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

1.1 Why Prolog

Prolog as a language and as an environment has been historically tied to automated theorem proving for its ability to express its algorithms naturally. One particularly convenient characteristic of Prolog is its automatic management of backtracking, in most other languages we would have had to use exceptions to walk down the stack, or a queue of unfinished computations, which would have made the code much less readable.

Most Prolog implementations also support CLP or constraint logic programming. This is implemented through a series of libraries, these libraries allow to express constraint in the body of the clauses referencing some attribute of the variables; in our case we will use CLP($\mathcal{B}$)[5], which provides tools to deal with boolean constraints. By boolean constraint in this context we mean an expression made of Prolog variables and the constants 1 and 0, respectively true and false. The allowed operators in these expressions 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. A constraint may be checked using the predicate `sat/1`, which succeeds iff it is satisfiable. Since we only deal with conjunctions we define the helper predicate `check/1`

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

The predicate `sat/1` may work a bit unexpectedly, to clarify its possible outcomes we give a little example:

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

Here we can see three different cases:

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

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 variables be wrapped by the functor `v/1`. This small explanation sums up the extent of the library we use in our prover.

Chapter 2

The focused calculus

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. Intuitively, a sequent is in NNF if all its formulae are in NNF.

A generic formula is then normalized applying recursively the DeMorgan rules for linear logic, until NNF is reached. The process of normalization 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 $\Delta^{\perp}$.

This choice has some implementation-wise advantages, but for now we will only care about the fact that it shrinks the size of the complete calculus by roughly half since we only have to deal with the right rules of the connectives.

$$
\begin{array}{rcccccccl}
\phi & ::= & 1 & | & \phi \otimes \phi & | & \bot & | & \phi \parr \phi \quad \text{(Multiplicatives and their constants)} \\
 & | & 0 & | & \phi \oplus \phi & | & \top & | & \phi \,\&\, \phi \quad \text{(Additives and their constants)} \\
 & | & !\phi & | & ?\phi & & & & \text{(Exponentials)} \\
 & | & \alpha^{\perp} & | & \alpha & & & & \text{(Where } \alpha \text{ is a term)}
\end{array}
$$

Figure 1: Normalized linear logic formulae

As seen in Figure 1 we will use ϕ for formulae and α for terms.

2.2 Focusing

Focusing is a technique described by Andreoli in his seminal paper [1]. There he recognizes two alternating phases in a proof: a deterministic phase, where the order of rule application to the sequent does not matter; and a non deterministic phase, where several choices may be tried. These two phases are respectively called asynchronous and synchronous phase.

Definition 2.2.1. Given a formula ϕ we define the following predicates: "ϕ asy" indicates that the rule for the top-level connective of the formula ϕ is asynchronous on the right, these connectives are

$$\parr, \&, ?, \top, \bot$$

conversely we define "ϕ sync" to indicate that the rule for the top-level connective of ϕ is synchronous on the right, these are

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

Furthermore in focusing everything is assigned a negative or positive polarity. Synchronous connectives are defined to have a positive polarity, whereas asynchronous connectives are define to have a negative polarity. Terms also have a polarity, which may be assigned with some arbitrarily complex mechanisms. We will follow [3] and simply assign atoms with a negative polarity and negated atoms with a positive one.

Definition 2.2.2. "α atom" is a predicate that is true only when α is a negative literal (i.e. an atom). Conversely "α pos lit" is a predicate that is only when α is a positive literal (i.e. a negated atom).

2.3 Constraints

Our calculus uses constraints to manage the resources.

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.

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

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

These mean respectively that:

- the variable name x has not yet occurred in any expression in the proof tree, i.e. does not appear in any constraint of the father, or of its (the father) siblings and their sub-trees.
- each variable name $x_i, x_j \in X$ has not yet occurred in the proof and each variable in X is distinct.

$$\forall i, j \mid i \neq j \Rightarrow x_i \neq x_j$$

As seen in Figure 2 we will call e such a conjunction and x the single boolean variables.

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

Figure 2: Definition of a boolean variable and expression

Definition 2.3.3 (Annotated formula)**.** Given a formula ϕ defined as in Figure 1 and a boolean expression e defined as in Definition 2.3.1, an *annotated formula* is simply a term

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

that associates the formula to the expression. We denote

- $\exp(\text{af}(\phi, e)) = e$ as the operation of extracting the boolean expression associated to a given formula; and then extend this notation to sequents such that $\exp(\Delta)$ is the set of all boolean expressions of Δ.

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

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 depend on the variables 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.

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

Definition 2.3.4 (Constraints)**.** Given an annotated formula $\text{af}(\phi, e)$ as in Definition 2.3.3, a constraint λ may be of two kinds

- "e used" states that the formula ϕ gets consumed in this branch of the proof. This corresponds to saying the expression e is true or

$$x_i \wedge \cdots \wedge x_j = \top$$

- "e avail" 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

$$x_i \wedge \cdots \wedge x_j = \bot$$

We then extend these predicates to sequents

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

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 (Evaluation)**.** Given a boolean expression e and a function V mapping variables to their values

$$\mathrm{V} : \{x_1, x_2, \ldots, x_n\} \to \{\top, \bot\}$$

with $\mathrm{vars}(e) \subseteq \mathrm{Dom}(\mathrm{V})$; 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 with the assignment V. We extend this notation, such that

- given a constraint λ and an assignment V, such that $\mathrm{vars}(\lambda) \subseteq \mathrm{Dom}(\mathrm{V})$

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

- given an annotated sequent Δ and an assignment V, such that $\mathrm{vars}(\Delta) \subseteq \mathrm{Dom}(\mathrm{V})$

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

A branch is then considered as correct if its constraints are satisfiable. Otherwise the branch of the proof fails.

Definition 2.3.6 (Satisfaiability of constraints)**.** Given a set of constraints Λ and an assignment function V with $\mathrm{vars}(\Lambda) \subseteq \mathrm{Dom}(\mathrm{V})$; we write

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

Fact 2.3.1. *We can translate an assignment function to a set of constraints using*

simple reversible transformations:

$$\begin{aligned} \mathrm{V} &= \{\dots, (x_i, \top), (x_j, \bot), \dots\} \\ &= \{\dots, x_i = \top, x_j = \bot, \dots\} \\ &= \{\dots, x_i \text{ used}, x_j \text{ avail}, \dots\} \end{aligned}$$

We now expand the concept of triadic sequent of [1] by adding constraints

Definition 2.3.7 (Members of the sequent)**.** Given any sequent this can be in either two forms:

- focused or in the synchronous phase, written:

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

- in the asynchronous phase, written:

$$\vdash \Psi : \Delta \Uparrow \Phi \parallel \Lambda : \mathrm{V}$$

Where

- Ψ is a set of unrestricted non annotated formulae, or all formulae that can be freely discarded or duplicated.
- Δ and Φ are multisets 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 solution as defined in Definition 2.3.6. 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 [2], which prioritizes generality. The choice of letters is mainly a mnemonic or visual one, constraints Λ "go-up" the proof tree and solutions V "come down" from the leaves.

Definition 2.3.8 (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

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

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 respectively as the left projection and the right projection of the pair (Δ_L, Δ_R).

As a small example for clarity, given the sequent

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

this is split into

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

Figure 3 shows the calculus [2] using our notation of explicit contraints. What is actually show is roughly what they define as the "lazy" strategy. Figure 4 instead presents our focused constraint calculus.

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

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

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

$$[\otimes]\ \frac{\vdash_{\mathrm{PH}} \Delta_L^X, \mathrm{af}(\phi_1, e) \parallel \Lambda, e \text{ used} : \mathrm{V}' \qquad \vdash_{\mathrm{PH}} \Delta_R^X, \mathrm{af}(\phi_2, e) \parallel \mathrm{V}' : \mathrm{V}''}{\vdash_{\mathrm{PH}} \Delta, \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

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

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

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

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

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

Figure 3: The one sided version of the calculus from [2] with explicit constraint propagation

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

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

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

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

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

(a) Asynchronous rules

$$[\otimes]\ \frac{X \text{ new} \quad \vdash \Psi : \Delta_L^X \Downarrow \mathrm{af}(\phi_1, e) \parallel \Lambda, e \text{ used} : \mathrm{V}' \quad \vdash \Psi : \Delta_R^X \Downarrow \mathrm{af}(\phi_2, e) \parallel \mathrm{V}' : \mathrm{V}''}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

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

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

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

(b) Synchronous rules

$$[I_1]\ \frac{\alpha \text{ atom} \qquad \Lambda, e_1 \text{ used}, e_2 \text{ used}, \Delta \text{ avail}{\downarrow}\mathrm{V}}{\vdash \Psi : \mathrm{af}(\alpha, e_1), \Delta \Downarrow \mathrm{af}(\alpha^{\perp}, e_2) \parallel \Lambda : \mathrm{V}} \qquad [D_1]\ \frac{\neg\phi \text{ atom} \qquad \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel \Lambda : \mathrm{V}}{\vdash \Psi : \mathrm{af}(\phi, e), \Delta \Uparrow . \parallel \Lambda : \mathrm{V}}$$

$$[I_2]\ \frac{\alpha \text{ atom} \qquad \Lambda, e \text{ used}, \Delta \text{ avail}{\downarrow}\mathrm{V}}{\vdash \alpha, \Psi : \Delta \Downarrow \mathrm{af}(\alpha^{\perp}, e) \parallel \Lambda : \mathrm{V}} \qquad [D_2]\ \frac{\neg\phi \text{ atom} \qquad x \text{ new} \qquad \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, x) \parallel \Lambda, e \text{ used} : \mathrm{V}}{\vdash \phi, \Psi : \Delta \Uparrow . \parallel \Lambda : \mathrm{V}}$$

(c) Identity and decide rules

Figure 4: Focused constraint calculus for Linear Logic

Lemma 2.3.1. *If there exists* V *such that*

$$\vdash_{PH} \Delta \parallel \Delta \text{ used} : \mathrm{V}$$

then there is V′ *such that*

$$\vdash . : . \Uparrow \Delta \parallel \Delta \text{ used} : \mathrm{V}'$$

Lemma 2.3.2. *Forall assignments* V *and sequents* Δ*,*

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

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 be 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 $\phi \notin \Delta_L^X[\mathrm{V}]$. ■

Lemma 2.3.3. *Forall assignments* V *and sequents* Δ*,*

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

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 2.3.8). 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. ■

Lemma 2.3.4. *Given any assignment* V *and any sequent* Δ*, splitting induces a partition on the sequent* $\Delta[\mathrm{V}]$*.*

Proof. See Lemma 2.3.2 and 2.3.3. ■

Lemma 2.3.5. *Forall assignments* V *and set of constraints* Λ*, if (using the translation of Fact 2.3.1)*

$$\mathrm{V}, \Lambda \downarrow \mathrm{V}'$$

then $\mathrm{Dom}(\mathrm{V}) \subseteq \mathrm{Dom}(\mathrm{V}')$ *and*

$$\forall x_i \in \mathrm{Dom}(\mathrm{V}) \mid \mathrm{V}(x_i) = \mathrm{V}'(x_i)$$

In this case we say that V' *subsumes* V*.*

Proof. The proof is straightforward:

- the first condition trivially holds, since an assignment must be defined for each variable in the constraints and V' is an assignment defined for a superset of the constraints derived from V;
- for the second condition we have that

  $$\mathrm{V}, \Lambda = \{\ldots, x_i \text{ used}, x_j \text{ avail}, \ldots\} \cup \Lambda$$

  and to have that $\mathrm{V}, \Lambda{\downarrow}\mathrm{V}'$

  $$\bigwedge_{\lambda \in \mathrm{V}, \Lambda} \lambda[\mathrm{V}'] = \top$$

  thus the constraints derived from V must still hold with assignment V'.

■

Lemma 2.3.6. *Forall sets of constraints* Λ *and forall assignments* V*, if*

$$\Lambda{\downarrow}\mathrm{V}$$

then for any $\Lambda' \subseteq \Lambda$

$$\Lambda'{\downarrow}\mathrm{V}$$

Proof. Following the Definition 2.3.6 of satisfiability, every constraint in Λ must hold under assignment V. This trivially means that all the constraints in Λ' must also hold. ■

Theorem 2.3.1 (Soundness)**.**

Proof. As a soundness proof we give a translation to the triadic system defined by Andreoli in [1]:

$$[\bot]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \Phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \bot, \Phi} \qquad [\parr]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1, \phi_2, \Phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1 \parr \phi_2, \Phi} \qquad [?]\ \frac{\vdash_{\mathrm{A}} \phi, \Psi : \Delta \Uparrow \Phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow ?\phi, \Phi}$$

$$[\top]\ \frac{}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \top, \Phi} \qquad [\,\&\,]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1, \Phi \qquad \vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_2, \Phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi_1 \,\&\, \phi_2, \Phi}$$

$$[\oplus_L]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_1}{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi_1 \oplus \phi_2} \qquad [1]\ \frac{}{\vdash_{\mathrm{A}} \Psi : . \Downarrow 1} \qquad [!]\ \frac{\vdash_{\mathrm{A}} \Psi : . \Uparrow \phi}{\vdash_{\mathrm{A}} \Psi : . \Downarrow !\phi}$$

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

$$[R\Uparrow]\ \frac{\neg\phi \text{ asy} \qquad \vdash_{\mathrm{A}} \Psi : \phi, \Delta \Uparrow \Phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi, \Phi} \qquad [I_1]\ \frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \Psi : \alpha \Downarrow \alpha^{\perp}} \qquad [D_1]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi}{\vdash_{\mathrm{A}} \Psi : \phi, \Delta \Uparrow .}$$

$$[R\Downarrow]\ \frac{\phi \text{ negative} \qquad \vdash_{\mathrm{A}} \Psi : \Delta \Uparrow \phi}{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi} \qquad [I_2]\ \frac{\alpha \text{ atom}}{\vdash_{\mathrm{A}} \alpha, \Psi : . \Downarrow \alpha^{\perp}} \qquad [D_2]\ \frac{\vdash_{\mathrm{A}} \Psi : \Delta \Downarrow \phi}{\vdash_{\mathrm{A}} \phi, \Psi : \Delta \Uparrow .}$$

Figure 5: J.-M. Andreoli's triadic calculus.

The three base cases are proved essentially in the same way:

I_1: Given the judgment

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

looking at the constraints we trivially get that

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

so we can rewrite this as

$$[I_1]\ \frac{\alpha \text{ atom}}{\vdash \Psi : \alpha \Downarrow \alpha^{\perp}}$$

I_2: Given the judgment

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

proceeding as above we get

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

thus

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

1: Given the judgment

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

proceeding as above we get

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

thus

$$[1]\,\frac{}{\vdash \Psi : . \Downarrow 1}$$

The induction step is rather mechanic, with the exception of

- the tensor rule ($\otimes$):

$$[\otimes]\,\frac{X \text{ new} \quad \vdash \Psi : \Delta_L^X \Downarrow \mathrm{af}(\phi_1, e) \parallel \Lambda, e \text{ used} : \mathrm{V}' \quad \vdash \Psi : \Delta_R^X \Downarrow \mathrm{af}(\phi_2, e) \parallel \mathrm{V}' : \mathrm{V}''}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi_1 \otimes \phi_2, e) \parallel \Lambda : \mathrm{V}''}$$

  By the Lemma 2.3.5 we can say that $\Delta_L^X[\mathrm{V}'] = \Delta_L^X[\mathrm{V}'']$, since V'' subsumes V'. This being said, using Lemma 2.3.2 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. Finally Lemma 2.3.3 guarantees that the union of the two contexts returns the whole original context. Given this we get that this equals to

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

- the with rule (&):

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

which after erasing the unused formulae using the assignment becomes

$$[\,\&\,]\,\frac{\vdash \Psi : \Delta[\mathrm{V}'] \Uparrow \phi_2, \Phi[\mathrm{V}'] \qquad \vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_1, \Phi[\mathrm{V}'']}{\vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_1 \,\&\, \phi_2, \Phi[\mathrm{V}'']}$$

or equivalently

$$[\,\&\,]\,\frac{\vdash \Psi : \Delta[\mathrm{V}'] \Uparrow \phi_2, \Phi[\mathrm{V}'] \qquad \vdash \Psi : \Delta[\mathrm{V}''] \Uparrow \phi_1, \Phi[\mathrm{V}'']}{\vdash \Psi : \Delta[\mathrm{V}'] \Uparrow \phi_1 \,\&\, \phi_2, \Phi[\mathrm{V}']}$$

since $\Lambda \subseteq e$ used$, \Lambda$ and Lemma 2.3.6.

$(\,\parr\,)$: Assuming

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

maps to

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

under V. Then

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

maps to

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

Since the assignment V is the same, then $\Delta[\mathrm{V}]$ and $\Phi[\mathrm{V}]$ stay the same, and since the assignment respects the constraint e used, then $\phi_1 \parr \phi_2$ must be present.

$(R\Downarrow)$: Assuming

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

maps to

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

under V. Then

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

maps to

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

since the sequents do not change, only the phase shifts.

$$\bot \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow \Phi[\mathrm{V}]}{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow \bot, \Phi[\mathrm{V}]}$$

$$\parr \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow \phi_1, \phi_2, \Phi[\mathrm{V}]}{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow \phi_1 \parr \phi_2, \Phi[\mathrm{V}]}$$

$$\oplus_L \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1}{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1 \oplus \phi_2}$$

$$\oplus_R \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi_2}{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi_1 \oplus \phi_2}$$

$$! \quad \mapsto \quad \frac{\vdash \Psi : . \Uparrow \phi}{\vdash \Psi : . \Downarrow !\phi}$$

$$? \quad \mapsto \quad \frac{\vdash \phi, \Psi : \Delta[\mathrm{V}] \Uparrow \Phi[\mathrm{V}]}{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow ?\phi, \Phi[\mathrm{V}]}$$

$$R\Downarrow \quad \mapsto \quad \frac{\phi \text{ asy} \vee \phi \text{ atom} \qquad \vdash \Psi : \Delta[\mathrm{V}] \Uparrow \phi}{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi}$$

$$R\Uparrow \quad \mapsto \quad \frac{\neg\phi \text{ asy} \qquad \vdash \Psi : \phi, \Delta[\mathrm{V}] \Uparrow \Phi[\mathrm{V}]}{\vdash \Psi : \Delta[\mathrm{V}] \Uparrow \phi, \Phi[\mathrm{V}]}$$

$$D_1 \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi}{\vdash \Psi : \phi, \Delta[\mathrm{V}] \Uparrow .}$$

$$D_2 \quad \mapsto \quad \frac{\vdash \Psi : \Delta[\mathrm{V}] \Downarrow \phi}{\vdash \phi, \Psi : \Delta[\mathrm{V}] \Uparrow .}$$

■

Chapter 3

Implementation

We now describe the main implementation details of the prover. When explaining the code we will use some common names for variables, these are

- `A` is a set of unrestricted atoms, which purpose will be explained in Section 3.3.2;
- `U` is a set of unrestricted formulae;
- `F`, `F1`, ..., are formulae, and `Fs` and `D` are a lists of them;
- `S` is the queue of currently usable unrestricted formulae, which purpose will be explained in Section 3.3.3;
- `In` is a list of constraints.

3.1 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 process has just two steps

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

This is possible since classic linear logic is symmetric and negation is an involution.

The purpose of normalization is to cut away a great deal of possible rules applicable to the sequent, sacrificing some of the structure of the sequent. In fact the number of rules one needs to implement, and thus the number of different choices the prover may have in a certain moment, is more than halved as said in Section 2.1.

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

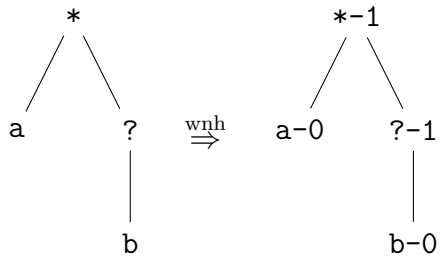
Definition 3.1.1 (Why-not height)**.** Why-not height is the maximum number of nested "why-not"s in a formula, or

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

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 3.3.1.
- In the case of plus ($\oplus$), the branch 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 3.3.3.

After this transformation formulae are attribute trees with at each node the why-not. 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: given a sequent Δ we obtain

$$\hat{\Delta} = \{\mathrm{af}(\phi, x) \mid x \text{ new}, \phi \in \Delta\}$$

To be clear, a variable is only assigned to the "top-level" formula, and sub-formulae are left unchanged.

The process of annotation is implemented by the predicate `annotate/3`. This also returns the initial constraints, with set each variable to "used". Thus stating that in the proof each and every formula must be used.

3.2 Helper predicates

We now define some helper predicates to work with the constraints. What we defined as Δ avail in Definition 2.3.4 corresponds to the predicate `set_to_zero/2`.

The other helper predicate implements the split function defined in Definition 2.3.8.

```
%! 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).
```

It is again a map over the list of formulae, that generates the new formulae and the constraints accordingly. Three new Prolog variables are introduced: `X`, `Y` and `Z`. `X` is the new variable, the annotated formulae refer to the variable `Y` and `Z`, and constraints are added so that

$$y = x \wedge e$$
$$y = \overline{x} \wedge e$$

Compare this to the original definition of Definition 2.3.8, one can see that the two are basically identical other than the fact that here the name of the variable (the atom)

and its value are treated separately. In the implementation the concept of variable is split in two: the name of the variable – represented by a Prolog atom, and the value of the variable – represented by a Prolog variable. This is needed since, after checking the constraints, the SAT-solver unifies the variable to its value if it finds a satisfiable solution; the purpose of the atom is to associate the variable value to its name if the final proof.

3.3 Focusing

3.3.1 Asynchronous and focusing phase

During the asynchronous phase we have a list of formulae which are being worked on and a list of formulae which are put to the side; with the former being called `Fs` and the latter `D`. At each step we analyze the first element of the list `Fs`, and we keep decomposing the members of the list until we cannot anymore. This process can be seen for example in the predicate for with (&)

```
async(A, U, D, [F|Fs], S, M, In) :-
  F = af(((F1-H1) & (F2-H2))-_), N, E), !,
  ( H2 > H1
  -> async(A, U, D, [af((F1-H1), N, E)|Fs], S, M, [v(E) =:= 1|In]),
     async(A, U, D, [af((F2-H2), N, E)|Fs], S, M, [v(E) =:= 1|In])
  ;  async(A, U, D, [af((F2-H2), N, E)|Fs], S, M, [v(E) =:= 1|In]),
     async(A, U, D, [af((F1-H1), N, E)|Fs], S, M, [v(E) =:= 1|In])
  ).
```

Here we can see both the choice being made based on the why-not height of the two sub-formulae (Definition 3.1.1), and how the with is broken down. Compare this with the & rule in Figure 4. The cut at line 2 is essential: it signifies that when an asynchronous connective is encountered the only thing to do is to break it apart.

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 can be seen in the rule `to_delta` which implements the rule $R\Uparrow$

```
async(A, U, D, [F|Fs], S, M, In, _) :-
  F = af((F1-_), _, _),
  not(is_asy(F1)), !,
  async(A, U, [F|D], Fs, S, M, In, _).
```

This process goes as long as `Fs` has still formulae inside.

When `Fs` is empty the phase switches, and the focusing process begins: we choose a formula – called `decide` – from either `D` or `U` and we break it down until either an

asynchronous connective or a negated atom is left. This is represented by the rules D_1 and D_2 that will be discussed further ahead in Section 3.3.3.

3.3.2 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 3.2, and the constraints are checked. This is encoded in the clause

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

A slightly different thing happens 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 approach based on the fact that once negative literals are put in a sequent they can never leave it, and is due to the fact that since `U` may be sorted many times, we try to keep the number of formulae in it small.

Some care is to be given to explaining how the constraints propagate. In fact, in contrast to Figure 4 the implementation does not have explicit propagation of the solution of the constraints. This is because Prolog's unification implicitly propagates a solution from one branch to another.

3.3.3 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, In) :-
  \+ U = [],
```

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

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.
  ```

  Line 4 is a sort in increasing order on the second attribute (the why-not height), keeping duplicates. 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, In) :-
  gensym(x, X),
  focus(A, U, D, af(H, X, E), T, M, [v(E) =:= 1|In]).
early_stop(A, U, D, [_|T], M, In) :-
  early_stop(A, U, D, T, M, In).
```

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.

3.4 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

```
async(A, U, D, [F|Fs], S, M, In, node(par, A, U, D, [F|Fs], [Tree])) :-
  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|In], Tree).
```

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 tree 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 appending the sequents 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 forward 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 method of input/output . 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 give 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 [3]. 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.

4.2 llprover

`llprover` is a prover 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. Using classical logic as the system uses all the rules, included the ones for implicative linear logic. For this reason the prover does not implement normalization to NNF before proof search; 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

```
1 rule([ill,0], no,  r(*), S, [S1, S2], [r(N),r(N1),r(N2)]) :-
2   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 speed.

Chapter 5

Testing

5.1 Infrastructure

The infrastructure for the project is divided in two parts:

- the nix "glue",
- 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.

So 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 `call_sat_prover` takes as input the premises and the conclusions and returns the call to the prover as a list. The innermost decorator (line 3) then accepts a dictionary, and returns a funciton 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 function 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 succeded 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 [4] 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`

Furthermore 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	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 text 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, since 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 $\bot, \top, \ldots$. 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.

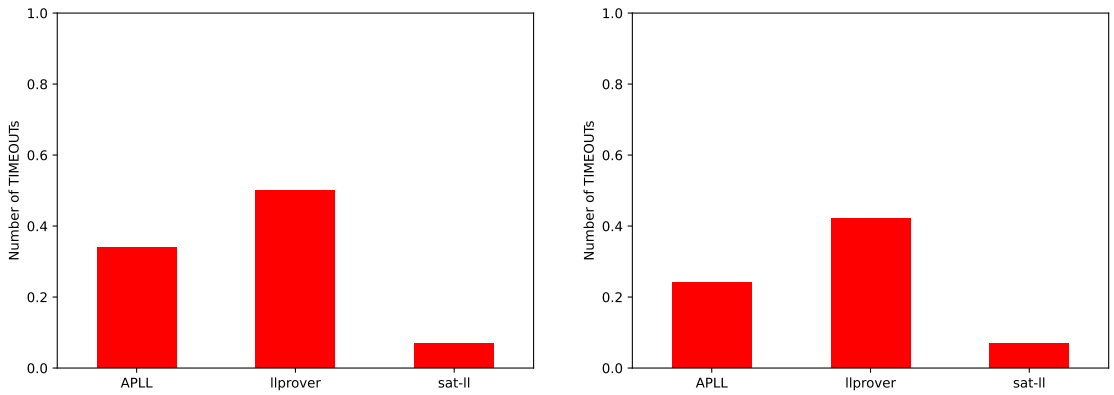
5.2 Benchmarking

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 used to quickly check for obvious bugs.

Randomly generated tests are only used for non-exponential 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 answer a certain formula is false just because it exhausted its number of contractions. This is a problem in the case of 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 clearly see that our prover outperforms APLL (and llprover) when dealing with formulae rich in multiplicatives (Figure 6a and Table 1a).



(a) Multiplicative case

(b) Multiplicative and additive case

Figure 6: 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 6b 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 6a

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 6b

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 6a's tests were composed of formulae of `size` 100, and `atoms` 50, using just tensor ($\otimes$) and par ($\parr$) as connectives;
- Figure 6b's tests were composed of formulae of `size` 500, and `atoms` 250, with all the connectives, except the exponentials ($\otimes$, $\parr$, & 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 – apart from the timeouts because of the added complexity – some failures not due to bugs, but because of the the bound. The benchmarks are done using a timeout of 60 s and a bound of 3. It can be seen in Table 2b and 2a that in this case 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 due to the fact that APLL is compiled, whereas our prover is interpreted.

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.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\nabla' \qquad \nabla''}{\vdash . : \mathrm{af}(a^{\perp}, x_2), \mathrm{af}(b^{\perp}, x_2) \Downarrow \mathrm{af}(a \otimes b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[\otimes]}{\vdash . : \mathrm{af}(a^{\perp}, x_2), \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a \otimes b, x_1) \Downarrow . \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[D_1]}{\vdash . : \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a \otimes b, x_1) \Downarrow \mathrm{af}(a^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Uparrow]}{\vdash . : \mathrm{af}(a \otimes b, x_1) \Downarrow \mathrm{af}(b^{\perp}, x_2), \mathrm{af}(a^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Uparrow]}{\vdash . : \mathrm{af}(a \otimes b, x_1) \Downarrow \mathrm{af}(a^{\perp} \parr b^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[\parr]}{\vdash . : . \Downarrow \mathrm{af}(a \otimes b, x_1), \mathrm{af}(a^{\perp} \parr b^{\perp}, x_2) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Uparrow]$$

with

$$\nabla' = \dfrac{\dfrac{\dfrac{\dfrac{x_1 \text{ used}, x_2 \text{ used}, x_2x_3 \text{ used}, x_2x_4 \text{ avail}\downarrow x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}{\vdash . : \mathrm{af}(a, x_1), \cancel{\mathrm{af}(b^{\perp}, x_2x_4)} \Downarrow \mathrm{af}(a^{\perp}, x_2x_3) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[I_1]}{\vdash . : \mathrm{af}(a, x_1), \mathrm{af}(a^{\perp}, x_2x_3), \cancel{\mathrm{af}(b^{\perp}, x_2x_4)} \Downarrow . \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[D_1]}{\vdash . : \mathrm{af}(a^{\perp}, x_2x_3), \cancel{\mathrm{af}(b^{\perp}, x_2x_4)} \Downarrow \mathrm{af}(a, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Uparrow]}{\vdash . : \mathrm{af}(a^{\perp}, x_2x_3), \cancel{\mathrm{af}(b^{\perp}, x_2x_4)} \Downarrow \mathrm{af}(a, x_1) \parallel x_1 \text{ used}, x_2 \text{ used} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Downarrow]$$

$$\nabla'' = \dfrac{\dfrac{\dfrac{\dfrac{x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail}, x_2\overline{x_4} \text{ used}, x_2\overline{x_3} \text{ avail}\downarrow x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}{\vdash . : \mathrm{af}(b, x_1), \cancel{\mathrm{af}(a^{\perp}, x_2\overline{x_3})} \Downarrow \mathrm{af}(b^{\perp}, x_2\overline{x_4}) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[I_1]}{\vdash . : \mathrm{af}(b, x_1), \cancel{\mathrm{af}(a^{\perp}, x_2\overline{x_3})}, \mathrm{af}(b^{\perp}, x_2\overline{x_4}) \Downarrow . \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[D_1]}{\vdash . : \cancel{\mathrm{af}(a^{\perp}, x_2\overline{x_3})}, \mathrm{af}(b^{\perp}, x_2\overline{x_4}) \Downarrow \mathrm{af}(b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Uparrow]}{\vdash . : \cancel{\mathrm{af}(a^{\perp}, x_2\overline{x_3})}, \mathrm{af}(b^{\perp}, x_2\overline{x_4}) \Downarrow \mathrm{af}(b, x_1) \parallel x_1 \text{ used}, x_2 \text{ used}, x_3 \text{ used}, x_4 \text{ avail} : x_1 \mapsto \top, x_2 \mapsto \top, x_3 \mapsto \top, x_4 \mapsto \bot}[R\Downarrow]$$

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

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

Bibliography

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

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

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

[4] 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.

[5] 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.