutral−+S :
    (A : For) −>
    (A +S botGA) ≡s A
  opp−+S :
    (A : For) −>
    (A +S (−S A)) ≡s botGA
  -- lattice axioms
  asso−⊔S :
    (A B C : For) −>
    (A ⊔S (B ⊔S C)) ≡s ((A ⊔S B) ⊔S C)
  commu−⊔S :
    (A B : For) −>
    (A ⊔S B) ≡s (B ⊔S A)
  abso−⊔S :
    (A B : For) −>
    (A ⊔S (A ⊓S B)) ≡s A
  asso−⊓S :
    (A B C : For) −>
    (A ⊓S (B ⊓S C)) ≡s ((A ⊓S B) ⊓S C)
  commu−⊓S :
    (A B : For) −>
    (A ⊓S B) ≡s (B ⊓S A)
  abso−⊓S :
    (A B : For) −>
    (A ⊓S (A ⊔S B)) ≡s A
  -- compatibility axioms
  compa−+S :
    (A B C : For) −>
    (((A ⊓S B) +S C) ⊓S (B +S C)) ≡s ((A ⊓S B) +S C)
```

We also define the $\leq$ relation by $A \leq B$ iff $A \sqcap B = A$.

*Remark* 4.1. $A \leq B$ can be seen as an equality so $\mathbb{T} \vdash A \leq B$ means $\mathbb{T} \vdash A \sqcap B = A$.

Here is an example of derivation and its formalization in Agda:

$$\dfrac{\dfrac{}{\mathbb{A} \vdash 0 + A = A + 0}\,\text{ax} \qquad \dfrac{}{\mathbb{A} \vdash A + 0 = A}\,\text{ax}}{\mathbb{A} \vdash 0 + A = A}\,\text{trans}$$

implemented by:

```
0+A=A : (A : For) → botGA +S A ≡s A
0+A=A A =
  transs (botGA +S A) (A +S botGA) A
    (commu−+S botGA A)
    (neutral−+S A)
```

A good part of the intership was to formalize some basic equalities and Horn clauses - propositions of the forms

$$(\bigwedge_{i=1}^{n} \mathbb{T} \vdash A_i = B_i) \Rightarrow \mathbb{T} \vdash A = B$$

translated in types

$$A_1 \equiv_s B_1 \to \ldots \to A_n \equiv_s B_n \to A \equiv_s B$$

- that are useful in proving the soundness of our system in Agda. As you can see, even the most simple equation ($0+A = A$) can not be done in a single step. As another example, the simplest derivation we found of $- - A = A$ requires eight applications of the transitivity rule. We also faced some unexpected complications. For instance, proving $\mathbb{T} \vdash A + A = 0 \Rightarrow \mathbb{T} \vdash A = 0$ was quite difficult - we had to prove that $\mathbb{T} \vdash A + A = 0 \Rightarrow \mathbb{T} \vdash A \sqcup (-A) = 0$ and that $\mathbb{T} \vdash A \sqcup (-A) = 0 \Rightarrow \mathbb{T} \vdash A = 0$ and did not manage to find a more direct way of doing it. In the end, we had to prove more than seventy propositions, going from a few rules (three or four rules) to some very lengthy proofs (about one hundred lines of code for the distributivity of $\sqcup$ over $\sqcap$).

Here are some examples useful in my report:

**Theorem 4.2.** *The Horn clause $\mathbb{A} \vdash A - B = 0 \Rightarrow \mathbb{A} \vdash A = B$ is derivable.*

*Proof.*

$$\Pi_1 = \dfrac{\dfrac{\mathbb{A} \vdash A - B = 0}{\mathbb{A} \vdash (A - B) + B = 0 + B}\,\text{ctxt} \qquad \dfrac{\dfrac{}{\mathbb{A} \vdash 0 + B = B + 0}\,\text{ax} \qquad \dfrac{}{\mathbb{A} \vdash B + 0 = B}\,\text{ax}}{\mathbb{A} \vdash 0 + B = B}\,\text{trans}}{\mathbb{A} \vdash (A - B) + B = B}\,\text{trans}$$

$$\Pi_2 = \dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash B - B = (-B) + B}\,\text{ax}}{\mathbb{A} \vdash A + (B - B) = A + ((-B) + B)}\,\text{ctxt} \qquad \dfrac{\dfrac{}{\mathbb{A} \vdash A + ((-B) + B) = (A - B) + B}\,\text{ax} \qquad \dfrac{\Pi_1}{\mathbb{A} \vdash (A - B) + B = B}}{\mathbb{A} \vdash A + ((-B) + B)}}{\mathbb{A} \vdash A + (B - B) = B}\,\text{trans}$$

$$\dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash A + 0 = A}\,\text{ax}}{\mathbb{A} \vdash A = A + 0}\,\text{sym} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash B - B = 0}\,\text{ax}}{\mathbb{A} \vdash 0 = B - B}\,\text{sym}}{\mathbb{A} \vdash A + 0 = A + (B - B)}\,\text{ctxt} \qquad \dfrac{\Pi_2}{\mathbb{A} \vdash A + (B - B) = B}}{\mathbb{A} \vdash A + 0 = B}\,\text{trans}}{\mathbb{A} \vdash A = B}\,\text{trans}$$

In Agda, it is:

```
Π₁ : {A B : For} → (A −S B) ≡ₛ botGA → (A −S B) +S B ≡ₛ B
Π₁ {A} {B} A−B=0 =
  transₛ ((A −S B) +S B) (botGA +S B) B
    (ctxtₛ (A −S B) botGA (● +C (CC B))
      A−B=0)
    (transₛ (botGA +S B) (B +S botGA) B
      (commu−+S botGA B)
      (neutral−+S B))

Π₂ : {A B : For} → ((A −S B) ≡ₛ botGA) → A +S (B −S B) ≡ₛ B
Π₂ {A} {B} A−B=0 =
  transₛ (A +S (B −S B)) (A +S ((−S B) +S B)) B
    (ctxtₛ (B −S B) ((−S B) +S B) ((CC A) +C ●)
      (commu−+S B (−S B)))
    (transₛ (A +S ((−S B) +S B)) ((A −S B) +S B) B
      (asso−+S A (−S B) B)
      (Π₁ A−B=0))

rule1 : {A B : For} → ((A −S B) ≡ₛ botGA) → A ≡ₛ B
rule1 {A} {B} A−B=0 =
  transₛ A (A +S botGA) B
    (symₛ (A +S botGA) A
      (neutral−+S A))
    (transₛ (A +S botGA) (A +S (B −S B)) B
      (ctxtₛ botGA (B −S B) ((CC A) +C ●)
        (symₛ (B −S B) botGA
          (opp−+S B)))
      (Π₂ A−B=0))
```

□

**Theorem 4.3.** $\mathbb{A} \vdash A \leq A$ *is derivable.*

*Proof.* Let's start by proving a lemma: $\mathbb{A} \vdash ((A \sqcap A) \sqcap B) = (A \sqcap B)$ is derivable.

$$\Pi_1 = \dfrac{\dfrac{}{\mathbb{A} \vdash ((B \sqcap A) + 0) = (B \sqcap A)}\,\text{ax} \qquad \dfrac{}{\mathbb{A} \vdash (B \sqcap A) = (A \sqcap B)}\,\text{ax}}{\mathbb{A} \vdash ((B \sqcap A) + 0) = (A \sqcap B)}\,\text{trans}$$

$$\Pi_2 = \dfrac{\dfrac{}{\mathbb{A} \vdash ((B \sqcap A) + 0) \leq (A + 0)}\,\text{ax} \qquad \dfrac{\Pi_1}{\mathbb{A} \vdash ((B \sqcap A) + 0) = (A \sqcap B)}}{\mathbb{A} \vdash (((B \sqcap A) + 0) \sqcap (A + 0)) = (A \sqcap B)}\,\text{trans}$$

$$\Pi_3 = \dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash (A \sqcap B) = (B \sqcap A)}\,\text{ax}}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap (A + 0)) = (((B \sqcap A) + 0) \sqcap (A + 0))}\,\text{ctxt} \qquad \dfrac{\Pi_2}{\mathbb{A} \vdash (((B \sqcap A) + 0) \sqcap (A + 0)) = (A \sqcap B)}}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap (A + 0)) = (A \sqcap B)}\,\text{trans}$$

$$\Pi_4 = \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash (A + 0) = A}\,\text{ax}}{\mathbb{A} \vdash A = (A + 0)}\,\text{sym}}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap A) = (((A \sqcap B) + 0) \sqcap (A + 0))}\,\text{ctxt} \qquad \dfrac{\Pi_3}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap (A + 0)) = (A \sqcap B)}}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap A) = (A \sqcap B)}\,\text{trans}$$

$$\Pi_5 = \dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash ((A \sqcap B) + 0) = (A \sqcap B)}\,\text{ax}}{\mathbb{A} \vdash (A \sqcap B) = ((A \sqcap B) + 0)}\,\text{sym}}{\mathbb{A} \vdash ((A \sqcap B) \sqcap A) = (((A \sqcap B) + 0) \sqcap A)}\,\text{ctxt} \qquad \dfrac{\Pi_4}{\mathbb{A} \vdash (((A \sqcap B) + 0) \sqcap A) = (A \sqcap B)}}{\mathbb{A} \vdash ((A \sqcap B) \sqcap A) = (A \sqcap B)}\,\text{trans}$$

$$\Pi_6 = \dfrac{\dfrac{}{\mathbb{A} \vdash (A \sqcap (A \sqcap B)) = ((A \sqcap B) \sqcap A)}\,\text{ax} \qquad \dfrac{\Pi_5}{\mathbb{A} \vdash ((A \sqcap B) \sqcap A) = (A \sqcap B)}}{\mathbb{A} \vdash (A \sqcap (A \sqcap B)) = (A \sqcap B)}\,\text{trans}$$

$$\dfrac{\dfrac{\dfrac{}{\mathbb{A} \vdash (A \sqcap (A \sqcap B)) = ((A \sqcap A) \sqcap B)}\,\text{ax}}{\mathbb{A} \vdash ((A \sqcap A) \sqcap B) = (A \sqcap (A \sqcap B))}\,\text{sym} \qquad \dfrac{\Pi_6}{\mathbb{A} \vdash (A \sqcap (A \sqcap B)) = (A \sqcap B)}}{\mathbb{A} \vdash ((A \sqcap A) \sqcap B) = (A \sqcap B)}\,\text{trans}$$

The Agda code for the lemma is:

```
A⊓A⊓B=A⊓B :
  {A B : For} →
  (A ⊓S A) ⊓S B ≡ₛ A ⊓S B
A⊓A⊓B=A⊓B {A} {B} =
  transₛ ((A ⊓S A) ⊓S B) (A ⊓S (A ⊓S B)) (A ⊓S B)
    (symₛ (A ⊓S (A ⊓S B)) ((A ⊓S A) ⊓S B)
      (asso−⊓S A A B))
    (transₛ (A ⊓S (A ⊓S B)) ((A ⊓S B) ⊓S A) (A ⊓S B) -- Π₆
      (commu−⊓S A (A ⊓S B))
      (transₛ ((A ⊓S B) ⊓S A) (((A ⊓S B) +S botGA) ⊓S A) (A ⊓S B) -- Π₅
        (ctxtₛ (A ⊓S B) ((A ⊓S B) +S botGA) (● ⊓C (CC A))
          (symₛ (A ⊓S B +S botGA) (A ⊓S B)
            (neutral−+S (A ⊓S B))))
        (transₛ (((A ⊓S B) +S botGA) ⊓S A) (((A ⊓S B) +S botGA) ⊓S (A +S botGA)) (A ⊓S B) -- Π₄
          (ctxtₛ A (A +S botGA) ((CC (A ⊓S B +S botGA)) ⊓C ●)
            (symₛ (A +S botGA) A
              (neutral−+S A)))
          (transₛ (((A ⊓S B) +S botGA) ⊓S (A +S botGA)) (((B ⊓S A) +S botGA) ⊓S (A +S botGA)) (A ⊓S B) -- Π₃
            (ctxtₛ (A ⊓S B) (B ⊓S A) ((● +C (CC botGA)) ⊓C (CC (A +S botGA)))
              (commu−⊓S A B))
            (transₛ (((B ⊓S A) +S botGA) ⊓S (A +S botGA)) (B ⊓S A +S botGA) (A ⊓S B)-- Π₂
              (compa−+S B A botGA)
              (transₛ (B ⊓S A +S botGA) (B ⊓S A) (A ⊓S B) -- Π₁
                (neutral−+S (B ⊓S A))
                (commu−⊓S B A)))))))
```

This lemma was the difficult part of the theorem so we will not put the rest here but the full theorem is implemented in Agda. □

The process of writing such proofs is - as showed in this report - very tedious and often requires some ingenuity. The issue is associated with the transivity rule of equational logic ($A = B, B = C \Rightarrow A = C$) which requires to correctly guess the right interpolant $B$ among infinitely many possible.

## 5 Hypersequent calculus for lattice-ordered abelian groups

We will now introduce the GA hypersequent calculus.

**Definition 5.1** (Sequent)**.** A *sequent* $H$ is a pair of lists of $\Sigma_{\mathbb{A}}$-terms. A sequent $(\Gamma, \Delta)$ is noted $\Gamma \vdash \Delta$.

**Definition 5.2** (Hypersequent)**.** A *hypersequent* $G$ is a non-empty list of sequents. The list $(H_1, ..., H_n)$ is noted $H_1|...|H_n$. The Agda definition is - implemented in the module `Syntax.HSeq`:

```
data HSeq : Set where
  head :
    (H : Seq) −>
    HSeq
  _|_ :
    (G : HSeq) −>
    (H : Seq) −>
    HSeq
```

**Definition 5.3** (GA system)**.** The *GA system* introduced in [MOG05] is defined by the following inference rules:

**Axioms:**

$$\frac{}{\vdash}\ \Delta \qquad \frac{}{A \vdash A}\ \text{ax}$$

**Structural rules:**

$$\frac{G}{G|\Gamma \vdash \Delta}\ \text{W} \qquad \frac{G|\Gamma \vdash \Delta|\Gamma \vdash \Delta}{G|\Gamma \vdash \Delta}\ \text{C}$$

$$\frac{G|\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}{G|\Gamma_1 \vdash \Delta_1|\Gamma_2 \vdash \Delta_2}\ \text{S} \qquad \frac{G|\Gamma_1 \vdash \Delta_1 \quad G|\Gamma_2 \vdash \Delta_2}{G|\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \text{M}$$

**Logical rules:**

$$\frac{G|\Gamma \vdash \Delta}{G|\Gamma, 0 \vdash \Delta}\ 0_L \qquad \frac{G|\Gamma \vdash \Delta}{G|\Gamma \vdash \Delta, 0}\ 0_R$$

$$\frac{G|\Gamma, A, B \vdash \Delta}{G|\Gamma, A + B \vdash \Delta}\ +_L \qquad \frac{G|\Gamma \vdash \Delta, A, B}{G|\Gamma \vdash \Delta, A + B}\ +_R$$

$$\frac{G|\Gamma \vdash \Delta, A}{G|\Gamma, -A \vdash \Delta}\ -_L \qquad \frac{G|\Gamma, A \vdash \Delta}{G|\Gamma \vdash \Delta, -A}\ -_R$$

$$\frac{G|\Gamma, A \vdash \Delta \quad G|\Gamma, B \vdash \Delta}{G|\Gamma, A \sqcup B \vdash \Delta}\ \sqcup_L \qquad \frac{G|\Gamma \vdash \Delta, A|\Gamma \vdash \Delta, B}{G|\Gamma \vdash \Delta, A \sqcup B}\ \sqcup_R$$

$$\frac{G|\Gamma, A \vdash \Delta|\Gamma, B \vdash \Delta}{G|\Gamma, A \sqcap B \vdash \Delta}\ \sqcap_L \qquad \frac{G|\Gamma \vdash \Delta, A \quad G|\Gamma \vdash \Delta, B}{G|\Gamma \vdash \Delta, A \sqcap B}\ \sqcap_R$$

**Cut rule:**

$$\frac{\Gamma_1, A \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2, A}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \text{cut}$$

**Exchange rules:**

$$\frac{\sigma(G)}{G}\ \text{hseq-ex}$$

$$\frac{G|\sigma(\Gamma) \vdash \delta(\Delta)}{G|\Gamma \vdash \Delta}\ \text{seq-ex}$$

with $\sigma$ and $\delta$ some permutations.

A derivation in the GA-system is called a *proof*.

*Remark* 5.1. In Metcalfe's paper [MOG05], the exchange rules are not necessary since multisets are used instead of lists.

**Definition 5.4** (Provability)**.** A hypersequent is said to be *provable* if there exists a proof in GA. It is said to be *cut-free provable* if there exists a proof that does not use the cut-rule.

Formalizing the definition of a proof in Agda was a little more complicated than expected. Indeed, in the rule

$$\frac{G|\Gamma \vdash \Delta}{G|\Gamma, 0 \vdash \Delta}\ 0_L$$

, $G$ is not really a hypersequent in the sense that it can be empty. The notational shortcut is not allowed in Agda, so in Agda the two rules

$$\frac{G|\Gamma \vdash \Delta}{G|\Gamma, 0 \vdash \Delta}\ 0_L$$

and

$$\frac{\Gamma \vdash \Delta}{\Gamma, 0 \vdash \Delta}\ 0_L$$

must both be implemented. It makes the definition of a proof quite lengthy - thirty four rules instead of eighteen. Moreover, for the exchange rules, a proof that the permutations exist is needed to apply the rule. The permutations themselves are not asked, only a proof of their existence - given by the binary predicat ListExchange for the lists and HSeqExchange for the hypersequent - is needed. Here are some rules in Agda - implemented in the modules `Syntax.Proof` and `Syntax.CutProof`:

```
data Proof : HSeq → Set where
  ax :
    (A : For) →
    Proof (head ([] :: A , ([] :: A)))
  Δ−ax :
    Proof (head ([] , []))
  -- lot of rules
  S :
    (G : HSeq) →
    (Γ1 Γ2 Δ1 Δ2 : GAList) →
    Proof (G | Γ1 , Δ1) →
    Proof (G | Γ1 , Δ1 | Γ2 , Δ2)
  S−head :
    (Γ1 Γ2 Δ1 Δ2 : GAList) →
    Proof (head (Γ1 , Δ1)) →
    Proof (head (Γ1 , Δ1) | Γ2 , Δ2)
  M−head :
    (Γ1 Γ2 Δ1 Δ2 : GAList) →
    Proof (head (Γ1 , Δ1)) →
    Proof (head (Γ2 , Δ2)) →
    Proof (head (union Γ1 Γ2 , union Δ1 Δ2))

  -- lot of rules
  plus−L :
    (G : HSeq) →
    (Γ Δ : GAList) →
    (A B : For) →
    Proof (G | Γ :: A :: B , Δ) →
    Proof (G | Γ :: (A +S B) , Δ)
  -- lot of rules
  seq−exchange :
    (G : HSeq) →
    (Γ1 Γ2 Δ1 Δ2 : GAList) −>
    ListExchange Γ1 Γ2 →
    ListExchange Δ1 Δ2 →
    Proof (G | Γ1 , Δ1) →
    Proof (G | Γ2 , Δ2)
```

Before formulating the completeness and soundness of this system, some more definitions are required - for instance, how to go from a hypersequent to a formula.

**Definition 5.5** (Interpretation)**.**

The interpretation of a list of formula $\Gamma = A_1, ..., A_n$, noted $[\![\Gamma]\!]$ is the sum $\sum_{i=1}^{n} A_i$.

The interpretation of the hypersequent $G = \Gamma_1 \vdash \Delta_1 | ... | \Gamma_n \vdash \Delta_n$, noted $[\![G]\!]$, is $\bigsqcup_{i=1}^{n}([\![\Delta_i]\!] - [\![\Gamma_i]\!])$.

The completeness and soundness of the system can now be defined:

- Completeness : If $\mathbb{A} \vdash A = B$ is derivable then are $A \vdash B$ and $B \vdash A$ provable in GA?
- Soundness : If $G$ is provable in GA then is $\mathbb{A} \vdash 0 \leq [\![G]\!]$ derivable?

**Lemma 5.1.** *$\mathbb{A} \vdash A = B$ is derivable if and only if $\mathbb{A} \vdash 0 \leq [\![B \vdash A]\!]$ and $\mathbb{A} \vdash 0 \leq [\![A \vdash B]\!]$ are derivable.*

**Idea of proof:** Implemented in the module `Semantic.SemEquality.Proprieties`.

If $\mathbb{A} \vdash A = B$ is derivable, then $\mathbb{A} \vdash A - B = 0$ and $\mathbb{A} \vdash B - A = 0$ are derivable. Since $\mathbb{A} \vdash 0 \leq 0$ (theorem 4.3) is derivable, so are $\mathbb{A} \vdash 0 \leq [\![B \vdash A]\!]$ and $\mathbb{A} \vdash 0 \leq [\![A \vdash B]\!]$.

If $\mathbb{A} \vdash 0 \leq [\![B \vdash A]\!]$ and $\mathbb{A} \vdash 0 \leq [\![A \vdash B]\!]$ are derivable. $\mathbb{A} \vdash 0 \leq B - A$ is derivable so $\mathbb{A} \vdash A - B \leq 0$ is derivable (proven in Agda). Let's now show that $\mathbb{A} \vdash A - B = 0$ is then derivable (theorem 4.2 allows to conclude). The proof tree is:

$$\dfrac{\dfrac{\mathbb{A} \vdash A - B \leq 0}{\mathbb{A} \vdash A - B = (A - B) \sqcap 0}\,sym \qquad \dfrac{\dfrac{}{\mathbb{A} \vdash (A - B) \sqcap 0 = 0 \sqcap (A - B)}\,ax \qquad \mathbb{A} \vdash 0 \leq A - B}{\mathbb{A} \vdash (A - B) \sqcap 0 = 0}\,trans}{\mathbb{A} \vdash A - B = 0}\,trans$$

As usual, the corresponding Agda code is:

```
antisym :
  {A B : For} →
  botGA ≤S (A −S B) →
  (A −S B) ≤S botGA →
  A −S B ≡s botGA
antisym {A} {B} 0<A−B A−B<0 =
  transs (A −S B) ((A −S B) ⊓S botGA) botGA
    (syms ((A −S B) ⊓S botGA) (A −S B)
      A−B<0)
    (transs ((A −S B) ⊓S botGA) (botGA ⊓S (A −S B)) botGA
      (commu−⊓S (A −S B) botGA)
      0<A−B)
```

□

**Theorem 5.1.** *If the system is sound and complete, to have an algorithm to find a proof of a hypersequent (if it exists) immediately gives an algorithm to find the derivation of an equality.*

**Idea of proof:** If the system is sound and complete and if we have an algorithm to find a proof of a hypersequent (if it exists), the algorithm to find a derivation for the equality $A = B$ is:

1. find a proof $\Pi_1$ of $A \vdash B$. If no proof exists, the equality is not derivable.
2. find a proof $\Pi_2$ of $B \vdash A$. If no proof exists, the equality is not derivable.
3. apply the soundness function to $\Pi_1$ and $\Pi_2$ to have derivation of $\mathbb{A} \vdash 0 \leq A - B$ and $\mathbb{A} \vdash 0 \leq B - A$.
4. apply the lemma 5.1 to obtain a derivation of $\mathbb{A} \vdash A = B$.

The correctness of this algorithm is given by the completeness: if a derivation of $\mathbb{A} \vdash A = B$ exists, then $A \vdash B$ and $B \vdash A$ are provable. So the algorithm does not fail and returns a derivation of $A = B$. □

All the tools to prove the soundness of the system are now defined but some other definitions are needed to prove the completeness.

**Definition 5.6** (Atomic hypersequent)**.** A hypersequent is said *atomic* if only variables appear in it.

**Definition 5.7** (Positive hypersequent)**.** A hypersequent $G$ is said *positive* if $\mathbb{A} \vdash 0 \leq [\![G]\!]$ is derivable.

**Definition 5.8** ($\lambda$-property)**.** Let $A$ be a formula and $k$ a natural number then $k * A = \sum_{i=1}^{k} A$.
A hypersequent $G = \Gamma_1 \vdash \Delta_1 | ... | \Gamma_n \vdash \Delta_n$ is said to have the $\lambda$-property if there exist $(\lambda_i)_{i \in [1..n]} \in \mathbb{N}^n$ such that:

- there is $i \in [1..n]$ such that $\lambda_i \neq 0$
- $\mathbb{A} \vdash \sum_{i=1}^{n} \lambda_i * \Delta_i - \sum_{i=1}^{n} \lambda_i * \Gamma_i = 0$

Everything needed to prove the completeness and the soundness is now defined. We will now give an overview on how we implemented the completeness and soundness in Agda.

## 6 Soundness ans completeness

We will first prove the soundness of the GA deductive system and then introduce two ways to prove the completeness of the cut-free GA deductive system. Both those implementations of the completeness have a "black box", theorems proven in Metcalfe's paper [MOG05] that we did not yet manage to prove in Agda.

### 6.1 Soundness

Soundness : If $G$ is provable in GA then $\mathbb{A} \vdash 0 \leq [\![G]\!]$ is derivable.

**Theorem 6.1.** *All rules of GA are sound, meaning that that if a rule has premises $P_1, ..., P_n$ and a conclusion C and that $\mathbb{A} \vdash 0 \leq [\![P_i]\!]$ is derivable for all i $\in [1..n]$ then $\mathbb{A} \vdash 0 \leq [\![C]\!]$ is derivable.*

**Idea of proof:** Implemented in the modules `Syntax.Proof.Soundness` and `Syntax.CutProof.Soundness.`
Most of the rules are straightfoward - for instance proving the soundness of the rule $0_L$, meaning that if $\mathbb{A} \vdash [\![G \mid \Gamma \vdash \Delta]\!]$ is derivable then $\mathbb{A} \vdash [\![G \mid \Gamma, 0 \vdash \Delta]\!]$ is derivable, is simply using the fact 0 is the neutral element of the addition. Or proving the soundness of the rule $+_L$, meaning that if $\mathbb{A} \vdash [\![G \mid \Gamma, A, B \vdash \Delta]\!]$ is derivable then $\mathbb{A} \vdash [\![G \mid \Gamma, A + B \vdash \Delta]\!]$, is derivable is mostly from the fact that a list of formula is interpreted by a sum.
Yet three rules were a lot more difficult to implement:

- the S-rule: in the proof of the S-rule, we want to prove that

$$0 \leq [\![G]\!] \sqcup [\![\Gamma_1 \vdash \Delta_1]\!] \sqcup [\![\Gamma_2 \vdash \Delta_2]\!]$$

  and we need to use a property that only holds for positive elements: if $a$ is positive, then $a \leq a + a$. Yet our elements are not necessarily positive. One solution is to realize that

$$0 \leq [\![G]\!] \sqcup [\![\Gamma_1 \vdash \Delta_1]\!] \sqcup [\![\Gamma_2 \vdash \Delta_2]\!]$$

  if and only if

$$0 = ([\![G]\!] \sqcap 0) \sqcap ([\![\Gamma_1 \vdash \Delta_1]\!] \sqcup [\![\Gamma_2 \vdash \Delta_2]\!] \sqcup 0)$$

  proven in `Semantic.SemEquality.Properties.` $[\![G]\!] \sqcap 0$ and $[\![\Gamma_1 \vdash \Delta_1]\!] \sqcup [\![\Gamma_2 \vdash \Delta_2]\!] \sqcup 0$ are positive and so we can use the property we mentionned earlier.

- the M-rule: here also, we needed to use a property that holds only for positive elements : for every positive elements $a, b, c$, $(a \sqcup b) + (a \sqcup c) \leq a \sqcup (b + c)$.
- the max-L-rule: the distributivity of $\sqcup$ over $\sqcap$ was needed and like we previously said, this property is difficult to prove.

□

**Theorem 6.2** (Soundness)**.** *If $G$ is provable then $\mathbb{A} \vdash [\![G]\!]$ is derivable.*

**Idea of proof:** Implemented in the modules `Syntax.Proof.Soundness` and `Syntax.CutProof.Soundness`.
This is done by induction on the proof of $G$, using the fact that all rules are sound. □ Now that the soundness is proven in GA+cut, we can now focus on the completeness of the system.

## 6.2 Completeness with cut

With the cut-rule, it is not difficult to prove the completeness of GA:

**Theorem 6.3** (Completeness of GA+cut)**.** *If $\mathbb{A} \vdash A = B$ is derivable then both $A \vdash B$ and $B \vdash A$ are provable.*

**Idea of proof:** Implemented in the module `Syntax.CutProof.Completeness`.
This theorem is proven by induction on the derivation of $\mathbb{A} \vdash A = B$. This proof by induction can be divided into two parts:

- finding a proof of $A \vdash B$ and a proof of $B \vdash A$ for all axioms $A = B$ of lattice-ordered Abelian groups. For instance, for the axiom $A + 0 = A$, the proof of $A + 0 \vdash A$ is:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{ax}}{A, 0 \vdash A}\,0_L}{A + 0 \vdash A}\,+_L$$

  which is implemented in Agda by:

```
completeness1 (neutral-+S A) =
  plus-L-head
    []
    ([] :: A)
    A
    botAG
    (Z-L-head
      ([] :: A)
      ([] :: A)
      (ax A))
```

- translating all inference rules. For instance, for the transitivity rule

$$\dfrac{\mathbb{A} \vdash A = B \quad \mathbb{A} \vdash B = C}{\mathbb{A} \vdash A = C}\,\text{trans}$$

  we have by induction a proof of $A \vdash B$ and a proof of $B \vdash C$. A proof of $A \vdash C$ is then:

$$\dfrac{\dfrac{\text{Induction Hypothesis}}{B \vdash C} \quad \dfrac{\text{Induction Hypothesis}}{A \vdash B}}{A \vdash C}\,\text{cut}$$

  which is implemented in agda by:

```
completeness1 (trans_s A=B B=C) =
 cut [] ([] :: A) ([] :: C) [] B
  (completeness1 B=C)
  (completeness1 A=B)
```

□

**Theorem 6.4** (Completeness of cut-free GA)**.** *If $G$ is provable, then $G$ is provable without using the cut-rule.*

Implemented in the module `Syntax.CutProof.Completeness`.

*Remark* 6.1. In Metcalfe's paper [MOG05], the completeness of cut-free GA is proven thanks to the $\lambda$-property and then they conclude that the GA deductive system and the cut-free GA deductive system are equivalent. This is one of the black box we have not yet managed to implement in Agda.

## 6.3 Completeness - how to get an atomic hypersequent

**Definition 6.1** (Invertible rule)**.** A rule with premises $P_1, ..., P_n$ and a conclusion $C$ is said *invertible* if $\mathbb{A} \vdash [\![C]\!]$ derivable implies that $\mathbb{A} \vdash [\![P_i]\!]$ derivable for all $i \in [1, ..., n]$.

**Theorem 6.5.** *All logical and exchange rules are invertible.*

**Idea of proof:** Implemented in the modules `Syntax.Proof.Invertibility` and `Syntax.CutProof.Invertibility.`

As in the Soundness theorem, we have proven the required long list of implications. Some turned out to be more complicated than others (e.g., S rules required 149 lines). In Agda, for every premise, a function was implemented - meaning that if a rule has premises $P_1, ..., P_n$ and a conclusion $C$, $n$ functions were implemented with type: botGA≤$[\![C]\!]$ → botGA≤$[\![P_i]\!]$. □

**Theorem 6.6.** *For all hypersequent G, there exists a list $l_G$ of atomic hypersequents such that if all hypersequents of this list are provable then G is provable.*

**Idea of proof:** Implemented in the module `Syntax.Preproof.Create.`

The idea is to apply logical and exchanges rules until there is only variables left in the hypersequents. If all hypersequents of $l_G$ are provable, the proof can then be completed to have a proof of $G$.
This theorem was probably the most tricky part to implement in Agda.

First of all, $l_G$ alone is not enough, the "way" to go from $G$ to $l_G$ by using the logical and exchange rules is also needed since the proof of $G$ is found by completing this "preproof". We had to create an other data type called "Preproof" that was similar to the type Proof, but without the axioms and structural rules. Then we implemented a function that, given a hypersequent, returns a preproof of this hypersequent and finally we proved that this function would return a preproof with only atomic hypersequents in the leaves.

The implementation of the function that returns this preproof was really difficult. It seems simple - apply all possible logical and exchange rules - but Agda only accepts terminating functions. We had to find a criteria that would help prove that the function was terminating - here, induction on a well-founded order on $\mathbb{N} \times \mathbb{N}$ - and we had to prove that at each application of a logical rule, this couple was indeed decreasing.

Quite a lot of implementations were needed that we will not mention here - like the fact that we need the hypersequent to be sorted at each step of the function - but this function was probably the biggest part of my internship despite its intuitive simplicity. □

**Theorem 6.7.** *For every hypersequent G, $\mathbb{A} \vdash [\![G]\!]$ is derivable if and only if for every hypersequent $G'$ in $l_G$ - the list of atomic hypersequents obtained by applying the logical and exchange rules - $\mathbb{A} \vdash [\![G']\!]$ is derivable.*

**Idea of proof:** Implemented in the module `Syntax.Preproof.Create.`

Simply an application of the soundness and invertibility of logical and exchange rules. □

With theorems 6.6 and 6.7, we can now conclude that the completeness of the system GA is equivalent to the completeness for atomic hypersequents.

## 6.4 Completeness - how to prove an atomic hypersequent

The idea to prove the completeness for atomic hypersequents relies on the $\lambda$-property. In Metcalfe's paper, it is proven that an atomic hypersequent $G$ is positive - meaning that $\mathbb{A} \vdash 0 \leq [\![G]\!]$ is derivable - if and only if the $\lambda$-property holds for this hypersequent. Then if an atomic hypersequent follows the $\lambda$-property, we prove that the W-rule, the C-rule and the S-rule can be used to obtain a hypersequent of the form $\Gamma \vdash \Gamma$ which is provable.

**Theorem 6.8.** *Let $\Gamma$ be a list of formulas. $\Gamma \vdash \sigma(\Gamma)$ is provable for all permutation $\sigma$.*

*Proof.* Implemented in the module `Syntax.Proof.Completeness.` We prove by induction on the length of $\Gamma$ that $\Gamma \vdash \Gamma$ is provable. If $\Gamma = []$ then the $\Delta$-rule proves it. Otherwise $\Gamma = \Gamma', A$. By induction hypothesis, $\Gamma' \vdash \Gamma'$ can be proven and with the M-rule:

$$\dfrac{\dfrac{\text{Induction Hypothesis}}{\Gamma' \vdash \Gamma'} \quad \dfrac{}{A \vdash A}\,ax}{\Gamma', A \vdash \Gamma', A}\,M$$

In Agda, the proof is:

```
  Γ⊢Γ :
    (Γ : GAList) ->
    Proof (head (Γ , Γ))
  Γ⊢Γ [] =
    Δ−ax
  Γ⊢Γ (Γ :: A) =
    M−head Γ ([] :: A) Γ ([] :: A)
      (Γ⊢Γ Γ)
      (ax A)
```

Then we apply the exchange rule:

$$\frac{\Gamma \vdash \Gamma}{\Gamma \vdash \sigma(\Gamma)}\ seq-ex$$

□

**Theorem 6.9.** *Let $\Gamma, \Delta$ be two lists of atomic formulas. If $\mathbb{A} \vdash [\![\Gamma]\!] = [\![\Delta]\!]$ is derivable then there exists a permutation $\sigma$ such that $\Delta = \sigma(\Gamma)$.*

**Idea of proof:** Implemented in the modules `Syntax.Seq.Properties` and `Syntax.ListTerm.Canonic.`

The idea is to introduce a canonical form for the atomic list of formulas - which is a list of variables. In Agda, variables are defined by the constructor $var : \mathbb{N} \to Formula$, so the canonical form of a list of variables is the corresponding increasing list of variables. For instance, the canonic form of $var5, var2, var9$ is $var2, var5, var9$.

Then we proved that if two lists had the same interpretation, then they had the same canonical form. For that, we used the fact that $\mathbb{Z}$ is a model of lattice-ordered Abelian group theory. Since $\mathbb{A} \vdash [\![\Gamma]\!] = [\![\Delta]\!]$ then for any valuation $v$, $v([\![\Gamma]\!]) = v([\![\Delta]\!])$. Applying this to the valuation that send only one variable to 1 and all others variables to 0, we obtain that $\Gamma$ and $\Delta$ have the same variables.

Since they have the same variables, they have the same canonical form, which is easily proven by induction on the canonical form. □

Theorem 6.9 was quite tricky to implement, it needed a lot of preliminary works. The idea to use a canonical form was not really intuitive - if two lists have the same variables, clearly we can reorder one list to get the other. The "clearly" part was not so obvious in Agda, where we mostly work by induction. But working on induction on two regular lists was not possible - if we know that $\Gamma, a$ and $\Delta, b$ have the same variables, we do not necessarily know that $\Gamma$ and $\Delta$ have the same variables. The canonical form allowed us to work by induction since, if two canonic lists $\Gamma, a$ and $\Delta, b$ have the same variables, necessarily $a = b$ and $\Gamma$ and $\Delta$ have the same variables.

**Theorem 6.10.** *If the $\lambda$-property holds for an atomic hypersequent then it is provable.*

**Idea of proof:** Implemented in the module `Syntax.Proof.Completeness.`

The coefficients of the $\lambda$-property explain how to use the rules to obtain the correct hypersequent:

- if $\lambda_i = 0$, the W-rule is used once
- if $\lambda_i > 0$, the C-rule is used $\lambda_i - 1$ times

Then the S-rule is used until only one sequent remains. At this step, there is two lists of formulas $\Gamma, \Delta$ such that the remaining sequent is $\Gamma \vdash \Delta$ and thanks to the $\lambda$-property, $\mathbb{A} \vdash [\![\Gamma]\!] = [\![\Delta]\!]$ is derivable. The theorems 6.8, 6.9 and 6.10 allow to conclude. □

**Theorem 6.11.** *If an atomic hypersequent is positive, then it has the $\lambda$-property.*

Admitted in the module `Syntax.Proof.Completeness.`

This theorem has been proved in Metcalfe's paper using the fact that $\mathbb{Q}$ is an universal model of lattice-ordered Abelian groups - meaning that $\mathbb{A} \vdash A = B$ is derivable if and only if $\mathbb{Q} \models A = B$. Yet, our work is thought in a way that the logic can be extended, and the fact that $\mathbb{Q}$ is an universal model will not hold in the extended logics so the proof in Metcalfe's paper is not really compatible with our work.

As for now, this theorem is admitted in our work since it would require the formalization of arguments from Linear arithmetics in Agda, and this goes well beyond the scope of this report.

**Theorem 6.12.** *If an atomic hypersequent is positive, then it is provable.*

*Proof.* Implemented in the module `Syntax.Proof.Completeness.`

A corollary of theorems 6.10 and 6.11. □

**Theorem 6.13** (Completeness)**.** *GA is complete.*

*Proof.* Implemented in the module `Syntax.Proof.Completeness.`

Theorem 6.7 states that is an hypersequent $G$ is positive, there is a list of atomic postive hypersequents $l_G$ such that if all hypersequents of $l_G$ are provable, then $G$ is provable. Theorem 6.12 states that all positive atomic hypersequents are provable. So $G$ is provable. □

## 7 Conclusion

Motivated by recent applications in the area of probabilistic logics (like [MFM17]) we have formalized in the proof assistant Agda the equational theory of lattice-ordered Abelian groups and the hypersequent calculus GA of [MOG05]. Regarding the equational theory, we have formally proved several equations and equational (Horn) implications, including some nontrivial ones requiring more than 100 lines of Agda code - as the proof of

$$\forall A, B, (A + A) \sqcup (B + B) = (A \sqcup B) + (A \sqcup B)$$

in the module `Semantic.SemEquality.Properties`.

Regarding the Hypersequent calculus GA of [MOG05] we have formalized all basic syntactical definitions, the proof of soundness and the proof of completeness. We have also formalized the core argument of the cut elimination theorem up-to a lemma (coming from the theory of Linear programming) which was hard to implement in Agda and that has been left as a black box

Since every proof in Agda corresponds to a computable function we get:

- From the soundness theorem: an algorithm that takes a GA-derivation (possibly with cut) and returns a derivation in equational logic.
- From the completeness theorem: an algorithm that takes a derivation in equational logic and returns a GA-derivation (possibly with cut).
- From the completeness theorem witouth cuts: an algorithm that takes a hypersequent and a proof that this hypersequent is positive and finds a GA derivation for it. Since we did not implement a lemma (from the theory of Linear programming) the algorithm works up-to a blackbox algorithm (which could be provided, for instance, from any solver for linear programming).

Unlike Metcalfe, Olivetti and Gabbay in [MOG05], we decided to prove completeness with purely syntactical means, not relying on model theoretic properties such as the fact that the variety of lattice-ordered Abelian groups is generated by $(\mathbb{R}, +, 0, \max, \min)$. These model theoretic results are extremely useful but rather hard to formalize in theorem provers, as they involve significant amounts of infinitary mathematics. Furthermore, our formalization based on syntactical arguments can serve as a basis for further work on theories (caming from probabilistic logics such as [MFM17] or [Mio18]) which do not enjoy such pleasant model-theoretic properties at all.

## References

[Bir35] Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4), 1935.

[Gen35] Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39:176–210, 1935.

[LFZdG13] Dongxi Liu, Neale L. Fulton, John Zic, and Martin de Groot. Verifying an aircraft proximity characterization method in coq. pages 86–101, 2013.

[MFM17] Mio, Furber, and Mardare. Riesz modal logic for Markov processes. *In Proceedings of LICS*, 2017.

[Mio18] Matteo Mio. Riesz modal logic with threshold operators. *In Proceedings of LICS*, 2018.

[MOG05] George Metcalfe, Nicola Olivetti, and Dov Gabbay. Sequent and hypersequent calculi for abelian and lukasiewicz logics. *ACM Trans. Comput. Logic*, 6(3):578–613, July 2005.

[Nor07] Ulf Norell. Towards a practical programming language based on dependent type theory. *PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology*, September 2007.