Chapter 1

Intro

1.1 State of the art

1.1.1 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

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

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 ...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 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`

```
1 let rec split_list_aux (acc1, acc2) l k = match l with
2   | [] -> acc1, acc2
3   | hd :: tl ->
4       if k mod 2 = 0 then split_list_aux (acc1, hd :: acc2) tl (k / 2)
5       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

```
1 (* ... *)
2 | Tensor (g, h) ->
3   let rec split_gamma k =
4     if k = -1 then None
5     else
6       let gamma1, gamma2 = split_list gamma k in
7         try
8           (* ... *)
9         with NoValue ->
10          split_gamma (k - 1)
11  in
12    let k = fast_exp_2 (List.length gamma) - 1 in
13      (* ... *)
```

As we will see in ... 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 called `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 (line 7) after being refilled 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 Prolog's choice

Prolog as a language and as an environment has been historically tied to automated theorem proving for its ability to express logical inference and backtracking algorithmns

naturally. One particularly conventient characteristic of Prolog is its automatic managing of backtracking, whereas in most other languages we would have had to use exceptions to walk down the stack, or a queue of unfinished computations.

Most Prolog implementations also support CLP or constraint logic programming. This allows to have constraints on 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.
```

Chapter 2

The focused calculus

Before describing the calculus we must give some definitions

Definition 2 (Annotated formula)**.** Given a formula ϕ 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 Δ.

As seen in Figure 2.1 and 2.2 we will usually use ϕ 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 Ψ of unrestricted formulae, or all formulae that can be freely discarded or duplicated;
- the multisets Δ and Φ 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 Λ of constraints, which is to be interpreted as the conjunction of its members, and the set V which represents a propagated solution. Furthermore if Λ 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 Λ "go-up" the proof tree and solutions V "come down" from the leaves.

Definition 5 (Splitting a sequent)**.** Given a sequent of annotated formulae Δ 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 Δ, and ϕ_i and e_i respectively the formula and the variable of the i-eth annotated formula in Δ using an arbitrary order.

It is worth noting that the variables $x_i \in X$ used for Δ^L and Δ^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 (Δ_L, Δ_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 ϕ we define the following predicates

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

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

- ϕ neg lit is a predicate that's true only when ϕ 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.

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

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

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

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

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

(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

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

$$[I_2]\ \frac{e_1 \text{ used}, \Delta \neg\text{used}, \Lambda \downarrow V}{\vdash \Psi, \phi : \Delta \Downarrow \mathrm{af}(\phi^{\perp}, e_1) \parallel \Lambda : V}\ [D_2]\ \frac{\phi\ \neg\text{neg lit} \qquad e \text{ new} \qquad \vdash \Psi : \Delta \Downarrow \mathrm{af}(\phi, e) \parallel e \text{ used}, \Lambda : V}{\vdash \Psi, \phi : \Delta \Uparrow . \parallel \Lambda : V}$$

(c) Identity and decide rules

Figure 2.3: The complete focused constraint calculus

Chapter 3

Implementation details