



# Proof Search in Propositional Linear Logic via Boolean Constraints Satisfaction

Laurea Triennale in Informatica

**Martino D'Adda** (964827)

16 Luglio 2024



UNIVERSITÀ  
DEGLI STUDI  
DI MILANO



# Indice

## 1 Introduzione

► Introduzione

► Il calcolo

► Conclusione



# Calcolo dei sequenti

## 1 Introduzione

Il calcolo dei sequenti (G. Ghentzen, 1934) è un formalismo per rappresentare un calcolo logico e strutturarne dimostrazioni. I concetti principali sono due:



# Calcolo dei sequenti

## 1 Introduzione

Il calcolo dei sequenti (G. Ghentzen, 1934) è un formalismo per rappresentare un calcolo logico e strutturarne dimostrazioni. I concetti principali sono due:

- il sequente  $\Delta \vdash \Gamma$ , che esprime una implicazione tra la congiunzione di  $\Delta$  e la disgiunzione di  $\Gamma$ ;



# Calcolo dei sequenti

## 1 Introduzione

Il calcolo dei sequenti (G. Ghentzen, 1934) è un formalismo per rappresentare un calcolo logico e strutturarne dimostrazioni. I concetti principali sono due:

- il sequente  $\Delta \vdash \Gamma$ , che esprime una implicazione tra la congiunzione di  $\Delta$  e la disgiunzione di  $\Gamma$ ;
- la regola, che descrive come si possono comporre i sequenti, ad esempio

$$\wedge \frac{\overbrace{\Delta \vdash \phi', \Gamma}^{\Pi'} \quad \overbrace{\Delta \vdash \phi'', \Gamma}^{\Pi''}}{\underbrace{\Delta \vdash \phi' \wedge \phi'', \Gamma}_{\Pi}}$$

stabilisce che se valgono  $\Pi'$  e  $\Pi''$ , allora vale  $\Pi$ .



# Calcolo dei sequenti cont'd

## 1 Introduzione

Una dimostrazione consiste in un albero avente come radice il sequente da dimostrare e ottenuto concatenando regole.

$$\begin{array}{c} \frac{A \frac{}{a, b \vdash b} \quad A \frac{}{a, b \vdash a}}{\wedge_R \frac{}{a, b \vdash b \wedge a}} \\ \wedge_L \frac{}{a \wedge b \vdash b \wedge a} \end{array}$$

Un **theorem prover** è un programma che, dato un sequente, prova a costruirne la dimostrazione applicando le regole di sopra. Nello specifico un theorem prover è detto bottom-up se parte dal sequente da dimostrare, e lo continua a scomporre applicando le regole “al contrario”.



# Logica lineare

## 1 Introduzione

La **logica lineare** (J.-Y. Girard, 1987) è una logica la cui caratteristica principale è che è generalmente proibita la duplicazione o l'eliminazione di formule, operazioni che altrimenti sono spesso lasciate implicite. Questa scelta porta ad alcune proprietà particolari, tra cui la duplicazione dei connettivi della logica classica secondo due interpretazioni diverse, una detta moltiplicativa e l'altra additiva.

	$\wedge$	$\vee$	$\top$	$\perp$
Molt.	$\otimes$	$\wp$	<b>1</b>	$\perp$
Add.	$\&$	$\oplus$	$\top$	<b>0</b>



# Splitting

## 1 Introduzione

Durante il proof searching bottom-up uno dei problemi principali è quello dello **splitting**.

$$\otimes \frac{\vdash \Delta', \phi' \quad \vdash \Delta'', \phi''}{\vdash \Delta', \Delta'', \phi' \otimes \phi''}$$

La scomposizione dei moltiplicativi, ad esempio nella regola del tensore di cui sopra, comporta il partizionamento di  $\Delta$  in  $\Delta'$  e  $\Delta''$  per continuare la dimostrazione. Questa operazione può richiedere un numero esponenziale di tentativi.

Nel 2001 D.Pym e J.Harland propongono un metodo per gestire lo splitting utilizzando vincoli booleani generati a partire da espressioni associate a ogni singola formula.





# Indice

## 2 Il calcolo

► Introduzione

► **Il calcolo**

► Conclusione



# Focusing e normalizzazione

## 2 Il calcolo

Partendo dal calcolo citato sopra di D.Pym e J.Harland, abbiamo applicato due classiche modifiche nell'ambito del proof searching (i.e. normalizzazione e focusing). Di questo nuovo calcolo – che chiameremo  $C$  – abbiamo poi mostrato la correttezza esibendo una traduzione verso un calcolo intermedio ( $A$ ) tale che il seguente diagramma commuta:

$$\begin{array}{ccc} C & \xrightarrow{\llbracket - \rrbracket_C} & A \\ & \searrow \text{dashed} & \downarrow \llbracket - \rrbracket_A \\ & \llbracket - \rrbracket_A \circ \llbracket - \rrbracket_C & LL \end{array}$$

con  $LL$  il calcolo classico della logica lineare e  $\llbracket - \rrbracket$  le rispettive traduzioni.



# Implementazione

## 2 Il calcolo

Per il calcolo di sopra sono state scritte:



# Implementazione

## 2 Il calcolo

Per il calcolo di sopra sono state scritte:

- un'implementazione in SWI-Prolog;





# Implementazione

## 2 Il calcolo

Per il calcolo di sopra sono state scritte:

- un'implementazione in SWI-Prolog;
- un generatore di test in OCaml;





# Implementazione

## 2 Il calcolo

Per il calcolo di sopra sono state scritte:

- un'implementazione in SWI-Prolog;
- un generatore di test in OCaml;
- una libreria per il testing e il benchmarking in Python.





# Implementazione

## 2 Il calcolo

Per il calcolo di sopra sono state scritte:

- un'implementazione in SWI-Prolog;
- un generatore di test in OCaml;
- una libreria per il testing e il benchmarking in Python.

Infine l'infrastruttura per tutti i programmi di sopra è stata gestita con Nix.





# Indice

## 3 Conclusione

► Introduzione

► Il calcolo

► Conclusione

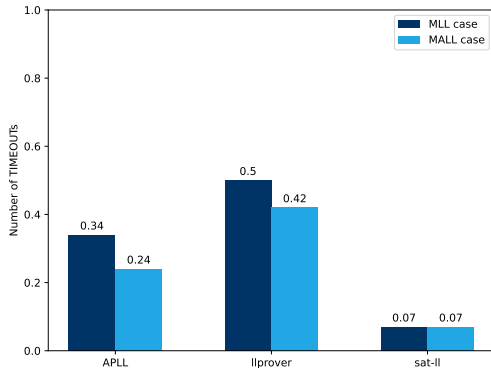




## Risultati – caso moltiplicativo

### 3 Conclusione

Una volta implementato il prover lo si è confrontato con altri due prover basati a grandi linee sullo stesso algoritmo (i.e. bottom-up focused proof search): Ilprover (1997) e APLL (2016). Nel caso moltiplicativo – quello principalmente soggetto al problema dello splitting – i risultati sono buoni:





# Risultati – caso generale

## 3 Conclusione

Nel caso dei test esponenziali utilizziamo due dataset preesistenti e vediamo che i risultati si livellano:

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

(a) Output per KLE-cbv

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

(b) Output per KLE-cbn



*Fine.*