UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
"GIOVANNI DEGLI ANTONI"

Corso di Laurea in Informatica

PROOF SEARCH IN PROPOSITIONAL LINEAR LOGIC VIA BOOLEAN CONSTRAINTS SATISFACTION

Relatore: Prof. Alberto Momigliano
Correlatore: Prof. Camillo Fiorentini

Tesi di Laurea di
Martino D'Adda
Matr. 964827

ANNO ACCADEMICO 2023-2024

Contents

Chapter 1

Intro

...

Sequent calculus is a formalism first introduced by G. Gentzen in 1934 [2, 3]. Very briefly, a sequent is just an implication between finite sequences of formulae, such that

$$\Delta \vdash \Gamma$$

is to be interpreted as

$$\bigwedge_{\phi \in \Delta} \phi \Rightarrow \bigvee_{\gamma \in \Gamma} \gamma$$

where Δ is called the antecedent, and Γ the succedent.

Sequents are manipulated using rules. These rules are represented by having premises and conclusion separated by what is called the inference line. A rule acting on the antecedent is called a left rule or an introduction rule, whereas a rule acting on the succedent is called a right rule or an elimination rule. To build a proof the conclusions of rules are chained to the premises of other rules, thus building a tree.

There are three special rules – called structural rules – often left implicit in proofs. We give their left versions:

- weakening: one may "weaken" the sequent by adding a proposition without changing its truth
$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$
- contraction: one may "contract" two copies of the same proposition into one without changing the truth of the sequent
$$\frac{\Gamma, \phi, \phi \vdash \Delta}{\Gamma, \phi \vdash \Delta}$$

- exchange: one may switch position of two propositions in a sequent freely without changing its truth

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \psi, \phi \vdash \Delta}$$

...

Linear logic is a logic proposed by Jean-Yves Girard in his seminal paper of 1987 [4]. The distinctive trait of this logic is that its formulae cannot be copied or discarded, but instead they are consumed. Put differently, its sequent calculus lacks the structural rules of weakening and contraction, making linear logic a substructural logic.

Under these rules a certain judgment is provable if and only if all of its formulae get used exactly once; for this reason this logic is sometimes called a logic of resources, in the same way classical logic is a logic of truths and intuitionistic logic is a logic of proofs.

In linear logic each connective of classical logic has two versions: an additive one – where the two branches keep the same context, and a multiplicative one – where the context gets partitioned between the two branches. To better understand why let's analyze classic conjunction, which can be defined as

$$\frac{\Delta \vdash \phi_2, \Gamma \qquad \Delta \vdash \phi_1, \Gamma}{\Delta \vdash \phi_1 \wedge \phi_2, \Gamma} \quad \frac{\Delta' \vdash \phi_1, \Gamma' \qquad \Delta'' \vdash \phi_2, \Gamma''}{\Delta', \Delta'' \vdash \phi_1 \wedge \phi_2, \Gamma', \Gamma''}$$

These two rules are equivalent only if the use of weakening and contraction is permitted, so in linear logic the two interpretations are distinct: the left one is the additive one, and the right one the multiplicative one. Obviously the constants $\top$ and $\bot$ also have two versions. We have that

Class.	Add.	Mult.
$\wedge$	$\&$	$\otimes$
$\vee$	$\oplus$	$⅋$
$\top$	$\top$	1
$\bot$	0	$\bot$

It is the multiplicative side which brings the most complexity: in bottom-up proof search, the action of searching the correct partition of the sequents – called splitting – may imply an exponential number of attempts.

Linear logic defined as of right now, albeit having the added complexity of splitting, is nonetheless decidable: since formulae are finite and they cannot be copied, it is possible to explore all the possibilities. To make linear logic as strong as classical logic two new connectives are added: $!\phi$ and $?\phi$ – called respectively "of-course" and “why-not”. These are called exponentials and their purpose is to localize uses of contraction and weakening. For example, formulas marked with ! can be used any number of times. Since undecidability may lead to non-termination during proof search, usally some kind of bound is put (e.g. on the number of contractions per branch). This has the side effect of making the proover sometimes reject judgments which may be true, but would need too many contractions.

In 2001 D. Pym and J. Harland publish a paper [5] where they propose a new way of tackling the problem of splitting in a number of different logics, by means of boolean constraints. These constraints are generated during proof search from boolean expressions associated to the formulae, and are used to enforce linearity. This way the complexity shifts from choosing the right set of formulas to prove a certain branch, to solving for boolean assignment – a problem for which there are much more sophisticated algorithms.

In §2 we define a focused and one-sided version to the calculus described in [5], and we give a proof of its soundness consisting of a forgetful functor to the triadic calculus of [1]. In §3 we discuss a Prolog implementation of the calculus of the previous chapter. In §4 we quickly describe the main implementation details of two other top-down provers for full linear logic: llprover and APLL. Finally in §5 we describe the framework built to test and compare our prover with the others from the previous section, and then we show the results of these benchmarks.

Chapter 2

The focused calculus

In this chapter we will define the focused one sided constraint calculus for full linear logic. This calculus is a hybrid between the one defined in [5] and the triadic calculus in [1]: it uses focusing for its performance wise advantages; and constraints to manage splitting efficiently.

Before all, we give we definition of a linear logic formula.

Definition 2.0.1 (Linear logic formula)**.** A linear logic formula is a term defined as follows

$$\begin{array}{rcccl} \phi & ::= & \phi \otimes \phi & \mid & 1 \\ & \mid & \phi \parr \phi & \mid & \bot \\ & \mid & \phi \multimap \phi & & \end{array}\Bigg\}\text{(Multiplicatives and their identities)}$$

$$\begin{array}{rcccl} & \mid & \phi \oplus \phi & \mid & 0 \\ & \mid & \phi \,\&\, \phi & \mid & \bot \end{array}\Big\}\text{(Additives and their identities)}$$

$$\begin{array}{rccccl} & \mid & !\phi & \mid & ?\phi & \text{(Exponentials)} \\ & \mid & \phi^{\bot} & & & \text{(Negation)} \\ & \mid & \alpha & & & \text{(Atom)} \end{array}$$

We will use ϕ to denote formulae and α to denote atoms. A sequent, written Δ or Γ, is a multiset of formulae.

2.1 Normalization

Since in linear logic negation is symmetric and an involution, it is usual to work only with formulae in negated normal form.

Definition 2.1.1 (Negated Normal Form – NNF)**.** A formula is in NNF if all its linear implications ($\multimap$) are expanded to pars ($\parr$) using the tautology

$$a \multimap b \Leftrightarrow a^{\perp} \parr b$$

and negation is pushed down to atoms. Alternatively a formula in NNF is defined as follows:

$$\begin{array}{rcccc} \phi & ::= & \phi \otimes \phi & \mid & 1 \\ & \mid & \phi \parr \phi & \mid & \perp \\ & \mid & \phi \oplus \phi & \mid & 0 \\ & \mid & \phi \,\&\, \phi & \mid & \perp \\ & \mid & !\phi & \mid & ?\phi \\ & \mid & \alpha^{\perp} & \mid & \alpha \end{array}$$

A sequent is in NNF iff all its formulae are in NNF.

A generic formula is normalized by applying recursively the DeMorgan rules for linear logic, until NNF is reached. Normalization of judgments instead takes a two-sided judgment of the form

$$\Delta \vdash \Gamma$$

and transforms it into a one-sided judgment

$$\vdash \Delta'$$

where the right side is composed of the normalization of Γ and $\Delta^{\perp}$.

Normalization has some implementation-wise advantages, but for now it is only important because it shrinks the size of the complete calculus by roughly half, since we only have to deal with the right rules of the connectives.

2.2 Focusing

Focusing is a technique described by Andreoli in his seminal paper [1]. In it he recognizes two alternating phases in a proof: a deterministic phase, where the order of rule application does not matter; and a non deterministic phase, where several choices may be available. These two phases are called respectively asynchronous and synchronous phase. To all formulae is assigned a polarity:

- connectives with a synchronous right rule are defined to have a positive polarity, these are:

$$\otimes, \oplus, \;!\;, 1$$

- whereas connectives with a asynchronous right rule have a negative one, these are:
$$\parr, \&, \; ? \;, \top, \bot$$

and thus the polarity of a formula depends on its top-level connective. For atoms instead polarities may be assigned with some arbitrarily complex mechanisms. We will follow [6] and simply assign atoms with a negative polarity and negated atoms with a positive one. Since we work in one sided linear logic and connectives have only right rules, positive connectives may also be called synchronous, and negative connectives asynchronous.

Definition 2.2.1. Based on the definitions above we define the following predicates:

- "ϕ atom" is true whenever ϕ is an atom;
- "ϕ asy" is true whenever ϕ has as an asynchronous top-level connective;
- "ϕ negative" is true whenever ϕ is either an atom or an asynchronous connective, so
$$\phi \text{ negative} = \phi \text{ atom} \vee \phi \text{ asy}$$

2.3 Constraints

During proof search the calculus generates constraints as the formulae get broken up. The purpose of these is to ensure that if the constraints generated are satisfiable, then the formulae have been used linearly. There are a number of ways in which solutions to these sets of constraints may be propagated during proof search. What we will use is the strategy called "lazy" by [5], which consists of checking for satisfiability on the leaves, and propagating the assignments to the next branch.

Definition 2.3.1 (Variables, expressions)**.** A boolean variable is simply a symbol to which one can associate a value of true or false. A boolean expression, in our case, is just a conjunction of possibly negated boolean variables as

$$\begin{array}{rclcll} x & ::= & x_i & \mid & \overline{x_i} & \text{(Variable)} \\ e & ::= & x \wedge e & \mid & x & \text{(Expression)} \end{array}$$

We will call e such a conjunction and x the single boolean variables. Given a boolean expression e we write
$$\text{vars}(e) = \{x_i \mid x_i \in e\}$$

Definition 2.3.2 (New variables)**.** Sometimes we will write

$$x \text{ new}, X \text{ new}$$

These respectively mean that:

- the variable name x has not yet occurred in any expression of the proof tree. One can think of a global counter which is incremented for each new variable.
- each variable name $x_i, x_j \in X$ has not yet occurred in the proof and each variable in X is distinct:

$$(\forall x_i \in X \mid x_i \text{ new}) \wedge (\forall x_i, x_j \in X \mid i \neq j \Rightarrow x_i \neq x_j)$$

Definition 2.3.3 (Annotated formula)**.** Given a formula ϕ defined as in Definition 2.0.1 and a boolean expression e defined as in Definition 2.3.1, an *annotated formula* is simply a pair

$$\text{af}(\phi, e)$$

that associates the formula to the expression. We denote

- the operation of extracting the boolean expression associated to a given formula as
$$\text{expr}(\text{af}(\phi, e)) = e$$
and then extend this notation to sequents such that expr(Δ) is the set of all boolean expressions of Δ
$$\text{expr}(\Delta) = \{\text{expr}(\phi) \mid \phi \in \Delta\}$$
- the operation of extracting the set of variables appearing in the expression e as
$$\text{vars}(\text{af}(\phi, e)) = \text{vars}(e)$$
and then extend this notation to sequents such that vars(Δ) is the set of all the variables appearing in the boolean expressions of the annotated formulae of Δ
$$\text{vars}(\Delta) = \{\text{vars}(\text{expr}(\phi)) \mid \phi \in \Delta\}$$

It is important to note that only the topmost connective gets annotated, and not the sub-formulae.

The purpose of putting formulae and expressions together in the annotated formula is twofold:

- the actions taken on the formula determine the constraints that will be generated, and these refer to the expressions associated to said formula;
- after the constraints are solved we can query the assignment of the variables and find out if the associated formula is used or not in a certain branch of a proof (see Definition 2.3.6).

The above constraints may be only of two kinds: "e avail" and "e used".

Definition 2.3.4 (Constraints)**.** Given an annotated formula $\mathrm{af}(\phi, e)$ as in Definition 2.3.3, a constraint λ is either

- "e used", which states that the formula ϕ gets consumed in this branch of the proof. This corresponds to saying the expression e is true or
$$x_i \wedge \cdots \wedge x_j \leftrightarrow \top$$
- "e avail", which states that the formula ϕ does not get consumed in this branch of the proof, and thus is available to be used in another branch. This corresponds to saying the expression e is not true or
$$x_i \wedge \cdots \wedge x_j \leftrightarrow \bot$$

We then extend to sequents, such that

$$\Delta \text{ used} = \{e \text{ used} \mid e \in \exp(\Delta)\}$$
$$\Delta \text{ avail} = \{e \text{ avail} \mid e \in \exp(\Delta)\}$$

We denote

- $\exp(\lambda) = e$ where λ is either "e used" or "e avail";
- $\mathrm{vars}(\lambda) = \mathrm{vars}(\exp(\lambda))$, and extend this notation to a set of constraints Λ
$$\mathrm{vars}(\Lambda) = \bigcup_{\lambda \in \Lambda} \mathrm{vars}(\lambda)$$

Definition 2.3.5 (Assignment)**.** An assignment is a function

$$\mathrm{V} : \{\ldots, x_i, x_j, \ldots\} \rightarrow \{\top, \bot\}$$

that associates the members of a set of variables to either true or false. Given a set of variables X, we say that the assignment V *covers* X if no variable in X is left undefined under V, or

$$X \subseteq \mathrm{Dom}(\mathrm{V})$$

Definition 2.3.6 (Evaluation)**.** Given a boolean expression e and an assignment V, such that V covers vars(e), we write

$$e[\mathrm{V}] = e[\ldots, x_i := \mathrm{V}(x_i), x_j := \mathrm{V}(x_j), \ldots]$$

as the value of the expression e substituting its variables using assignment V. We extend this notation, such that

- given a constraint λ and an assignment V that covers vars(λ)

$$\begin{cases} e \text{ used}[\mathrm{V}] = \top & \text{if } e[\mathrm{V}] = \top \\ e \text{ used}[\mathrm{V}] = \bot & \text{if } e[\mathrm{V}] = \bot \\ e \text{ avail}[\mathrm{V}] = \top & \text{if } e[\mathrm{V}] = \bot \\ e \text{ avail}[\mathrm{V}] = \bot & \text{if } e[\mathrm{V}] = \top \end{cases}$$

- given an annotated sequent Δ and an assignment V that covers vars(Δ)

$$\Delta[\mathrm{V}] = \{\phi \mid \mathrm{af}(\phi, e) \in \Delta, e[\mathrm{V}] = \top\}$$

Definition 2.3.7. Given two assignments V′ and V″ and a set of variables X, we say that V′ and V″ coincide for X – written $\mathrm{V}' \sim_X \mathrm{V}''$ – when they both map all the variables in X to the same values, or

$$\mathrm{V}' \sim_X \mathrm{V}'' \Leftrightarrow \forall x \in X \mid \mathrm{V}'(x) = \mathrm{V}''(x)$$

A judgment is considered provable if its constraints are satisfiable. Otherwise the branch of the proof fails.

Definition 2.3.8 (Satisfaiability)**.** Given a set of constraints Λ and an assignment function V that covers vars(Λ), we say that V satisfies Λ iff every constraint in Λ is true under V, or

$$\Lambda \downarrow \mathrm{V} \Leftrightarrow \bigwedge_{\lambda \in \Lambda} \lambda[\mathrm{V}] = \top$$

Figure 1 shows the triadic calculus of [1]. This has judgments with three members: a set of unrestricted formulae; a multiset of linear formulae put to the side; and either a single formula or another multiset of linear formulae. An arrow pointing up symbolizes the asynchronous phase, and an arrow pointing down the focusing phase. Phase switching happens using the decide rules ($[D_1]_\mathrm{A}$, $[D_2]_\mathrm{A}$), which non-deterministically choose a formula to focus on. We now expand the concept of triadic judgment by adding explicit constraints.

$$[\bot]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \Gamma}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \bot, \Gamma} \qquad [\top]_{\mathrm{A}}\ \frac{}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \top, \Gamma}$$

$$[\parr]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1, \phi_2, \Gamma}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1 \parr \phi_2, \Gamma} \qquad [\&]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1, \Gamma \qquad \vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_2, \Gamma}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1 \& \phi_2, \Gamma}$$

$$[\oplus_{\mathrm{L}}]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_1}{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_1 \oplus \phi_2} \qquad [\oplus_{\mathrm{R}}]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_2}{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_1 \oplus \phi_2}$$

$$[\otimes]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Gamma \Downarrow \phi_1 \qquad \vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_2}{\vdash_{\mathrm{A}} \Psi : \Gamma, \Delta \Downarrow \phi_1 \otimes \phi_2} \qquad [1]_{\mathrm{A}}\ \frac{}{\vdash_{\mathrm{A}} \Psi : . \Downarrow 1}$$

$$[\ ?\ ]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \phi, \Psi : \Delta \Uparrow \Gamma}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow ?\phi, \Gamma} \qquad [\ !\ ]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : . \Uparrow \phi}{\vdash_{\mathrm{A}} \Psi : . \Downarrow !\phi}$$

$$[I_1]_{\mathrm{A}}\ \frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \Psi : \alpha \Downarrow \alpha^{\bot}} \qquad [I_2]_{\mathrm{A}}\ \frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \alpha, \Psi : . \Downarrow \alpha^{\bot}}$$

$$[D_1]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi}{\vdash_{\mathrm{A}} \Psi : \phi, \Delta \Uparrow .} \qquad [D_2]_{\mathrm{A}}\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi}{\vdash_{\mathrm{A}} \phi, \Psi : \Delta \Uparrow .}$$

$$[R\Uparrow]_{\mathrm{A}}\ \frac{\neg\phi \text{ asy} \qquad \vdash_{\mathrm{A}} \Psi : \phi, \Delta \Uparrow \Gamma}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi, \Gamma} \qquad [R\Downarrow]_{\mathrm{A}}\ \frac{\phi \text{ negative} \qquad \vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi}$$

Figure 1: J.-M. Andreoli's triadic calculus.

Definition 2.3.9. Given any judgment in our calculus, it can be in either two forms:

- focused or in the synchronous phase, written:

$$\vdash \Psi : \Delta \Downarrow \phi \mid\mid \Lambda : \mathrm{V}$$

- in the asynchronous phase, written:

$$\vdash \Psi : \Delta \Uparrow \Gamma \mid\mid \Lambda : \mathrm{V}$$

Where

- Ψ is the same as the triadic calculus of [1]: a set of unrestricted non annotated formulae, or all formulae that can be freely discarded or duplicated.

- Δ and Γ are multi-sets of linear (annotated) formulae, these are respectively the formulas "put to the side" and the formulae which are being "worked on" during a certain moment of the asynchronous phase;
- Λ and V are the constraints and the assignment as defined in Definition 2.3.8. By adding these members we make the flow of constraints through the proof tree explicit, leaving no ambiguity to where the constraints should be checked. This approach to constraints differs from the one in [5], which prioritizes generality. The choice of letters is mainly a mnemonic one, constraints Λ "go-up" the proof tree and solutions V "come down" from the leaves.

Definition 2.3.10 (Splitting)**.** Given a sequent of annotated formulae Δ and a set of variables X such that $|\Delta| = |X|$, we define the operation of splitting it as a function

$$\mathrm{split}_X(\Delta) \mapsto (\Delta_L, \Delta_R)$$

where

$$\begin{aligned}\Delta_L &= \{\mathrm{af}(\phi_i, x_i \wedge e_i) \mid i \in \{1, \dots, n\}\} \\ \Delta_R &= \{\mathrm{af}(\phi_i, \overline{x_i} \wedge e_i) \mid i \in \{1, \dots, n\}\}\end{aligned}$$

with n the cardinality of Δ, and ϕ_i (resp. e_i) the formula (resp. the expression) of the i-eth annotated formula in Δ using an arbitrary order. The same holds for x_i and X. With a slight abuse of notation we will write Δ_L^X and Δ_R^X to mean respectively the left projection and the right projection of the pair (Δ_L, Δ_R).

As a small example for clarity, given the sequent

$$\begin{aligned}\Delta &= \mathrm{af}(a \otimes b, x_1), \mathrm{af}(c^\perp, x_2) \\ X &= \{x_3, x_4\}\end{aligned}$$

this is split into

$$\begin{aligned}\Delta_L^X &= \mathrm{af}(a \otimes b, x_3 \wedge x_1), \mathrm{af}(c^\perp, x_4 \wedge x_2) \\ \Delta_R^X &= \mathrm{af}(a \otimes b, \overline{x_3} \wedge x_1), \mathrm{af}(c^\perp, \overline{x_4} \wedge x_2)\end{aligned}$$

Figure 2 shows the calculus of [5] using our notation of explicit constraints implementing the "lazy" strategy. Figure 3 instead presents our focused constraint calculus.

Lemma 2.3.1. *For all sequents* Δ *and assignments* V *which cover* $\mathrm{vars}(\Delta) \cup X$*:*

$$\Delta_L^X[\mathrm{V}] \cap \Delta_R^X[\mathrm{V}] = \varnothing$$

$$[\parr]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1, e), \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1 \parr \phi_2, e) \parallel \Lambda : \mathrm{V}}$$

$$[\bot]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} \Delta \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\bot, e) \parallel \Lambda : \mathrm{V}} \qquad [\top]_{\mathrm{PH}}\ \frac{}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\top, -) \parallel - : -}$$

$$[\&]_{\mathrm{PH}}\ \frac{\begin{array}{l}\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}'' \\ \vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}' \qquad \mathrm{V}' \sim_{\mathrm{vars}(\Delta,\Gamma)} \mathrm{V}''\end{array}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1 \& \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

$$[\otimes]_{\mathrm{PH}}\ \frac{\begin{array}{l}\qquad\qquad \vdash_{\mathrm{PH}} \Delta_L^X, \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}' \\ X \text{ new} \qquad \vdash_{\mathrm{PH}} \Delta_R^X, \mathrm{af}(\phi_2, e) \parallel \mathrm{V}' : \mathrm{V}'' \qquad \mathrm{V}' \sim_{\mathrm{vars}(\Delta),X} \mathrm{V}''\end{array}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

$$[\oplus]_{\mathrm{PH}}\ \frac{x \text{ new} \qquad \vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1, x), \mathrm{af}(\phi_2, \overline{x}) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1 \oplus \phi_2, e) \parallel \Lambda : \mathrm{V}}$$

$$[1]_{\mathrm{PH}}\ \frac{e \text{ used}, \Lambda, \Delta \text{ avail}\!\downarrow\! \mathrm{V}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(1, e) \parallel \Lambda : \mathrm{V}} \qquad [\,!\,]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} ?\Delta, \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} ?\Delta, \mathrm{af}(!\phi, e) \parallel \Lambda : \mathrm{V}}$$

$$[\,?\,]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(?\phi, e) \parallel \Lambda : \mathrm{V}}$$

$$[W?]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} \Delta \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} ?\mathrm{af}(\phi, e), \Delta \parallel \Lambda : \mathrm{V}} \qquad [C?]_{\mathrm{PH}}\ \frac{\vdash_{\mathrm{PH}} \mathrm{af}(?\phi, e), \mathrm{af}(?\phi, e), \Delta \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash_{\mathrm{PH}} ?\mathrm{af}(\phi, e), \Delta \parallel \Lambda : \mathrm{V}}$$

$$[A]_{\mathrm{PH}}\ \frac{\alpha \text{ atom} \qquad \Lambda, e_1 \text{ used}, e_2 \text{ used}, \Delta \text{ avail}\!\downarrow\! \mathrm{V}}{\vdash_{\mathrm{PH}} \mathrm{af}(\alpha, e_1), \mathrm{af}(\alpha^{\perp}, e_2), \Delta \parallel \Lambda : \mathrm{V}}$$

Figure 2: The one sided version of the calculus from [5] with explicit constraint propagation.

Proof. This is a simple consequence of the fact that if $\phi \in \Delta_L^X[\mathrm{V}]$ there is a annotated formula $\mathrm{af}(\phi, e) \in \Delta$ such that

$$e[\mathrm{V}] = \top$$

Since e is defined as a conjunction on boolean variables, all the variables in it must evaluate to true. It is straightforward to see that if the variable added by the split in Δ_L^X is x_i, and the corresponding one in Δ_R^X is $\overline{x_i}$, then when x_i is true in the assignment V, ${x_i}^{\perp}$ is false. Hence $\phi \notin \Delta_R^X[\mathrm{V}]$. The same can be done to show that if $\phi \in \Delta_R^X[\mathrm{V}]$ then $\phi \notin \Delta_L^X[\mathrm{V}]$. ■

$$[\parr]\ \frac{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1, e), \mathrm{af}(\phi_2, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1 \parr \phi_2, e), \Gamma \parallel \Lambda : \mathrm{V}}$$

$$[\bot]\ \frac{\vdash \Psi : \Delta \Uparrow \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\bot, e), \Gamma \parallel \Lambda : \mathrm{V}} \qquad [\top]\ \frac{}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\top, -), \Gamma \parallel - : -}$$

$$[\&]\ \frac{\begin{array}{l}\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}'' \\ \vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_2, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}', \qquad \mathrm{V}' \sim_{\mathrm{vars}(\Delta,\Gamma)} \mathrm{V}''\end{array}}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1 \& \phi_2, e), \Gamma \parallel \Lambda : \mathrm{V}''}$$

$$[\,?\,]\ \frac{\vdash \phi, \Psi : \Delta \Uparrow \Gamma \parallel \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(?\phi, -), \Gamma \parallel \Lambda : \mathrm{V}}$$

$$[R\Uparrow]\ \frac{\neg\phi \text{ asy} \qquad \vdash \Psi : \mathrm{af}(\phi, e), \Delta \Uparrow \Gamma \parallel \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e), \Gamma \parallel \Lambda : \mathrm{V}}$$

(a) Asynchronous rules.

$$[\otimes]\ \frac{\begin{array}{ll} & \vdash \Psi : \Delta_L^X \Downarrow \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}', \\ X \text{ new} & \vdash \Psi : \Delta_R^X \Downarrow \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}'' \qquad \mathrm{V}' \sim_{\mathrm{vars}(\Delta), X} \mathrm{V}''\end{array}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

$$[\oplus_{\mathrm{L}}]\ \frac{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \oplus \phi_2, e) \parallel \Lambda : \mathrm{V}} \qquad [\oplus_{\mathrm{R}}]\ \frac{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \oplus \phi_2, e) \parallel \Lambda : \mathrm{V}}$$

$$[1]\ \frac{e \text{ used}, \Delta \text{ avail}, \Lambda {\downarrow} \mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(1, e) \parallel \Lambda : \mathrm{V}} \qquad [\,!\,]\ \frac{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Delta \text{ avail}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(!\phi, e) \parallel \Lambda : \mathrm{V}}$$

$$[R\Downarrow]\ \frac{\phi \text{ negative} \qquad \vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel \Lambda : \mathrm{V}}$$

(b) Synchronous rules.

$$[I_1]\ \frac{\alpha \text{ atom} \qquad e_1 \text{ used}, e_2 \text{ used}, \Delta \text{ avail}, \Lambda{\downarrow}\mathrm{V}}{\vdash \Psi : \mathrm{af}(\alpha, e_1), \Delta \Downarrow \mathrm{af}(\alpha^{\perp}, e_2) \parallel \Lambda : \mathrm{V}}$$

$$[I_2]\ \frac{\alpha \text{ atom} \qquad e \text{ used}, \Delta \text{ avail}, \Lambda{\downarrow}\mathrm{V}}{\vdash \alpha, \Psi : \Delta \Downarrow \mathrm{af}(\alpha^{\perp}, e) \parallel \Lambda : \mathrm{V}}$$

$$[D_1]\ \frac{\neg\phi \text{ atom} \qquad \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \Psi : \mathrm{af}(\phi, e), \Delta \Uparrow . \parallel \Lambda : \mathrm{V}}$$

$$[D_2]\ \frac{\neg\phi \text{ atom} \qquad x \text{ new} \qquad \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, x) \parallel e \text{ used}, \Lambda : \mathrm{V}}{\vdash \phi, \Psi : \Delta \Uparrow . \parallel \Lambda : \mathrm{V}}$$

(c) Identity and decide rules.

Figure 3: Focused constraint calculus for Linear Logic.

Lemma 2.3.2. *For all sequents* Δ *and assignments* V *which cover* $\mathrm{vars}(\Delta) \cup X$*,*

$$\Delta[\mathrm{V}] = \Delta_L^X[\mathrm{V}] \cup \Delta_R^X[\mathrm{V}]$$

Proof. The simpler side is $\Delta_L^X[\mathrm{V}] \cup \Delta_R^X[\mathrm{V}] \subseteq \Delta[\mathrm{V}]$, since it holds by the definition of the split (Definition 2.3.10). For the other side, suppose there was a formula ϕ such that $\phi \in \Delta[\mathrm{V}]$ and $\phi \notin \Delta_L^X[\mathrm{V}] \cup \Delta_R^X[\mathrm{V}]$. This means that for some variable x_i

$$\begin{aligned} \mathrm{af}(\phi, e) \in \Delta &\Rightarrow e[\mathrm{V}] = \top \\ \mathrm{af}(\phi, x_i \wedge e) \notin \Delta_L^X &\Rightarrow x_i \wedge e[\mathrm{V}] = \bot \\ \mathrm{af}(\phi, \overline{x_i} \wedge e) \notin \Delta_R^X &\Rightarrow \overline{x_i} \wedge e[\mathrm{V}] = \bot \end{aligned}$$

But either x_i or $\overline{x_i}$ must be true in a certain assignment, thus either $x_i \wedge e$ or $\overline{x_i} \wedge e$ must be true, contradicting the hypothesis. ■

Theorem 2.3.1 (Soundness). *For any judgment in our constraint calculus:*

- *if* $\vdash \Psi : \Delta \Uparrow \Gamma \parallel \Lambda : \mathrm{V}$*, then* $\vDash ?\Psi, \Delta, \Gamma$
- *if* $\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel \Lambda : \mathrm{V}$*, then* $\vDash ?\Psi, \Delta, \phi$

Proof. As a soundness proof we give a translation from our constraint calculus to A: the triadic calculus of [1] defined in Figure 1. We will show that

$$\begin{aligned} \vdash \Psi : \Delta \Uparrow \Gamma \parallel \Lambda : \mathrm{V} &\Rightarrow \vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \Gamma[\mathrm{V}] \\ \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel \Lambda : \mathrm{V} &\Rightarrow \vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi \end{aligned}$$

The four base cases are proved essentially in the same way, with the exception of top ($\top$):

(I_1) : Given the rule $[I_1]$

$$[I_1]\,\frac{\alpha \text{ atom} \qquad e_1 \text{ used}, e_2 \text{ used}, \Delta \text{ avail}, \Lambda{\downarrow}\mathrm{V}}{\vdash \Psi : \mathrm{af}(\alpha, e_1) \Downarrow \mathrm{af}(\alpha^\perp, e_2), \Delta \parallel \Lambda : \mathrm{V}}$$

looking at the constraints we get that

$$\begin{aligned} \Delta[\mathrm{V}] &= \varnothing && \text{(Because of } \Delta \text{ avail)} \\ \mathrm{af}(\alpha, e_1)[\mathrm{V}] &= \alpha && \text{(Because of } e_1 \text{ used)} \\ \mathrm{af}(\alpha^\perp, e_2)[\mathrm{V}] &= \alpha^\perp && \text{(Because of } e_2 \text{ used)} \end{aligned}$$

so we can rewrite this as

$$[I_1]\,\frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \Psi : \alpha \Downarrow \alpha^\perp}$$

(I_2) : Given the rule $[I_2]$

$$[I_2]\,\frac{\alpha \text{ atom} \qquad e \text{ used}, \Delta \text{ avail}, \Lambda{\downarrow}\mathrm{V}}{\vdash \Psi, \alpha : \Delta \Downarrow \mathrm{af}(\alpha^\perp, e) \parallel \Lambda : \mathrm{V}}$$

proceeding as above we get

$$\begin{aligned} \Delta[\mathrm{V}] &= \varnothing \\ \mathrm{af}(\alpha^\perp, e)[\mathrm{V}] &= \alpha^\perp \end{aligned}$$

thus

$$[I_2]\,\frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \alpha, \Psi : . \Downarrow \alpha^\perp}$$

(1) : Given the rule $[1]$

$$[1]\,\frac{e \text{ used}, \Delta \text{ avail}, \Lambda{\downarrow}\mathrm{V}}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(1, e) \parallel \Lambda : \mathrm{V}}$$

proceeding as above we get

$$\begin{aligned} \Delta[\mathrm{V}] &= \varnothing \\ \mathrm{af}(1, e)[\mathrm{V}] &= 1 \end{aligned}$$

thus

$$[1]\,\frac{}{\vdash_{\mathrm{A}} \Psi : . \Downarrow 1}$$

($\top$) : Given the rule [$\top$] we have that

$$\frac{}{\vdash \Psi : \Delta \Uparrow \mathrm{af}(\top, -), \Gamma \parallel - : -}$$

Thus we can choose whichever assignment V that covers vars(Δ, Γ), and obtain

$$\frac{}{\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \top, \Gamma[\mathrm{V}]}$$

The induction then follows like this:

($\otimes$) : Given the rule [$\otimes$], we apply the inductive hypothesis, and from the premises

$$\vdash \Psi : \Delta_L^X \Downarrow \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}'$$
$$\vdash \Psi : \Delta_R^X \Downarrow \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}''$$

we extract the triadic proofs

$$\vdash_{\mathrm{A}} \Psi : \Delta_L^X[\mathrm{V}'] \Downarrow \phi_1$$
$$\vdash_{\mathrm{A}} \Psi : \Delta_R^X[\mathrm{V}''] \Downarrow \phi_2$$

Since $\mathrm{V}' \sim_{\mathrm{vars}(\Delta), X} \mathrm{V}''$, this can be rewritten as

$$\vdash_{\mathrm{A}} \Psi : \Delta_L^X[\mathrm{V}''] \Downarrow \phi_1$$
$$\vdash_{\mathrm{A}} \Psi : \Delta_R^X[\mathrm{V}''] \Downarrow \phi_2$$

Furthermore because of Lemma 2.3.1 we have that $\Delta_L^X[\mathrm{V}'']$ and $\Delta_R^X[\mathrm{V}'']$ are disjoint, so the contexts of the two branches are separated. Hence we can apply $[\otimes]_{\mathrm{A}}$, and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta_L^X[\mathrm{V}''], \Delta_R^X[\mathrm{V}''] \Downarrow \phi_1 \otimes \phi_2$$

which, because of Lemma 2.3.2, correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''$$

should be mapped to.

(&) : Given the rule [&], we apply the inductive hypothesis, and from the premises

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}'$$
$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_2, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}''$$

we extract the triadic proofs

$$\vdash \Psi : \Delta[\mathrm{V}'] \Uparrow \phi_1, \Gamma[\mathrm{V}']$$
$$\vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_2, \Gamma[\mathrm{V}'']$$

Now, since $\mathrm{V}' \sim_{\mathrm{vars}(\Delta,\Gamma)} \mathrm{V}''$, we can write

$$\vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_1, \Gamma[\mathrm{V}'']$$
$$\vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_2, \Gamma[\mathrm{V}'']$$

this way we can apply $[\&]_{\mathrm{A}}$ to obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_1 \,\&\, \phi_2, \Gamma[\mathrm{V}'']$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1 \,\&\, \phi_2, e), \Gamma \parallel \Lambda : \mathrm{V}''$$

should be mapped to.

$(\invamp)$: Given the rule $[\invamp]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1, e), \mathrm{af}(\phi_2, e), \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \phi_1, \phi_2, \Gamma[\mathrm{V}]$$

Then we apply $[\invamp]_{\mathrm{A}}$, and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \phi_1 \invamp \phi_2, \Gamma[\mathrm{V}]$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi_1 \invamp \phi_2, e), \Gamma \parallel \Lambda : \mathrm{V}$$

should be mapped to.

$(\oplus_{\mathrm{L}})$: Given the rule $[\oplus_{\mathrm{L}}]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1, e) \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1$$

Then we apply $[\oplus_{\mathrm{L}}]_{\mathrm{A}}$, and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1 \oplus \phi_2$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \oplus \phi_2, e) \parallel \Lambda : \mathrm{V}$$

should be.

$(\oplus_{\mathrm{R}})$: Given the rule $[\oplus_{\mathrm{R}}]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_2, e) \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi_2$$

Then we apply $[\oplus_{\mathrm{R}}]_{\mathrm{A}}$, and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1 \oplus \phi_2$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \oplus \phi_2, e) \parallel \Lambda : \mathrm{V}$$

should be.

$(\,!\,)$: Given the rule $[\,!\,]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Delta \text{ avail}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : . \Uparrow \phi$$

since $\Delta[\mathrm{V}] = \varnothing$ under assignment V. We then apply $[\,!\,]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \Psi : . \Downarrow !\phi$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(!\phi, e) \parallel \Lambda : \mathrm{V}$$

should be mapped to

(?) : Given the rule [?], we apply the inductive hypothesis, and from the premise

$$\vdash \phi, \Psi : \Delta \Uparrow \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \phi, \Psi : \Delta[\mathrm{V}] \Uparrow \Gamma[\mathrm{V}]$$

Then we apply $[\,?\,]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow ?\phi, \Gamma[\mathrm{V}]$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(?\phi, e), \Gamma \parallel \Lambda : \mathrm{V}$$

should be mapped to.

$(\bot)$: Given the rule $[\bot]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Uparrow \Gamma \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \Gamma[\mathrm{V}]$$

Then we apply $[\bot]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \bot, \Gamma[\mathrm{V}]$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\bot, e), \Gamma \parallel \Lambda : \mathrm{V}$$

should be mapped to.

(D_1) : Given the rule $[D_1]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi$$

Then we apply $[D_1]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \Psi : \phi, \Delta[\mathrm{V}] \Uparrow .$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \mathrm{af}(\phi, e), \Delta \Uparrow . \parallel \Lambda : \mathrm{V}$$

should be mapped to.

(D_2) : Given the rule $[D_2]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, x) \parallel x \text{ used}, \Lambda : \mathrm{V}$$

where x new, we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi$$

Then we apply $[D_2]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \phi, \Psi : \Delta[\mathrm{V}] \Uparrow .$$

which correctly corresponds to what the conclusion

$$\vdash \phi, \Psi : \Delta \Uparrow . \parallel \Lambda : \mathrm{V}$$

should be mapped to.

$(R\Uparrow)$: Given the rule $[R\Uparrow]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \mathrm{af}(\phi, e), \Delta \Uparrow \Gamma \parallel \Lambda : \mathrm{V}$$

when we extract the triadic proof, two cases arise:

- ϕ disappears under assignment V, and we get

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \Gamma[\mathrm{V}]$$

  which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e), \Gamma \parallel \Lambda : \mathrm{V}$$

should be mapped to, since if ϕ disappears from the premise, it will also disappear from the conclusion.

– ϕ remains under assignment V, and we get

$$\vdash_{\mathrm{A}} \Psi : \phi, \Delta[\mathrm{V}] \Uparrow \Gamma[\mathrm{V}]$$

Then we apply $[R{\Uparrow}]_{\mathrm{A}}$ and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \phi, \Gamma[\mathrm{V}]$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e), \Gamma \parallel \Lambda : \mathrm{V}$$

should be mapped to.

$(R{\Downarrow})$: Given the rule $[R{\Downarrow}]$, we apply the inductive hypothesis, and from the premise

$$\vdash \Psi : \Delta \Uparrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : \mathrm{V}$$

we extract the triadic proof

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Uparrow \phi$$

Then we apply $[R{\Downarrow}]_{\mathrm{A}}$, and obtain

$$\vdash_{\mathrm{A}} \Psi : \Delta[\mathrm{V}] \Downarrow \phi$$

which correctly corresponds to what the conclusion

$$\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel \Lambda : \mathrm{V}$$

should be mapped to.

■

Chapter 3

Implementation

We now describe the main implementation details of our prover. During this section, to distinguish the variables from 2.3.1 from Prolog's ones, we will always refer to the latter as Prolog variables. When explaining the code we will use some common names for Prolog variables:

- `A` is a set of unrestricted atoms, which purpose will be explained in Section 3.4.1;
- `U` is a set of unrestricted formulae; this more or less corresponds to Ψ from §2;
- `F`, `F1`, ..., are formulae, and `Fs` and `D` are a lists of them; these more or less correspond to Δ and Γ from §2;
- `S` is the queue of currently usable unrestricted formulae, which purpose will be explained in Section 3.4.2;
- `L` is a list of constraints.

3.1 Why Prolog

Prolog as a language and as an environment has been historically tied to automated theorem proving for its ability to express backtracking algorithms naturally. Most Prolog implementations also support CLP (constraint logic programming) through dedicated libraries. These allow to express constraints referring some attributes of Prolog variables in the body of the clauses; in our case we will use CLP($\mathcal{B}$) [8], which provides tools to deal with boolean constraints. Which, in this context, are expressions made of Prolog variables and the constants 1 and 0, respectively true and false. The allowed operators are the usual ones one would expect; in our case we will use exclusively conjunction, negation and equality, respectively:

```
X * Y.
~ X.
X =:= Y.
```

Usually constraints will be accumulated in a list, for this reason CLP($\mathcal{B}$) provides the functor `*(L)` to express the conjunction of its members. The main predicate of the library is `sat/1`, which checks for the satisfiability of a constraint. Since we only deal with conjunctions we define the helper predicate `check/1`

```
check(L) :-
  sat(*(L)).
```

The better understand how the predicate works, we give some examples:

```
?- check([X * Y =:= 1, X =:= 0]).
false.
?- check([X * Y =:= 0]).
sat(1#X*Y).
?- check([X * Y =:= 0, X =:= 1]).
X = 1,
Y = 0.
```

Here three different outcomes can be seen:

- the first case is unsatisfiable, so the predicate fails;
- the second case is not instantiated enough and so the constraints just get reduced;
- the third case shows how one constraint (`X =:= 1`) can force a Prolog variable to be unified to a value.

It is important to point out, altought abvious, that constraint satisfaction also unifies the Prolog variables to their assigned value. This mechanism is used to implcitly deal with the propagation of solutions.

We use the library with the flag `clpb_monotonic` set to `true`. This makes the algorithm many orders of magnitude faster but mandates that all Prolog variables be wrapped by the functor `v/1`. This small explanation sums up the extent of the library we use in our prover.

3.2 Formula transformations

Before beginning the proof a sequent passes through a number of transformations. These transformations both preprocess the sequent to a more convenient form, and also add information about the sub-formulae.

As a first transformation the sequent gets normalized into a sequent in negated normal form (NNF) as defined in Definition 2.1.1. Normalization is a common technique – used in all the provers we compare with. The implementation mirrors exactly the transformation from two-sided judgment to one-sided judgment of Section 2.1:

1. the left sequent is negated and appended to the right sequent, implemented by the predicate `negate_premises/3`;
2. the predicate `nnf/2`, which encodes the DeMorgan rules, is mapped recursively over the new sequent.

From an implementation point of view the purpose of normalization is to reduce the number of available choices the prover has at a certain moment, although by doing so we sacrifice some of the structure of the formula.

Next, to each formula we assign its why-not height, a measure borrowed from APLL's implementation.

Definition 3.2.1 (Why-not height)**.** Why-not height is the maximum number of nested "why-not"s in a formula, or

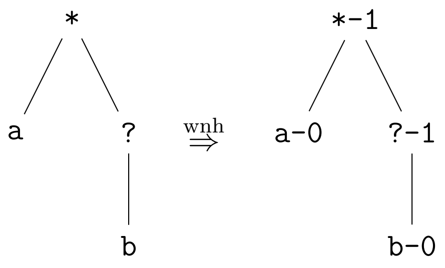
$$\mathrm{wnh}(\phi) = \begin{cases} 0 & \text{if } \phi \in \{\bot, \top, 1, 0\} \\ \max(\mathrm{wnh}(\phi_1), \mathrm{wnh}(\phi_2)) & \text{if } \phi \in \{\phi_1 \otimes \phi_2, \phi_1 \parr \phi_2, \phi_1 \oplus \phi_2, \phi_1 \,\&\, \phi_2\} \\ \mathrm{wnh}(\phi_1) & \text{if } \phi \in \{{\phi_1}^{\perp}, !\phi_1\} \\ 1 + \mathrm{wnh}(\phi_1) & \text{if } \phi \in \{?\phi_1\} \end{cases}$$

The purpose of this attribute is to guide the prover to the reasonably simpler choice – the one with the least nested exponentials – at different times during proof search. This happens in three ways:

- When a rule generates two branches ($\otimes$, $\&$), the branch associated to the formula with the least why-not height is tried first. This will presumably make the prover choose the simpler branch of the two, making it so we can fail early if the branch turns out to be false. An example of this is Section **??**.
- In the case of plus ($\oplus$), the branch associated to the least why-not height is tried first; this makes it so we can continue if the branch turns out to be true, ignoring the other harder branch.

- During the D_2 rule, the exponentials are tried in order of ascending why-not height. This process is further explained in Section 3.4.2.

After this transformation formulae are attribute trees with at each node the why-not height. For example the formula `*(a, ?(b))` becomes `*(a-0, ?(b-0)-1)-1`, or



As a third and final transformation, each formula gets annotated as in Definition 2.3.3. The process of annotation is implemented by the predicate `annotate/3`. This also returns the initial constraints, which set each formula in the sequent to “used”; thus stating that in the proof each and every formula must be used.

3.3 Helper predicates

The prover needs a series of helper predicates ranging from predicates that just implement ϕ asy from Definition 2.2.1, to ones which aid the generation of constraints. In particular we now cover the implementation of the splitting function from Definition 2.3.10

```
%! split_ctx(+[AFs], -[AFs], -[AFs], -[Cns], -[Cns]) is det.
split_ctx(Afs, Pos, Neg, PCns, NCns) :-
  maplist([ af(F, N, E)
          , af(F, VarPos, Y)
          , af(F, VarNeg, Z)
          , v(Y) =:= v(X) * v(E)
          , v(Z) =:= (~ v(X)) * v(E)
          ]>>(
               gensym(x, V),
               atomic_list_concat([N, V], '.', VarPos),
               atomic_list_concat([N, V], '.~', VarNeg)
  ), Afs, Pos, Neg, PCns, NCns).
```

The code itself is nothing special, consisting of a simple map over the list of formulae generating the constraints. What is important is that the snippet above shows the clear distinction between variable names – represented by atoms (lines 10-11), and the value of a variable – represented by Prolog variables (lines 4-5). This separation of name and value is needed because – after checking the constraints – the solver unifies the Prolog variables to their values if it finds a valid assignment; the purpose of the atom is then to associate the variable value to its name if the final proof tree.

Finally, it must be noted that there is no predicate implementing the concept of two coinciding assignments (Definition 2.3.7): this – as mentioned before – is implicitly handled by Prolog's unification mechanism.

3.4 Focusing

Focusing is implemented by two mutually defined predicates: `async/8` and `focus/8`, implementing respectively the asynchronous and the synchronous rules of Figure 3. The structure of these two predicates reflects almost exactly that of the sequents.

During the asynchronous phase we have two lists: `Fs` and `D`, representing Γ and Δ. The asynchrnous rules always work on the head of `Fs`, breaking it down into its sub-formulae. This process can be seen for example in the predicate for with (&):

```
1 async(A, U, D, [F|Fs], S, M, L) :-
2   F = af(((F1-H1) & (F2-H2))-_), N, E), !,
3   ( H2 > H1
4   -> async(A, U, D, [af((F1-H1), N, E)|Fs], S, M, [v(E) =:= 1|L]),
5      async(A, U, D, [af((F2-H2), N, E)|Fs], S, M, [v(E) =:= 1|L])
6   ;  async(A, U, D, [af((F2-H2), N, E)|Fs], S, M, [v(E) =:= 1|L]),
7      async(A, U, D, [af((F1-H1), N, E)|Fs], S, M, [v(E) =:= 1|L])
8   ).
```

Here, we chose this particular rule to also show how the prover is guided using why-not height (line 3) as mentioned in Definition 3.2.1. Compare this with the & rule in Figure 3. The cut at line 2 is important to guarantee the determinism of the asynchronous phase and is present in all the rules for the negative connectives.

If a formula cannot be further be broken apart – i.e. it is either an atom, a negated atom, or it has a top-level synchronous connective – then it is put to the side in `D`. This process goes as long as `Fs` is not empty.

When each formula is moved from `Fs` to `D`, the phase switches and the focusing process begins: a formula is chosen from either `D` or `U` by applying one of the decide rules. This formula gets broken down until either an asynchronous connective or a negated atom is left. Unlike the asynchronous phase, the rules applied in this phase are non-deterministic and may be backtracked. Other than this the implementation of the predicate `focus/8` is almost the same as `async/8`: the formula is broken down into its sub-formulae, and – if necessary – the why not heights are used to guide the proof searc. The decide rules will be discussed further ahead in Section 3.4.2.

3.4.1 Identity rules

This process of alternating asynchronous and synchronous phases in classic focusing goes on until only a positive literal (in our case a negated atom) in `Fs` is left; and the corresponding negative literal (in our case just an atom) in either `U` or `D`. When this happens the axioms – rules I_1 or I_2 – are applied to close the branch of the proofs. In our case when we are focusing and we have a positive literal in `Fs`, we check if the corresponding negative literal exists in `D`. If this is true, then the variables of all the other formulae in `D` are set to zero using the predicate `set_to_zero/2` defined in Section 3.3, and the constraints are checked. This is encoded by the clause

```
1 focus(A, U, D, F, _, _, L) :-
2   F = af(((~ T)-_), _, E1),
3   is_term(T),
4   select(af((T-_), _, E2), D, D1),
5   set_to_zero(D1, Dz),
6   append([v(E1) =:= 1, v(E2) =:= 1|Dz], L, Cns),
7   check(Cns).
```

where `set_to_zero/2` (line 4) is to be interpreted as − avail. A slightly different approach is taken if instead a correspondence is found in `A` instead of `D`. Here `A` is a special set containing just unrestricted atoms. This is a small modification to APLL's algorithm based on the fact that once negative literals are put in a sequent they can never leave it.

3.4.2 Decide rules

For the decide rules, particularly for D_2, we use a modified version of APLL's algorithm defined in Section 4.1. The method consists of not using directly the set Ψ in the D_2 rule, but instead a queue of ordered unrestricted formulae which can be refilled only a certain number of times per-branch. This can be seen in the definition of the rule `decide_2` for the `async/8` predicate

```
async(A, U, D, [], [H|T], M, L) :-
  \+ U = [],
  gensym(x, X),
  focus(A, U, D, af(H, X, E), T, M, [v(E) =:= 1|L]).
async(A, U, D, [], [_|T], M, L) :-
  \+ U = [],
  async(A, U, D, [], T, M, L).
async(A, U, D, [], [], M, L) :-
  \+ U = [],
  refill(U, M, S, M1),
  early_stop(A, U, D, S, M1, L).
```

Here the fifth argument is the queue and the sixth is the bound. Two cases arise:

- if `S = []` and `M > 0` then the sequent of unrestricted formulae `U` is taken and it is sorted based on why-not height. This can be seen in the predicate `refill/4`

  ```
  refill(U, M, S, M1) :-
    \+ M = 0,
    \+ U = [],
    sort(2, @=<, U, S),
    M1 is M - 1.
  ```

  This new list of unrestricted formulae becomes the new `S` and `M` is decreased. Otherwise if if `M` is 0 (line 2) the branch fails.

- if `S` is not empty, then the first formula in the queue is extracted and added to the working set. If the branch fails the formula gets discarded and the next one in the queue is tried.

In particular, if the queue `S` is refilled, we do not directly call `async/8`, but instead call the predicate `early_stop/7` (line 11), defined as:

```
early_stop(_, _, _, [], _, _) :-
  false.
early_stop(A, U, D, [H|T], M, L) :-
  gensym(x, X),
  focus(A, U, D, af(H, X, E), T, M, [v(E) =:= 1|L]).
early_stop(A, U, D, [_|T], M, L) :-
  early_stop(A, U, D, T, M, L).
```

This is due the simple fact that if the branch was not provable and we instead called directly `async/8` at line 11, we would try to refill the branch `M` times. What `early_stop/7` does is fail if the queue has just been refilled and it turns out the branch was not provable.

All the rules D_1, I_1 are defined before the unrestricted counterparts, so that they are tried first.

3.5 Building the tree

In the listings above we omitted one parameter from the calls to `focus/8` and `async/8`, which purpose is to build the proof tree. For example in the clause for par ($\parr$)

```
async(A, U, D, [F|Fs], S, M, L, node(par, A, U, D, [F|Fs], [T])) :-
  F = af(((F1 / F2)-_), N, E), !,
  Fs1 = [af(F1, N, E), af(F2, N, E)|Fs],
  async(A, U, D, Fs1, S, M, [v(E) =:= 1|L], T).
```

we can see clearly the structure of one node of the tree: a label, the context and an – optionally empty – list of sub-trees. A leaf is just a node with an empty list of sub-trees.

This term can be used in the end to reconstruct the actual proof tree by visiting it and – for each formula of each node – querying whether its variable is set to one, deleting it otherwise (this process is the same used in the proof for Theorem 2.3.1). A classic proof tree without the focusing infrastructure may be built by removing all the nodes regarding the phases (i.e. $R\Downarrow$ $R\Uparrow$ and decide rules) and by rebuilding the original sequent by appending `A`, `U`, `D` and `Fs` together as explained in [1]. A more sophisticated algorithm may even cancel out unwanted unrestricted formulae, that otherwise remain lingering in the sequent.

Chapter 4

State of the art

Most bottom-up provers for classic linear logic use some combination of focusing and normalization to structure their proofs, with the notable exception of llprover not using normalization. We confront our prover with two other provers: llprover (1997,) and APLL (circa 2019,).

Usually the splitting is handled in two ways: trying every partition possible, or using something called the input/output method. The latter tries to do one branch of the proof of a multiplicative, and then feeds the remaining formulae in the sequent of the other branch.

We now take a deeper look at the provers we confront with.

4.1 APLL

APLL is the underlying prover of click&collect. It provides 4 different searches – forward and backwards for classic and intuitionistic linear logic. We will focus on the backwards algorithm for classic linear logic.

The program is written in OCaML and implements a standard focused proof search on normalized formulae as seen in [6]. In this section we will illustrate two noteworthy characteristics of its implementation:

- Sequent splitting when encountering a tensor is done by generating all the numbers up to $2^{|\Delta|}$ – where Δ is the sequent – and using the bit representation of those to create the two subsets. This can be seen in the function `split_list`, which in turn calls `split_list_aux`

```
let rec split_list_aux (acc1, acc2) l k = match l with
  | [] -> acc1, acc2
  | hd :: tl ->
      if k mod 2 = 0
```

```
    then split_list_aux (acc1, hd :: acc2) tl (k / 2)
    else split_list_aux (hd :: acc1, acc2) tl (k / 2)
```

where the argument `k` is the number that determines the decomposition of the sequent. This function is called recursively when a tensor is encountered during proof search, starting at $k = 2^{|\Delta|}$ and decreasing by one at each iteration

```
(* ... *)
| Tensor (g, h) ->
  let rec split_gamma k =
    if k = -1 then None
    else
      let gamma1, gamma2 = split_list gamma k in
        try
          (* ... *)
        with NoValue ->
          split_gamma (k - 1)
  in
    let k = fast_exp_2 (List.length gamma) - 1 in
      (* ... *)
```

As we will see in 5.2 this implementation choice will result in a degradation of performance on formulae with a high number of multiplicatives.

- Their particular usage of why-not height and implementation of the decide rule has been already described in §3. For the sake of clarity though, the prover uses a fixed local bound. This bound represents the number of times per branch the unrestricted formulae can be copied in the queue.

4.2 `llprover`

`llprover` is a prover written in Prolog by Naoyuki Tamura. Where APLL had different provers for implicative and classical linear logic, this prover encodes all the rules as the same predicate `rule/6`, using the first argument as a selector for the system. Choosing full classical logic as the system uses all the rules, included the ones for implicative linear logic. For this reason the prover does normalize; instead the option is given to transform the two-sided proof to a one-sided one.

Tensor splitting is implemented similarly to APLL by trying every possible partition

```
rule([ill,0], no,  r(*), S, [S1, S2], [r(N),r(N1),r(N2)]) :-
  match(S,  ([X]-->[Y1,[A*B],Y2])),
3   merge(X1, X2, X),
4   merge(Y11, Y12, Y1),
5   merge(Y21, Y22, Y2),
6   match(S1, ([X1]-->[Y11,[A],Y21])),
7   match(S2, ([X2]-->[Y12,[B],Y22])),
8   length(Y1, N), length(Y11, N1), length(Y12, N2).
```

Here `merge/3` (lines 3, 4 and 5) – called with the only the third argument bound – generates all possible lists that when merged together return the original sequents.

Another particular characteristic of llprover is that it uses a local bound with iterative deepening: llprover will try to prove the formula with bounds $1, 2, \ldots$ up to the maximum specified. This guarantees finding the simplest proof, at the expense of the overall performance.

Chapter 5

Testing

5.1 Infrastructure

The infrastructure for the project is divided in two parts:

- the nix "glue",
- some python utilities to call and confront provers.

In nix everything must be packaged as a derivation: a recipe specifying all the needed programs (as other derivations) and the steps needed to build the current one. These steps are then executed in a environment containing only the specified programs and little else.

Derivations are written for all the programs we interact with:

- the provers, such as APLL or llprover;
- the formula generator adapted from APLL described in Section 5.1.3;
- the LLTP parser.

All these, together with one for the python environment, are then used to define a single development environment to run the jupyter notebook with all the necessary dependencies. Furthermore, having used "flakes" (a nix experimental feature), all dependencies are locked to a certain commit thus ensuring better reproducibility.

The main logic for the benchmarking and testing is defined in the python module `testprover.py`. Here for simplicity we assume that the executable of a prover returns a code of 0 if it has found a proof, or any other number otherwise. This condition is already met by our prover and by llprover; for APLL we instead provide a wrapper.

Defining the call to our prover using this library is as simple as writing

```
1 pc = Registered()
2
3 @pc.register('sat-ll')
4 @call_prover(PrefixTree.SAT_LL_DICT)
5 def call_sat_prover(premises, conclusions):
6   return [ 'sat-ll'
7     , '-b', '3'
8     , f'{premises} |- {conclusions}'
9     ]
```

The function defining a call to a prover must take as inputs two arguments, the list of premises and the list of conclusions (using the format described in Section 5.1.1), and return the command calling the prover as a list. The innermost decorator (line 3) then accepts a dictionary, and returns a function which automatically calls the prover above with a test entry (as in Section 5.1.2), times it, and eventually terminates the process if it takes too much. Lastly, the outermost decorator (line 4) simply adds an entry `'sat-ll'` associated with the prover call to the register `pc`. This register is just an association name to function, and it is what is actually passed to the benchmarking functions which will use the names in the output tables.

The library then provides two functions:

- `testall` accepts a single prover and a test suite; the function checks if the output of the prover corresponds to the expected value for the test;
- `benchmark` accepts a register and a test suite and returns the times and outcomes of each prover. This function is used to compare different provers.

All the outputs of the functions above are pandas `DataFrame`s, which means that they can be easily queried, dumped to csv, aggregated, or visualized using most plotting libraries. For example let `out` be the output of a call to `testall` with some prover and test suite:

```
out['outcome'].map(lambda x: x == Outcome.SUCCESS).all()
len(out[(out['outcome'] == Outcome.FAILURE)
        | (out['outcome'] == Outcome.TIMEOUT)])
```

test respectively whether all the tests succeeded and the number of failed tests.

5.1.1 Prefix format

Since for benchmarking we will interface with a lot of different provers – each with its own syntax for expressions – the need arose for a common format which was easy to parse and translate. For this purpose we define a prefix format for linear logic formulae inspired by the format used by [7] for implicational formulae:

formula	symbol
$\phi_A \otimes \phi_B$	`*AB`
$\phi_A \parr \phi_B$	`|AB`
$\phi_A \oplus \phi_B$	`+AB`
$\phi_A \,\&\, \phi_B$	`&AB`
$\phi_A \multimap \phi_B$	`@AB`
${\phi_A}^\perp$	`^A`
$?\phi_A$	`?A`
$!\phi_A$	`!A`

Each single character not representing an operator is considered as a variable name. Longer names can be specified by enclosing them in single quotes as in `'varname'`. As an example we give the translation of DeMorgan for the tensor:

$$\text{trans}((a \otimes b)^\perp \multimap a^\perp \parr b^\perp) = \texttt{@\^{}ab|\^{}a\^{}b}$$

5.1.2 File formats

We use json as a standard format to store the tests, because of its vast adoption by most programming languages. A test suite is thus defined as a list of test cases; where a test case is just an object with these three mandatory fields:

id	is a number with the sole purpose of tracing back the test case from the output;
premises	is a list of premises as prefix formulae;
conclusions	is a list of conclusions as prefix formulae.

For example

```
{
        "id": 1,
        "premises": [ "^*ab" ],
        "conclusions": [ "|^a^b" ]
}
```

is the test case representing $(a \otimes b)^\perp \vdash a^\perp \parr b^\perp$. Other arbitrary fields may be present; for example we will use the following optional fields:

thm	tells whether this test case is a tautology or not, may be null;
***, +, ?, ...**	is the number of times a specific connective appears in the test case;
notes	is a human readable string about the test case, e.g. its infix representation;
size	is an indicative number of the size of the formula;
atoms	is the upper bound on the number of atoms.

The fields `size` and `atoms` will be further explained in Section 5.1.3.

5.1.3 Formula generation

One of the sources of formulae we'll use in Section 5.2 is APLL's random formula generator. The version we'll use is a slight modification of it, where:

- the output is in the json format described in Section 5.1.2;
- one can whether choose to generate normalized formulae or not;
- one can choose which connectives appear in the generated formula.

A noteworthy detail is how the parameters `size` and `atoms` mentioned in Section 5.1.2 are defined, as these are directly related to how the formulae are generated:

- when one specifies a number of atoms, the generator initializes an array containing that number of atoms, their negations, and the constants $\perp, \top, \ldots$. During the generation of the formula this array is randomly accessed, choosing an element when needed. This means that `atoms` represents an upper bound to the number of different atoms that may appear in the formula, not their exact number.
- when a formula is generated, at each step it is chosen whether to generate a unary or binary connective based on a threshold:
  - if a unary connective is chosen, the process continues with a size of
    $$\texttt{size} - 1$$
  - if a binary connective is chosen, the program chooses a random value between 0 and `size`, and it generates the two branches of the formula, with size respectively k and $\texttt{size} - k$.

  This means that `size` is just an indicative value of the number of connectives in the formula.

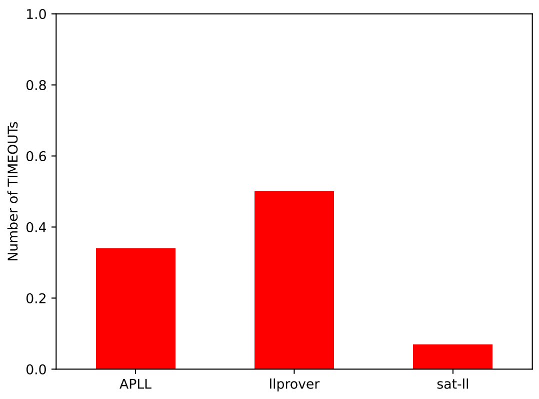
5.2 Benchmarking

We'll mainly use three sources for formulae: llprover's tests; LLTP [?], especially the translations of Kleene's intuitionistic formulae; and randomly generated formulae made by the generator described in 5.1.3. Out of these, llprover's tests are composed primarily of simple linear logic tautologies, such as the DeMorgan rules; for this reason these tests are just used to quickly check for obvious bugs.

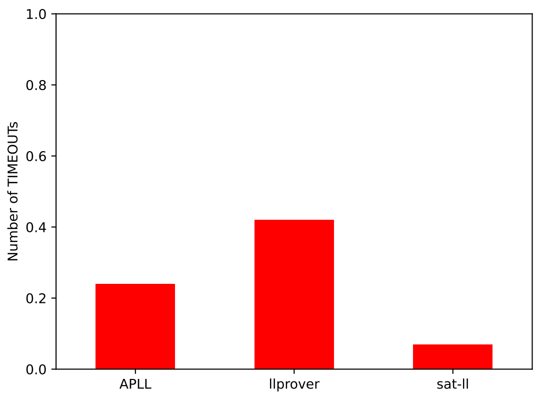
Randomly generated formulae will be used only for non-exponential test for the following reasons:

- datasets of linear logic theorems without exponentials are rare and most often the formulae in them do not have a significant size;
- when dealing with exponential formulae a prover may wrongfully deem a true formula false, just because it exhausted its bound. This problem is exacerbated when dealing generated tests: since the expected output is not known we do not know if the prover terminated because of the bound, or if the test is actually a non-theorem.

Using random formulae we can see that our prover outperforms APLL (and llprover) when dealing with formulae rich in multiplicatives (Figure 4a and Table 1a).



(a) Multiplicative case



(b) Multiplicative and additive case

Figure 4: Percentage of number of timeouts out of a hundred formulae

We can also see that in the multiplicative and additive case the difference begin to level (Figure 4b and Table 1b). The additive case is not that significant as the formulae remain manageable and no major differences can be seen. When reviewing

prover	**timeouts**	**successes**	**success rate**	**avg. time (succ.)**
APLL	34	66	0.66	1.441 s
llprover	50	50	0.50	3.006 s
sat-ll	7	93	0.93	1.874 s

(a) Data corresponding to Figure 4a

prover	**timeouts**	**successes**	**success rate**	**avg. time (succ.)**
APLL	24	76	0.76	1.213 s
llprover	42	58	0.58	0.807 s
sat-ll	7	93	0.93	1.289 s

(b) Data corresponding to Figure 4b

Table 1: Non exponential tests.

these outputs it is important to remember that randomly generated tests lack the structure of real formulae, and this may impact the quality of the measurement. All these test were done generating suites of 100 tests using a timeout of 60 s:

- Figure 4a's tests were composed of formulae of `size` 100, and `atoms` 50, using just tensor ($\otimes$) and par ($⅋$) as connectives;
- Figure 4b's tests were composed of formulae of `size` 500, and `atoms` 250, with all the connectives, except the exponentials ($\otimes$, $⅋$, $\&$ and $\oplus$)

We now show the results of running the provers on two datasets: KLE-cbn and KLE-cbv, respectively the call-by-name and call-by-value translations of Kleene's theorems. These translations introduce a high number of exponentials; this causes – as said before – some failures not due to bugs, but because of the the exhausted number of contractions. The benchmarks are done using a timeout of 60 s and a bound of 3. It can be seen in Table 2a and 2b that the differences of the previous tests are completely leveled, and instead our prover performs slightly worse than APLL.

Overall, since llprover uses incremental search, its times are often skewed towards the slower side. Similarly our prover is consistently slightly slower that APLL, this difference is negligible and can be probably ascribed to differences languages' run-times.

prover	**timeouts**	**failures**	**successes**	**success rate**	**avg. (succ.)**	**avg. (tot.)**
APLL	0	17	71	≈ 0.80	0.035 s	0.326 s
llprover	20	6	62	≈ 0.70	0.981 s	2.179 s
sat-ll	5	15	68	≈ 0.77	0.443 s	0.496 s

(a) Outputs for KLE-cbv

prover	**timeouts**	**failures**	**successes**	**success rate**	**avg. (succ.)**	**avg. (tot.)**
APLL	0	16	72	≈ 0.80	0.037 s	0.055 s
llprover	20	6	62	≈ 0.70	1.709 s	3.253 s
sat-ll	4	18	66	≈ 0.75	0.130 s	0.185 s

(b) Outputs for KLE-cbn

Table 2: Exponential tests.

Chapter 6

Conclusion

placeholder.

Appendix A

Example derivation

The proof to the judgment

$$(a \otimes b)^{\perp} \vdash a^{\perp} \parr b^{\perp}$$

which is normalized to

$$\vdash (a \otimes b), a^{\perp} \parr b^{\perp}$$

corresponds in our calculus to

$$
\begin{array}{r c}
 & \nabla' \qquad\qquad\qquad \nabla'' \\ \cline{2-2}
[\otimes] & \vdash . : \mathrm{af}(a^{\perp}, x_2), \mathrm{af}(b^{\perp}, x_2) \Downarrow \mathrm{af}(a \otimes b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[D_1] & \vdash . : \mathrm{af}(a^{\perp}, x_2), \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a \otimes b, x_1) \Uparrow . \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[R\Uparrow] & \vdash . : \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a \otimes b, x_1) \Uparrow \mathrm{af}(a^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[R\Uparrow] & \vdash . : \mathrm{af}(a \otimes b, x_1) \Uparrow \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[\parr] & \vdash . : \mathrm{af}(a \otimes b, x_1) \Uparrow \mathrm{af}(a^{\perp} \parr b^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[R\Uparrow] & \vdash . : . \Uparrow \mathrm{af}(a \otimes b, x_1), \mathrm{af}(a^{\perp} \parr b^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V}
\end{array}
$$

with

$$\nabla' = \begin{array}{rc}
 & x_1 \text{ used}, x_2 \text{ used}, x_2x_3 \text{ used}, x_2x_4 \text{ avail}\downarrow \mathrm{V} \\ \cline{2-2}
[I_1] & \vdash . : \mathrm{af}(a, x_1), \cancel{\mathrm{af}(b^\perp, x_2x_4)} \Downarrow \mathrm{af}(a^\perp, x_2x_3) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[D_1] & \vdash . : \mathrm{af}(a, x_1), \mathrm{af}(a^\perp, x_2x_3), \cancel{\mathrm{af}(b^\perp, x_2x_4)} \Uparrow . \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[R\Uparrow] & \vdash . : \mathrm{af}(a^\perp, x_2x_3), \cancel{\mathrm{af}(b^\perp, x_2x_4)} \Uparrow \mathrm{af}(a, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V} \\ \cline{2-2}
[R\Downarrow] & \vdash . : \mathrm{af}(a^\perp, x_2x_3), \cancel{\mathrm{af}(b^\perp, x_2x_4)} \Downarrow \mathrm{af}(a, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : \mathrm{V}
\end{array}$$

$$\nabla'' = \begin{array}{rc}
 & x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail}, x_2\overline{x_4} \text{ used}, x_2\overline{x_3} \text{ avail}\downarrow \mathrm{V} \\ \cline{2-2}
[I_1] & \vdash . : \mathrm{af}(b, x_1), \cancel{\mathrm{af}(a^\perp, x_2\overline{x_3})} \Downarrow \mathrm{af}(b^\perp, x_2\overline{x_4}) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : \mathrm{V} \\ \cline{2-2}
[D_1] & \vdash . : \mathrm{af}(b, x_1), \cancel{\mathrm{af}(a^\perp, x_2\overline{x_3})}, \mathrm{af}(b^\perp, x_2\overline{x_4}) \Uparrow . \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : \mathrm{V} \\ \cline{2-2}
[R\Uparrow] & \vdash . : \cancel{\mathrm{af}(a^\perp, x_2\overline{x_3})}, \mathrm{af}(b^\perp, x_2\overline{x_4}) \Uparrow \mathrm{af}(b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : \mathrm{V} \\ \cline{2-2}
[R\Downarrow] & \vdash . : \cancel{\mathrm{af}(a^\perp, x_2\overline{x_3})}, \mathrm{af}(b^\perp, x_2\overline{x_4}) \Downarrow \mathrm{af}(b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : \mathrm{V}
\end{array}$$

$$\mathrm{V} = \{x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot\}$$

For reference we show the classic proof (non focused, without constraints) for the same judgment:

$$[\parr]\,\dfrac{[\otimes]\,\dfrac{[A]\,\dfrac{}{\vdash a, a^\perp} \qquad [A]\,\dfrac{}{\vdash b, b^\perp}}{\vdash a^\perp, b^\perp, (a \otimes b)}}{\vdash (a \otimes b), (a^\perp \parr b^\perp)}$$

Bibliography

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[2] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, Dec 1935.

[3] Gerhard Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische Zeitschrift*, 39(1):405–431, Dec 1935.

[4] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[5] James Harland and David J. Pym. Resource-distribution via boolean constraints. *ACM Trans. Comput. Log.*, 4(1):56–90, 2003.

[6] Chuck C. Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.

[7] Paul Tarau and Valeria de Paiva. Deriving theorems in implicational linear logic, declaratively. In Francesco Ricca, Alessandra Russo, Sergio Greco, Nicola Leone, Alexander Artikis, Gerhard Friedrich, Paul Fodor, Angelika Kimmig, Francesca A. Lisi, Marco Maratea, Alessandra Mileo, and Fabrizio Riguzzi, editors, *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, volume 325 of *EPTCS*, pages 110–123, 2020.

[8] Markus Triska. Boolean constraints in SWI-Prolog: A comprehensive system description. *Science of Computer Programming*, 164:98 – 115, 2018. Special issue of selected papers from FLOPS 2016.