# Contents

# Chapter 1

# Intro

In 2001 Pym and Harland publish a paper [2] where they propose a new way to tackle the problem of splitting sequents during linear logic proof search using boolean constraints. The paper proposes a new calculus for linear logic that associates to each formula a boolean variable, and enforces linearity by constraints on said variables. 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 algorithmns.

We examine the efficiency of this method and we compare it to other provers for different substets of linear logic.

## 1.1 Linear logic

Linear logic is a logic proposed by Jean-Yves Girard in his seminal paper of 1987 [1]. The distintive trait of this logic is that its formulae cannot be copied (called weakening) or discarded (called contraction), but instead they are consumed. And a certain sequent it true if and only if all its formulae get consumed exactly once. For this reason this logic is sometimes called a logic of resource, 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 is doubled. To better see this let's analyze classic conjuction, this can be defined as

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

On the other hand, these two rules are not equivalent in linear logic, since the former implies some weakening and contraction. This is exactly the reason why in linear logic all connectives have two versions: an additive one – where the two branches keep the same context, and a multiplicative one – where the context gets partioned between the two branches. Obviously the constants $\top$ and $\bot$ also have two versions. We have that

| | Additive | Multiplicative |
|---|---|---|
| $\wedge$ | $\&$ | $\otimes$ |
| $\vee$ | $\oplus$ | $\parr$ |
| $\top$ | $\top$ | $1$ |
| $\bot$ | $0$ | $\bot$ |

It is the multiplicative side which bring the most complexity. The action of partitioning the context – called splitting – implies an exponential number of attempts to find which subset of the multiset is right for a certain branch.

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 bang 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.

## 1.2 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 to the other branch.

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

### 1.2.1 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.

Before diving into the analysis of this prover, we give the definition of *why-not height*

**Definition 1** (Why-not height)**.** Why-not height is the maximum number of nested "why-not"s in a formlua, or

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

This measure is used in their particular way of dealing with uncontrained formulae, but is also used as a way to decide which branch to prove first for any normal operator.

The program is written in OCaML and implements a pretty standard focused proof search on normalized formulae as seen in [3]. In this section we will illustrate two noteworthy charateristics of its implementation:

- Sequent splitting when encountering a tensor is done by generating all the numbers up to $2^{|\Delta|}$ – where $\Delta$ 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 4.5 this implementation choice will result in a degradation of performance on formulae with a high number of multiplicatives.

- This prover does not use a simple limit to the number of applications of the contraction rule in a branch, instead an initially empty queue of unrestricted formulae (`select_d2`) and a counter (`max_d2`) are kept during the search. Two cases arise:
  - if `select_d2 = []` and `max_d2 > 0` then the sequent of unrestricted formulae is taken, negative terms are filtered out and it is sorted based on why-not height.

    ```
    1 (* ... *)
    2 if select_d2 = [] then begin
    3   (if max_d2 = 0 then (bl := true; raise NoValue));
    4   let select_d2' =
    5     sort_whynot (List.filter (fun x -> not (is_neg x))
    6     (Set_formula.elements theta)) in
    7   if select_d2' = [] then None
    8   else
    9     apply_d2 select_d2' (max_d2 - 1) end
    10    (* ... *)
    ```

    This new list of unrestricted formule becomes the new `select_d2`. Otherwise if `select_d2` is still empty after being refilled (line 7) or if `max_d2` is 0 (line 3) the branch fails.
  - if `select_d2` 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.

  The main purpose of this whole process is to avoid infinite loops that always contract on the same formula. Instead all the formulae are tried one by one, starting from the simplest (lower why-not height).

The counter `max_d2` is a local bound, since decreasing it in a branch does not affect other branches.

### 1.2.2 `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.

Another particular characteristic of llprover is that it uses a local bound with iterative deepening, so in the benchmarks for formulae which need a lot of contractions, it will perform slightly worse.

## 1.3 Why Prolog

Prolog as a language and as an environment has been historically tied to automated theorem proving for its ability to express these kind of algorithmns naturally. One particularly conventient characteristic of Prolog is its automatic managing 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 allows to have constraints referencing some attributes of variables in the body of clauses, in our case we use CLP($\mathcal{B}$) for boolean constraints and an handy interface to a sat-solver. The library exposes operators to compose boolean formulas made of prolog variables

```
X = (X =:= 1),
Y = (X * X)
```

and to check the satisfaiability of said formulas

```
?- sat(X * Y).
X = Y, Y = 1.
```

One other characteristic of Prolog which revealed to be very handy for our prover is unification. Using this we didn't have to explicitly propagate the solutions of the SAT-solver, which instead were automatically propagated between branches.

# Chapter 2

# The focused calculus

Before describing the calculus we must give some definitions

**Definition 2** (Annotated formula)**.** Given a formula $\phi$ defined as in Figure 2.1 and a boolean expression $e$ defined as in Figure 2.2, an *annotated formula* is simply a term

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

that associates the formula to the expression. We denote

$$\exp(\mathrm{af}(\phi, e)) = e$$

and then extend this notation to sequents, such that $\exp(\Delta)$ is the set of all boolean expressions of $\Delta$.

As seen in Figure 2.1 and 2.2 we will usually use $\phi$ to refer to formulas, $x$ to refer to boolean variables, and $e$ to refer to boolean expressions (conjuctions of variables).

The purpose of putting formulae and variables 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 assignement of the variables and find out if the associated formula is used or not in a certain branch of a proof.

$$\begin{array}{rcccccccccl}
\phi & ::= & 1 & | & \phi \otimes \phi & | & \bot & | & \phi \parr \phi & & \text{(Multiplicatives and their constants)} \\
& | & 0 & | & \phi \oplus \phi & | & \top & | & \phi \,\&\, \phi & & \text{(Additives and their constants)} \\
& | & !\phi & | & ?\phi & & & & & & \text{(Exponentials)} \\
& | & \phi^{\perp} & | & \text{name} & & & & & &
\end{array}$$

Figure 2.1: Linear logic connectives

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

Figure 2.2: Definition of a boolean variable and expression

**Definition 3** (Consuming formulae)**.** Given a boolean expression $e$ as in Figure 2.2, we use the following notation

- "$e$ used" to state that the formula associated to the boolean expression $e$ gets consumed in this branch of the proof, this corresponds to saying the constraint $e$ is true;
- "$e$ $\neg$used" to state that the formula associated to the boolean variable $e$ does not get consumed in this branch of the proof, this corresponds to saying the constraint $e$ is not true.

We then extends these predicates to sequents

$$\Delta \text{ used} = \{e_2 \text{ used} \mid e_2 \in \exp(\Delta)\}$$
$$\Delta \neg\text{used} = \{e_2 \neg\text{used} \mid e_2 \in \exp(\Delta)\}$$

**Definition 4** (Members of the sequent)**.** Given any sequent this can be in either two forms:

- focused or in the synchronous phase, written:

$$\vdash \Psi : \Delta \Downarrow \phi \parallel \Lambda : V$$

- in the asynchronous phase, written:

$$\vdash \Psi : \Delta \Uparrow \Phi \parallel \Lambda : V$$

These two have more or less the same members, which are

- the set $\Psi$ of unrestricted formulae, or all formulae that can be freely discarded or duplicated;
- the multisets $\Delta$ and $\Phi$ 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;
- the set $\Lambda$ of constraints, which is to be interpreted as the conjunction of its members, and the set $V$ which represents a propagated solution. Furthermore if $\Lambda$ is satisfiable we write $\Lambda{\downarrow}V$, so:

$$\Lambda{\downarrow}V \iff \bigwedge_{e \in \Lambda} e \text{ is } \textit{sat} \text{ by } V$$

  $V$ is in and of itself a boolean expression as in Figure 2.2, in fact it can be seen as the conjunction of the variables assigned to true and the negation of the variables assigned to false. As such it can be used as a constraint, stating that a certain solution must be respected in a new one.

  This approach to constraints helps to make the flow of the variables and solutions through the proof tree more explicit and clear and leaves no ambiguity to where the constraints should be checked.

  The choice of letters is mainly a mnemonic or visual one, constraints $\Lambda$ "go-up" the proof tree and solutions $V$ "come down" from the leaves.

**Definition 5** (Splitting a sequent)**.** Given a sequent of annotated formulae $\Delta$ we define the operation of splitting it as a function

$$\text{split}(\Delta) \mapsto (\Delta^L, \Delta^R)$$

where, given a set $X$ of new variable names $x_i$ for each formula $\phi_i \in \Delta$

$$\Delta^L = \{\text{af}(\phi_i, x_i \wedge e_i) \mid i \in \{1, \ldots, n\}\}$$
$$\Delta^R = \{\text{af}(\phi_i, \overline{x_i} \wedge e_i) \mid i \in \{1, \ldots, n\}\}$$

with $n$ the cardinality of $\Delta$, and $\phi_i$ and $e_i$ respectively the formula and the variable of the $i$-eth annotated formula in $\Delta$ using an arbitrary order.

It is worth noting that the variables $x_i \in X$ used for $\Delta^L$ and $\Delta^R$ must be the same, this condition is necessary for the mechanism ensuring that a formula used on the left side of a tensor proof is not used on the right side and viceversa.

With a slight abuse of notation we will write $\text{split}(\Delta)_L$ and $\text{split}(\Delta)_R$ respectively as the left projection and the right projection of the pair $(\Delta_L, \Delta_R)$.

As a small example for clarity, given the sequent

$$\Delta = \text{af}(a \otimes b, x_1), \text{af}(c^\perp, x_2)$$

this is split into

$$\text{split}(\Delta)_L \mapsto \text{af}(a \otimes b, x_3 \wedge x_1), \text{af}(c^\perp, x_4 \wedge x_2)$$
$$\text{split}(\Delta)_R \mapsto \text{af}(a \otimes b, \overline{x_3} \wedge x_1), \text{af}(c^\perp, \overline{x_4} \wedge x_2)$$

**Definition 6.** Lastly given a formula $\phi$ we define the following predicates

- $\phi$ asy is a predicate that's true only when $\phi$ is an asynchronous formula, which are

$$\phi ::= \phi \parr \phi \mid \phi \,\&\, \phi \mid ?\phi \mid \top \mid \perp$$

- $\phi$ neg lit is a predicate that's true only when $\phi$ is a negative literal, in our implementation negative literals are atoms, and positive literals are negated atoms.

We are now ready to present the full calculus.

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

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

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

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

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

(a) Asynchronous rules

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

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

$$[1]\ \frac{e_1 \text{ used}, \Delta\ \neg\text{used}, \Lambda{\downarrow}V}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(1, e_1) \parallel \Lambda : V} \qquad [\,!\,]\ \frac{\vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e_1) \parallel e_1 \text{ used}, \Delta\ \neg\text{used}, \Lambda : V}{\vdash \Psi : \Delta \Downarrow \mathrm{af}(!\phi, e_1) \parallel \Lambda : V}$$

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

(b) Synchronous rules

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

$$[I_2]\ \frac{e_1 \text{ used}, \Delta\ \neg\text{used}, \Lambda{\downarrow}V}{\vdash \Psi, \phi : \Delta \Downarrow \mathrm{af}(\phi^{\perp}, e_1) \parallel \Lambda : V}$$

(c) Identity rules

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

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

(d) Decide rules

Figure 2.3: The complete focused constraint calculus

# Chapter 3

# Implementation

## 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 subformulae.

As a first transformation the sequent gets normalized into a sequent in negated normal form (NNF). NNF is the form where all negations are pushed down to atoms and all linear implications ($\multimap$) are expanded into pars ( $\parr$ ) using the following tautology

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

Normalization is a common technique – used in all the provers we compare with. The process is composed of 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 involutive.

The purpose of this process 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 we need to implement after normalization is more than halved, since we now need just the right rules, without the ones for negation and linear implication.

As a second transformation, to each formula we assign its why-not height, as defined in Definition 1. Why-not height is used during proof search to decide which branch to do first and which exponential to decide first. This is a technique borrowed from APLL and its obvious purpose is to try first the branches with less exponentials: in the case the first probably simpler branch fails, we do not have to try the other. After this transformation formulae are attribute trees, with at each node the why not height of the subformula.

As a third and final transformation, each formula gets annotated. This means we associate a variable to each formula in the sequent as in Definition 2. Given a sequent $\Delta$ 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 subformulae are left unchanged.

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 satisfaiable solution, so the purpose of the atom is to associate the variable value to its name if the final proof. The process of annotation is implemented by the predicate `annotate/3`

```
1 %! annotate(+[Formulae], -[AFs], -[Constraints]) is det.
2 annotate(Fs, Afs, Cns) :-
3   maplist([F, af(F, X, Var), v(Var) =:= 1]>>(gensym(x, X)), Fs, Afs, Cns).
```

which is a simple map over the sequent. The constraints returned state that each formula must have its variable to one, that is to say each formulae must be used. These are the constraints the proof search starts with.

## 3.2 Constraint propagation

Some care is to be given to explaining how the constraints propagate. In fact, in constrast to Figure 2.3 the implementation does not have explicit propagation of the solution to the constraints. This is because Prolog's unification implicitly propagates a solution from one branch to another.

## 3.3 Helper predicates

We now define some helper predicates to work with the constraints. What we defined as $\Delta$ $\neg$used in Definition 3 corresponds to the predicate `set_to_zero/2`

```
%! set_to_zero(+[AFs], -[Constr]) is det.
set_to_zero(Fs, Cns) :-
    maplist([af(_, _, E), v(E) =:= 0]>>true, Fs, Cns).
```

The other helper predicate implements the split function defined in Definition 5.

```
%! 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 5, 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.

## 3.4 Focusing

When explaining the code we will use some common names for variables, these are

- `A` is a set of unrestricted atoms;
- `U` is a set of unrestriched formulae;
- `F`, `F1`, ..., are formulae, and `Fs` and `D` are a lists of them;
- `S` is the queue of currently usable unrestricted formulae;
- `In` is a list of constraints.

### 3.4.1 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 `Fss` and the latter `D`. At each step we analyze the first element of the list `Fs`, and we keep scomposing the memebers of the list untill we can't anymore. This process can be seen for example in the predicate for &

```
1 async(A, U, D, [F|Fs], S, M, In, _) :-
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|In], _),
5      async(A, U, D, [af((F2–H2), N, E)|Fs], S, M, [v(E) =:= 1|In], _)
6   ;  async(A, U, D, [af((F2–H2), N, E)|Fs], S, M, [v(E) =:= 1|In], _),
7      async(A, U, D, [af((F1–H1), N, E)|Fs], S, M, [v(E) =:= 1|In], _)
8   ).
```

Here we can see both the choice being made based on the why-not height of the two subformulae, and how the with is scomposed. Compare this with the & rule in Figure 2.3. The cut at line 2 represents the main concept of the asynchronous phase: if an asynchronous connective is encountered the only thing we ought to do is to scompose it.

If a formula cannot be further be broken apart – i.e. it is either an atom, a negated atom, or it has a toplevel synchronous connective – then it is put to the side in `D`. This can be seen in the rule `to_delta` which implements the rule $R\Uparrow$

```
1 async(A, U, D, [F|Fs], S, M, In, _) :-
2   F = af((F1–_), _, _),
3   not(is_asy(F1)), !,
4   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 scompose it untill either an asynchronous connective is left or a negated atom. This is represented by the rules $D_1$ and $D_2$ that will be discussed further ahead in Section 3.4.3.

### 3.4.2 Identity rules

This process of alternating asynchronous and synchronous phases in classic focusing goes on untill we have a only positive literal (in our case a negated atom) in `Fs` 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. 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 in the clause

```
1 focus(A, U, D, F, _, _, In, node(id_1, A, U, D, F, [])) :-
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], In, Cns),
7   check(Cns).
```

A slightly different process 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 slight modification to APLL 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.

### 3.4.3 Decide rules

For the decide rules, particularly for $D_2$, we use a slight modification to APLL algorithm defined in Section 1.2.1. Like APLL we keep 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`

```
1 async(A, U, D, [], [H|T], M, In, node(decide_2, A, U, D, [], [Tree])) :-
2   \+ U = [],
3   gensym(x, X),
4   focus(A, U, D, af(H, X, E), T, M, [v(E) =:= 1|In], Tree).
5 async(A, U, D, [], [_|T], M, In, Tree) :-
6   \+ U = [],
7   async(A, U, D, [], T, M, In, Tree).
8 async(A, U, D, [], [], M, In, Tree) :-
9   \+ U = [],
10  refill(U, M, S, M1),
11  early_stop(A, U, D, S, M1, In, Tree).
```

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

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

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 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 put before the unrestricted counterparts, so that they are tried first.

## 3.5 Building the tree

In the listings above we omitted one parameter of the predicates, which purpose is to build the proof tree. At each call of async and focus one node of the proof tree is built. This contains the context of the call. 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 the node: a label, the context and an – optionally empty – list of subtrees. A leaf is just a node with an empty list of subtrees.

This tree can be used in the end to reconstruct the actual proof tree, by visiting it and – for each formula – querying whether its variable is set of one, and cancelling it otherwise. A more sophisticated algorithm may even cancel out unwanted unrestricted formulae, that otherwise remain lingering in the sequent.

# Chapter 4

# Testing

## 4.1 Infrastructure

### 4.1.1 Reproducibility

The prover's tests and benchmarks are made using a jupyter notebook. To ensure reproducibility we use Nix, which is a build system based on reproducile and declarative recipes, called *derivations*. A docker image is also given, which in turn calls nix without the need to install it on the system. Furthermore for the project infrastructure we used nix flakes, which are an experimental feature of nix that makes the process more hermetic and pure.

### 4.1.2 Prefix format

Since for benchmarking we will interface with a lot of different provers, each with its own syntax for expressions, the need for a common format which was easy to parse and translate arose. For this purpose we define use 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 apices as in `'varname'`. As an example we give the translation of DeMorgan for the tensor:

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

### 4.1.3 File formats

As a standard format to store the tests we use json because of its vast adoption by most programming languages. A test suite is defined as a list of test cases

```
TestCase ::= {
  "id": <Number>,
  "premises": [ <PrefixFormula>+ ],
  "conclusions": [ <PrefixFormula> ]
}
```

where

**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

Other arbitrary fields may be present, for example we will use the following optional fields:

**thm**
: whether this test case is a tautology or not, may be null;

***, , +, ...**
: the number of times a specific appears in the test case;

**notes**
: human readable text about the test case, for example its infix representation;

**size**
: an indicative number of the size of the formula;

**atoms**
: the upper bound on the number of atoms.

## 4.2 Benchmarking

We use llprover's test suite to prove our own. This test suite is composed mainly of simple tautologies, so it's only purpose is to quickly determine that an iteration of the prover is free of trivial bugs.

The problem with datasets with a high number of exponential formulae is the high number of timeouts and failures. Failures happen when the bound is too small for a formula, and all the branches are exhausted before the timeout.

Nervertheless we use test our prover using some of the tests from LLTP, like Kleene's fromulae using call-by-name or call-by-value translations. We run our provers using these two datasets, composed each of 88 cases, using a bound of 3 and a timeout of 60 seconds.

We now show how our prover compares mainly to APLL, which is the only prover we found using the backwards method with for full classical linear logic. The adapt the random formula generator from APLL to make it ablee to choose which connectives to use. And we generate some datasets of 100 formulae in mll, all, mall and finally cll.

Kleene CBN

| prover | timeouts | failures | successes | success rate | average time (succ) |
|---|---|---|---|---|---|
| APLL | ??? | ??? | ??? | ??? | ??? |
| llprover | ??? | ??? | ??? | ??? | ??? |
| sat-ll | ??? | ??? | ??? | ??? | ??? |

Kleene CBV

| prover | timeouts | failures | successes | success rate | average time (succ) |
|---|---|---|---|---|---|
| APLL | 0 | 17 | 71 | $\approx 0.80$ | 0.326s |
| llprover | 20 | 6 | 62 | $\approx 0.70$ | 2.179s |
| sat-ll | 5 | 15 | 68 | $\approx 0.77$ | 0.496s |

### 4.2.1 Results

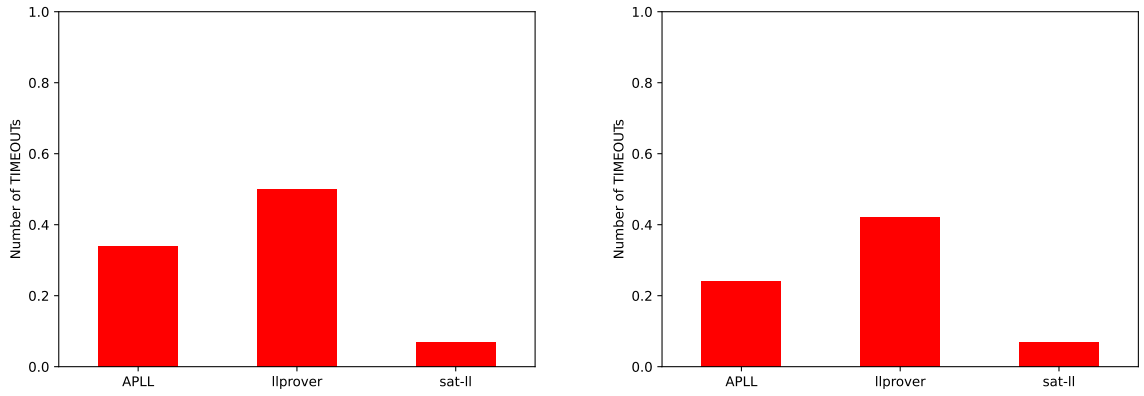
Using random formulae we can clearly see that our prover outperforms APLL (and llprover) with formulae rich in multiplicatives. As soon as exponentials come into play the differences smooth out.

Most of these outcomes see a persistent slght delay in the times of our prover in respect to APLL ones. This is to be expected since OCaML and is to be ascibed to the interpreted nature of Prolog.

Since llprover uses incremental search, its times are one of the slowest.

All tests are run with a timeout of 600 seconds, and an indicative bound of 5. Each prover treats the bound differently so sometimes a provers prematurely fail because of a small bound.

We run the tests on a dataset of 100 random normalized multiplicative formulae. These results show what we said before, APLL choice for splitting greatly hampers it's ability with formulae rich in multiplicatives, to the point that APLL consistently exhausted its time on almont a fourth of the formulae tried. ...




(a) Multiplicative case (b) Multiplicative and additive case

Figure 4.1: Percentage of number of timeouts out of a hundred formulae

The additive case is not that significant, since the only choice is the one for plus, the formulae remain manageable and no major differences can be seen.


all

We can see that in the multiplicative and additive case the difference begin to level, Finally in classical linear logic our prover performs slightly worse than APLL.

# Bibliography

[1] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 1987.

[2] James Harland and David Pym. Resource-distribution via boolean constraints. 2018.

[3] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 2009.

[4] Paul Tarau and Valeira de Paiva. Deriving theorems in implicational linear logic. 2020.