User's Guide of a Linear Logic Theorem Prover (llprover)

Naoyuki Tamura (tamura@kobe-u.ac.jp)
Department of Computer and Systems Engineering
Faculty of Engineering
Kobe University, Japan
http://bach.seg.kobe-u.ac.jp/llprover/

May 29, 1998

1 Introduction

This program finds a cut-free proof of a given sequent (however, cut rules are used when axioms are given). That is, it creates a proof from bottom to up by applying applicable rules. The CLL-X system used here does not have exchange rules explicitly (see Appendix A). Therefore, the resulting proof does not include exchange rules.

In addition, this program only searches proofs in which the number of cut and contraction rules is limited by the given threshold value. Therefore, in this program, the failure of searching proofs does not mean the unprovability. It means there exist no proofs which use the cut and contraction rules less or equal times to the threshold value.

There was a bug for processing eigen variables in version 1.1beta. It is fixed in this version.

2 How to use

The program (`llprover.pl`) is written in SICStus Prolog. Please use it as follows.

```
% prolog
SICStus 2.1 #3: Thu May 20 21:08:04 JST 1993
| ?- compile(llprover).
{compiling llprover.pl...}
{llprover.pl compiled, 10330 msec 120016 bytes}

yes
| ?- ll.
Linear Logic Prover ver 1.1 for SICStus Prolog
       by Naoyuki Tamura (tamura@kobe-u.ac.jp)
cll(full)> a->b->c --> b->a->c.
Trying to prove with threshold = 0
Succeed in proving a->b->c --> b->a->c (119 msec.)
twosided:pretty:1 =
            ------- Ax  ------- Ax
            b --> b     c --> c
------- Ax  ------------------- L->
a --> a        b,b->c --> c
--------------------------- L->
     a,b,a->b->c --> c
     ------------------ R->
     b,a->b->c --> a->c
     ------------------- R->
     a->b->c --> b->a->c
yes
cll(full)> quit.
yes
Exit from Linear Logic Prover...
Total CPU time = 270 msec.

yes
| ?- halt.
```

Please do not forget to add period (".") at the end of your input!
You can abort the execution by typing `C-c` (Control `C`).

```
Prolog interruption (h for help)? a
{ Execution aborted }
| ?-
```

3 Notations

Syntax follows SICStus Prolog. But some operators are redefined as follows.

```
:- op(1200, xfy, & ).
:- op(1190, xfx, [ -->, <--> ]).
:- op(1000, fx, @ ).
:- op( 910, xfy, -> ).
:- op( 500, xfy, [ +, /\, \/ ]).
:- op( 400, xfy, * ).
:- op( 300, fy,  [ ~, !, ? ]).
```

Smaller number means stronger (narrower) scope. Also, `xfx` means an infix operator, `xfy` means a right associative operator (e.g. $A+B+C = A+(B+C)$), `fy` means a prefix operator.
Please follow the following notation to represent variables, etc.

- Variables: Please use Prolog variables (uppercase letters): `X`, `Y`, `X0`, ..., etc.
- Constants: Please use Prolog atoms (lowercase letters): `a`, `b`, `1`, ..., etc.
- Function and Predicate symbols: Please use Prolog functor symbols (lowercase letters): `a`, `b`, `c`, ..., etc.

Logical connectives are written as follows (please refer [1],[2] for Girard's notation, and please refer [3] for Troelstra's notation):

Notation used here	Troelstra's notation	Girard's notation
`~A`	$\sim A$	$A^{\perp}$
`A*B`	$A \star B$	$A \otimes B$
`A+B`	$A + B$	$A \wp B$ ($A ✠ B$)
`1`	$\mathbf{1}$	$\mathbf{1}$
`0`	$\mathbf{0}$	$\perp$
`A/\B`	$A \sqcap B$	$A \,\&\, B$
`A\/B`	$A \sqcup B$	$A \oplus B$
`top`	$\top$	$\top$
`bot`	$\perp$	$\mathbf{0}$
`A->B`	$A \multimap B$	$A \multimap B$
`all(X,A)`	$\forall x.A$	$\bigwedge x.A$
`exists(X,A)`	$\exists x.A$	$\bigvee x.A$
`!A`	$!A$	$!A$
`?A`	$?A$	$?A$

Please note that `0` and `bot` are used differently from Girard's notation.

Sequents are written as follows:

Notation used here	Troelstra's notation	
`A1,A2,...,Am --> B1,B2,...,Bn`	$A_1, A_2, \ldots, A_m \Rightarrow B_1, B_2, \ldots, B_n$	$(m, n > 0)$
`A1,A2,...,Am --> []`	$A_1, A_2, \ldots, A_m \Rightarrow$	$(m > 0)$
`[] --> B1,B2,...,Bn`	$\Rightarrow B_1, B_2, \ldots, B_n$	$(n > 0)$
`[] --> []`	$\Rightarrow$	

Some examples:

Notation used here	Troelstra's notation
`1,a,b --> a*b,0`	$\mathbf{1}, A, B \Rightarrow A \star B, \mathbf{0}$
`a->b->c --> b->a->c`	$A \multimap (B \multimap C) \Rightarrow B \multimap (A \multimap C)$
`a->b/\c --> (a->b)/\(a->c)`	$A \multimap (B \sqcap C) \Rightarrow (A \multimap B) \sqcap (A \multimap C)$
`~ ~a --> a`	$\sim\sim A \Rightarrow A$
`!(a/\b) --> !a * !b`	$!(A \sqcap B) \Rightarrow !A \star !B$
`!all(X,a(X)) --> all(X,!a(X))`	$!\forall x.A(x) \Rightarrow \forall x.!A(x)$

4 Selecting a System

You can use the following systems described in [3]. Precisely speaking, CLL, ILZ, and ILL are CLL-X, ILZ-X, and ILL-X in Appendix A respectively.

System				Notation used here			
CLL	$\mathbf{CLL}_q$	$\mathbf{CLL}_e$	$\mathbf{CLL}_0$	`cll(full)`	`cll(q)`	`cll(e)`	`cll(0)`
ILZ	$\mathbf{ILZ}_q$	$\mathbf{ILZ}_e$	$\mathbf{ILZ}_0$	`ilz(full)`	`ilz(q)`	`ilz(e)`	`ilz(0)`
ILL	$\mathbf{ILL}_q$	$\mathbf{ILL}_e$	$\mathbf{ILL}_0$	`ill(full)`	`ill(q)`	`ill(e)`	`ill(0)`

Currently selected system is shown in the prompt message. You can select other systems by inputing the system name.

```
cll(full)> ill(0).
yes
ill(0)>
```

Initial selection is `cll(full)`.

5 Setting Threshold Value

You need to specify the maximum usage number of contraction and cut rules to use this prover (cut rules are only used for axioms). At first, it searches proofs with zero use of `C!`, `C?`, `Cut` rules for each path of the proof, then proofs with at most one use of the rules for each path of the proof, then at most two uses..., then finally stops when the limit is greater than the threshold value.

Current setting of the threshold value is shown by `threshold(_)` command. You can set the new value by `threshold(`*n*`)`. The initial value is 5 (you might feel this is very small...).

```
cll(full)> threshold(_).
threshold(5)
yes
cll(full)> threshold(10).
yes
cll(full)> threshold(_).
threshold(10)
yes
```

6 Setting Axioms

You can set axioms as follows:

cll(full)> axioms([Axioms$_1$,Axioms$_2$,...,Axioms$_n$]).

All axioms are formulas (you can not use sequents). Also, you can not write axiom schema (by using Prolog variables).

```
ill(0)> axioms([((A->0)->0)->A]).
Error ((_127->0)->0)->_127 is not a formula.
```

Current axioms can be examined by `axioms(_)`. Initial setting is `axioms([])`.

The prover searches cut-free proofs when no axioms are set, but it searches proofs with cuts when some axioms are set. Therefore, in this case, it will be relatively slow even in the propositional fragment.

```
cll(full)> ill(0).
yes
ill(0)> axioms(_).
axioms([])
yes
ill(0)> (a->0)->(b->0) --> b->a.
Trying to prove with threshold = 0 1 2 3 4 5
Fail to prove (a->0)->b->0 --> b->a (1170 msec.)
yes
ill(0)> axioms([ ((a->0)->0)->a ]).
yes
ill(0)> (a->0)->(b->0) --> b->a.
Trying to prove with threshold = 0 1
Succeed in proving (a->0)->b->0 --> b->a (7119 msec.)
twosided:pretty:2 =
                                         ------- Ax  ------- Ax
                                         b --> b     0 --> 0
                       ------------- Ax  ------------------- L->
                       a->0 --> a->0        b,b->0 --> 0
                       --------------------------------- L->
                           a->0,b,(a->0)->b->0 --> 0
                           ---------------------------- R->  ------- Ax
                           b,(a->0)->b->0 --> (a->0)->0      a --> a
------------------- Axiom  ----------------------------------------- L->
 --> ((a->0)->0)->a           ((a->0)->0)->a,b,(a->0)->b->0 --> a
----------------------------------------------------------------- Cut
                      b,(a->0)->b->0 --> a
                      --------------------- R->
                      (a->0)->b->0 --> b->a
yes
```

7 Selecting a Style

You can select one of the following styles.

Command	Style
`style(twosided)`	Two-sided calculus
`style(onesided)`	One-sided calculus
`style(proofnet)`	Proof Net PN1 (see [1])

The prover searches two-sided proofs, then convert the proof to the selected style (see Appendix B). You can examine the current style by `style(_)`. Initial setting is `style(twosided)`.

In `style(twosided)`, axiom lines are not drawn (it is too difficult for me...). They are shown by using labels, such as `Ax(1)`. Also, $\sqcap$ rules are not converted (it is also difficult for me...).

```
cll(full)> style(proofnet).
yes
cll(full)> a->b->c --> b->a->c.
Trying to prove with threshold = 0
Succeed in proving a->b->c --> b->a->c (199 msec.)
proofnet:pretty:3 =
       Ax(1)  Ax(3)       Ax(2)  Ax(3)
       ~a     c           b      ~c
Ax(2)  -------- +  Ax(1)  --------- *
~b       ~a+c      a        b* ~c
------------- +    -------------- *
  ~b+ ~a+c             a*b* ~c
yes
```

8 Selecting an Output Style

You can select one of the following output styles.

Command	Output Style
`output(pretty)`	Proof tree (character base)
`output(tex)`	LaTeX output using `proof.sty` by Makoto Tatsuta (`tatsuta@riec.tohoku.ac.jp`)
`output(standard)`	Indented style
`output(term)`	Internal form

Current selection can be examined by `output(_)`. Initial setting is `output(pretty)`.

The most difficult part of this prover program is the output routine for `output(pretty)` (more difficult than the automatic proving!).

The output routine for `output(tex)` is written by Eiji Sugiyama at first. Variables are replaced by x, y, z, x_1, y_1, z_1, ..., etc. First letters of predicate symbols are replaced by uppercase letters. The following is an example.

```
cll(full)> output(tex).
yes
cll(full)> !all(X,a(X)) * !all(Y,a(Y)) --> all(Z,a(Z)).
Trying to prove with threshold = 0
Succeed in proving !all(_119,a(_119))*!all(_176,a(_176)) --> all(_523,a(_523)) (94 msec.)
twosided:tex:4 =
\begin{displaymath}
\infer[\LR{\tprod}]{! (\lall x.A(x)) \tprod ! (\lall y.A(y)) \drv \lall z.A(z)}{
  \infer[\RR{\lall}]{! (\lall x.A(x)), ! (\lall y.A(y)) \drv \lall z.A(z)}{
    \infer[\WR{!}]{! (\lall x.A(x)), ! (\lall y.A(y)) \drv A(z)}{
      \infer[\LR{!}]{! (\lall y.A(y)) \drv A(z)}{
        \infer[\LR{\lall}]{\lall y.A(y) \drv A(z)}{
          \infer[\Ax]{A(z) \drv A(z)}{}
        }
      }
    }
  }
}
\end{displaymath}
yes
```

If you put it in your LaTeX file, you can get a beautiful output (you need `proof.sty` by Makoto Tatsuta and `linear.sty` included in this package).

```
\documentstyle[proof,linear]{article}
\begin{document}
```

```
\begin{displaymath}
  .....
\end{displaymath}
\end{document}
```

$$\begin{prooftree}
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$A(z) \;\Rightarrow\; A(z)$}
\RightLabel{L$\forall$}
\UnaryInfC{$\forall y.A(y) \;\Rightarrow\; A(z)$}
\RightLabel{L!}
\UnaryInfC{$!(\forall y.A(y)) \;\Rightarrow\; A(z)$}
\RightLabel{W!}
\UnaryInfC{$!(\forall x.A(x)), !(\forall y.A(y)) \;\Rightarrow\; A(z)$}
\RightLabel{R$\forall$}
\UnaryInfC{$!(\forall x.A(x)), !(\forall y.A(y)) \;\Rightarrow\; \forall z.A(z)$}
\RightLabel{L$\star$}
\UnaryInfC{$!(\forall x.A(x)) \star !(\forall y.A(y)) \;\Rightarrow\; \forall z.A(z)$}
\end{prooftree}$$

You can produce the output in Girard's notation by adding `\girardnotation` before `\begin{displaymath}` or surrounding the `displaymath` environment with `\begin{girardnotation}` and `\end{girardnotation}`. If you use Girard's notation, $\vdash$ is used as sequent arrows.

9 Logging

You can record your log by `log(yes)` or `log('file')`. File `llprover.log` is used when you use `log(yes)`.

The command `log(no)` stops the logging. The command `log(_)` displays the current log file name.

In the log record, your input lines are not exactly same with what you actually input (especially if it includes some Prolog variables).

For example, even if you input

```
cll(full)> !all(X,a(X)) --> all(X,!a(X)).
```

The record will be:

```
cll(full)> !all(_290,a(_290))-->all(_290,!a(_290)).
```

10 Other Issues

- The command `quit` will terminate the prover program.
- The command `help` displays a summary of commands.
- You can input from a file by `[file]`.
- The input Γ`<-->`Δ proves both Γ`-->`Δ and Δ`-->`Γ (this is meaningless if Γ or Δ is not a single formula).
- The command `init` does the initialization (that is, `cll(full)`, `threshold(5)`, `axioms([])`, `style(twosided)`, `output(pretty)`, `log(no)`, and the reset of output counter).
- The positive integer n redisplays the n-th output. If n is negative, $|n|$-th previous output is redisplayed.
- You can specify multiple commands by using the delimiter `&`.
- The command @G executes Prolog goal G.

The package includes `llproverex` as an input sample. The result is `llproverex.log`.

11 Summary

This program is not efficient and might have some remaining bugs... But if you are OK, please use (and I am very happy). Please freely distribute or modify it.

Remaining problems are:

- Drawing links in `style(proofnet)` output
- Cut elimination
- Conversion to Natural deduction or Combinator system (such as `style(natural)`)

I will explain CLL-X system in the following Appendix.

A CLL-X

At first, we define CLLX as follows. CLLX includes exchange rules, but it will be proved later that they are redundant.

- Logical axiom and Structural rules:

$$\frac{}{A \Rightarrow A}\ \text{Ax}$$

$$\frac{\Gamma \Rightarrow \Delta_1, A, \Delta_2 \quad \Gamma'_1, A, \Gamma'_2 \Rightarrow \Delta'}{\Gamma'_1, \Gamma, \Gamma'_2 \Rightarrow \Delta_1, \Delta', \Delta_2}\ \text{Cut}$$

$$\frac{\Gamma_1, B, A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, A, B, \Gamma_2 \Rightarrow \Delta}\ \text{LX} \qquad \frac{\Gamma \Rightarrow \Delta_1, B, A, \Delta_2}{\Gamma \Rightarrow \Delta_1, A, B, \Delta_2}\ \text{RX}$$

- Rules for the propositional connectives:

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow A, \Delta}{\Gamma_1, \sim A, \Gamma_2 \Rightarrow \Delta}\ \text{L}\sim \qquad \frac{A, \Gamma \Rightarrow \Delta_1, \Delta_2}{\Gamma \Rightarrow \Delta_1, \sim A, \Delta_2}\ \text{R}\sim$$

$$\frac{\Gamma_1, A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, A \sqcap B, \Gamma_2 \Rightarrow \Delta}\ \text{L}\sqcap_1$$

$$\frac{\Gamma_1, B, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, A \sqcap B, \Gamma_2 \Rightarrow \Delta}\ \text{L}\sqcap_2 \qquad \frac{\Gamma \Rightarrow \Delta_1, A, \Delta_2 \quad \Gamma \Rightarrow \Delta_1, B, \Delta_2}{\Gamma \Rightarrow \Delta_1, A \sqcap B, \Delta_2}\ \text{R}\sqcap$$

$$\frac{\Gamma_1, A, B, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, A \star B, \Gamma_2 \Rightarrow \Delta}\ \text{L}\star \qquad \frac{\Gamma'_1 \Rightarrow \Delta'_1, A, \Delta'_2 \quad \Gamma''_1 \Rightarrow \Delta''_1, B, \Delta''_2}{\Gamma_1 \Rightarrow \Delta_1, A \star B, \Delta_2}\ \text{R}\star$$

$$\frac{\Gamma \Rightarrow \Delta_1, A, \Delta_2}{\Gamma \Rightarrow \Delta_1, A \sqcup B, \Delta_2}\ \text{R}\sqcup_1$$

$$\frac{\Gamma_1, A, \Gamma_2 \Rightarrow \Delta \quad \Gamma_1, B, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, A \sqcup B, \Gamma_2 \Rightarrow \Delta}\ \text{L}\sqcup$$

$$\frac{\Gamma \Rightarrow \Delta_1, B, \Delta_2}{\Gamma \Rightarrow \Delta_1, A \sqcup B, \Delta_2}\ \text{R}\sqcup_2$$

$$\frac{\Gamma'_1, A, \Gamma'_2 \Rightarrow \Delta'_1 \quad \Gamma''_1, B, \Gamma''_2 \Rightarrow \Delta''_1}{\Gamma_1, A + B, \Gamma_2 \Rightarrow \Delta_1}\ \text{L}+ \qquad \frac{\Gamma \Rightarrow \Delta_1, A, B, \Delta_2}{\Gamma \Rightarrow \Delta_1, A + B, \Delta_2}\ \text{R}+$$

$$\frac{\Gamma'_1, \Gamma'_2 \Rightarrow A, \Delta'_1 \quad \Gamma''_1, B, \Gamma''_2 \Rightarrow \Delta''_1}{\Gamma_1, A \multimap B, \Gamma_2 \Rightarrow \Delta_1}\ \text{L}\multimap \qquad \frac{A, \Gamma \Rightarrow \Delta_1, B, \Delta_2}{\Gamma \Rightarrow \Delta_1, A \multimap B, \Delta_2}\ \text{R}\multimap$$

Here, $\Gamma_i \in \text{Merge}(\Gamma'_i, \Gamma''_i)$ and $\Delta_i \in \text{Merge}(\Delta'_i, \Delta''_i)$ in R$\star$, L$+$, L$\multimap$ rules. The $\text{Merge}(\Gamma, \Delta)$ is a set of lists defined as follows:

- If Γ is empty, $\text{Merge}(\Gamma, \Delta) = \{\Delta\}$.
- If Δ is empty, $\text{Merge}(\Gamma, \Delta) = \{\Gamma\}$.

– If $\Gamma = A, \Gamma'$ and $\Delta = B, \Delta'$,
  $\mathrm{Merge}(\Gamma, \Delta) = \{A, X \mid X \in \mathrm{Merge}(\Gamma', \Delta)\} \cup \{B, Y \mid Y \in \mathrm{Merge}(\Gamma, \Delta')\}$.

- Rules for the propositional constants:

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, \mathbf{1}, \Gamma_2 \Rightarrow \Delta}\,\mathrm{L}\mathbf{1} \qquad \frac{}{\Rightarrow \mathbf{1}}\,\mathrm{R}\mathbf{1}$$

$$\frac{}{\Gamma \Rightarrow \Delta_1, \top, \Delta_2}\,\mathrm{R}\top$$

$$\frac{}{\mathbf{0} \Rightarrow}\,\mathrm{L}\mathbf{0} \qquad \frac{\Gamma \Rightarrow \Delta_1, \Delta_2}{\Gamma \Rightarrow \Delta_1, \mathbf{0}, \Delta_2}\,\mathrm{R}\mathbf{0}$$

$$\frac{}{\Gamma_1, \bot, \Gamma_2 \Rightarrow \Delta}\,\mathrm{L}\bot$$

- Rules for the quantifiers:

$$\frac{\Gamma_1, A[x/t], \Gamma_2 \Rightarrow \Delta}{\Gamma_1, \forall x.A, \Gamma_2 \Rightarrow \Delta}\,\mathrm{L}\forall \qquad \frac{\Gamma \Rightarrow \Delta_1, A[x/y], \Delta_2}{\Gamma \Rightarrow \Delta_1, \forall x.A, \Delta_2}\,\mathrm{R}\forall$$

$$\frac{\Gamma_1, A[x/y], \Gamma_2 \Rightarrow \Delta}{\Gamma_1, \exists x.A, \Gamma_2 \Rightarrow \Delta}\,\mathrm{L}\exists \qquad \frac{\Gamma \Rightarrow \Delta_1, A[x/t], \Delta_2}{\Gamma \Rightarrow \Delta_1, \exists x.A, \Delta_2}\,\mathrm{R}\exists$$

  Here, y is not free in the lower sequent in R$\forall$ and L$\exists$.

- Rules for the exponentials:

$$\frac{\Gamma_1, !A, !A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\mathrm{C}! \qquad \frac{\Gamma_1, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\mathrm{W}!$$

$$\frac{\Gamma_1, A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\mathrm{L}! \qquad \frac{!\Gamma \Rightarrow ?\Delta_1, A, ?\Delta_2}{!\Gamma \Rightarrow ?\Delta_1, !A, ?\Delta_2}\,\mathrm{R}!$$

$$\frac{\Gamma \Rightarrow \Delta_1, A, \Delta_2}{\Gamma \Rightarrow \Delta_1, ?A, \Delta_2}\,\mathrm{W}? \qquad \frac{\Gamma \Rightarrow \Delta_1, ?A, ?A, \Delta_2}{\Gamma \Rightarrow \Delta_1, ?A, \Delta_2}\,\mathrm{C}?$$

$$\frac{!\Gamma_1, A, !\Gamma_2 \Rightarrow ?\Delta}{!\Gamma_1, ?A, !\Gamma_2 \Rightarrow ?\Delta}\,\mathrm{L}? \qquad \frac{\Gamma \Rightarrow \Delta_1, A, \Delta_2}{\Gamma \Rightarrow \Delta_1, ?A, \Delta_2}\,\mathrm{R}?$$

Proposition 1 CLLX is equivalent to CLL.

(Proof) It is obvious because each rule of one system can be represented by using the corresponding rule of the other system and LX, RX rules.

Proposition 2 Cut is redundant is CLLX.

(Proof) Any cut-free proof of CLL is also a cut-free proof of CLLX.

Proposition 3 LX and RX rules are redundant in CLLX.

(Proof) The following can be shown easily: If $\Gamma \Rightarrow \Delta$ is provable in CLLX, any permutation $\Sigma \Rightarrow \Phi$ of $\Gamma \Rightarrow \Delta$ is provable in CLLX without LX and RX.

Now, we define CLL-X as CLLX excluding LX and RX rules. The system used in this theorem prover is CLL-X.

We can see the following propositions for CLL-X.

Proposition 4 If there are some axioms of the form $\Rightarrow A$ (A is a formula) in CLLX, it is possible to limit the usage of cuts as follows. Also, cut rules are redundant if there are no axioms.

$$\dfrac{\dfrac{}{\Rightarrow A}\,\text{Axiom} \qquad A, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\,\text{Cut}$$

(Proof) Obvious from the proof of the cut elimination theorem.

Proposition 5 In CLL-X, the rules immediately above C! can be limited to R$\star$, L$+$, L$\multimap$, C!, C?, and L!. Also, the rules immediately above C? can be limited to R$\star$, L$+$, L$\multimap$, C!, C?, and R?.

(Proof) C! and C? rules are:

$$\dfrac{\Gamma_1, !A, !A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{C!} \qquad \dfrac{\Gamma \Rightarrow \Delta_1, ?A, ?A, \Delta_2}{\Gamma \Rightarrow \Delta_1, ?A, \Delta_2}\,\text{C?}$$

We only prove for C!. The C? case can be proved similarly.
We consider cases depending on the rule R which appears immediately above the C! rule.

(1) Ax, Cut, R**1**, and L**0** can not be R.

(2) When R is R$\top$ or L$\bot$, the deduction of C! itself is redundant.

(3) When R is one of the rules for the propositional connectives except R$\star$, L$+$, and L$\multimap$.

$$\dfrac{\dfrac{\Gamma_1', !A, !A, \Gamma_2' \Rightarrow \Delta'}{\Gamma_1, !A, !A, \Gamma_2 \Rightarrow \Delta}\,\text{R}}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{C!} \quad \textit{or} \quad \dfrac{\dfrac{\Gamma_1', !A, !A, \Gamma_2' \Rightarrow \Delta' \qquad \Gamma_1'', !A, !A, \Gamma_2'' \Rightarrow \Delta''}{\Gamma_1, !A, !A, \Gamma_2 \Rightarrow \Delta}\,\text{R}}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{C!}$$

Since $!A$ can not be the principle formula, $!A$ should also appear in the upper sequent of R. Therefore, it can be transformed as follows:

$$\dfrac{\dfrac{\Gamma_1', !A, !A, \Gamma_2' \Rightarrow \Delta'}{\Gamma_1', !A, \Gamma_2' \Rightarrow \Delta'}\,\text{C!}}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{R} \quad \textit{or} \quad \dfrac{\dfrac{\Gamma_1', !A, !A, \Gamma_2' \Rightarrow \Delta'}{\Gamma_1', !A, \Gamma_2' \Rightarrow \Delta'}\,\text{C!} \qquad \dfrac{\Gamma_1'', !A, !A, \Gamma_2'' \Rightarrow \Delta''}{\Gamma_1'', !A, \Gamma_2'' \Rightarrow \Delta''}\,\text{C!}}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{R}$$

(4) When R is one of the rules for the quantifiers, similar to the case (3).

(5) When R is R!, W?, L?, or R?, similar to the case (3).

(6) When R is W!.

(6a) When $!A$ is the principal formula of W!.

$$\dfrac{\dfrac{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, !A, !A, \Gamma_2 \Rightarrow \Delta}\,\text{W!}}{\Gamma_1, !A, \Gamma_2 \Rightarrow \Delta}\,\text{C!}$$

This can be eliminated as a whole.

(6b) When $!A$ is not the principal formula of W!.

$$\cfrac{\cfrac{\Gamma_1', !A, !A, \Gamma_2' \ \Rightarrow\ \Delta}{\Gamma_1, !A, !A, \Gamma_2 \ \Rightarrow\ \Delta}\ \mathrm{W!}}{\Gamma_1, !A, \Gamma_2 \ \Rightarrow\ \Delta}\ \mathrm{C!}$$

This can be transformed as follows:

$$\cfrac{\cfrac{\Gamma_1', !A, !A, \Gamma_2' \ \Rightarrow\ \Delta}{\Gamma_1', !A, \Gamma_2' \ \Rightarrow\ \Delta}\ \mathrm{C!}}{\Gamma_1, !A, \Gamma_2 \ \Rightarrow\ \Delta}\ \mathrm{W!}$$

It is also possible to limit the usage of rules as follows, but the current implementation does not perform these.

- $!A$ and $?A$ should be split to different upper sequents in multiplicative rules.
- Application of L! on C! can be limited in the following case. Application of R? on C? can be considered similarly.

$$\cfrac{\cfrac{\Gamma_1, A, !A, \Gamma_2 \ \Rightarrow\ \Delta}{\Gamma_1, !A, !A, \Gamma_2 \ \Rightarrow\ \Delta}\ \mathrm{L!}}{\Gamma_1, !A, \Gamma_2 \ \Rightarrow\ \Delta}\ \mathrm{C!}$$

B Conversion to Other Styles

B.1 Conversion to One-sided Style

Each sequent $\Gamma \ \Rightarrow\ \Delta$ in CLL-X proof is converted to $\sim\Gamma, \Delta$. After the conversion, exchange rules might be implicitly used.

B.2 Conversion to Proof Net

Under preparation. Please look at the program `(^_^;)`

C Outline of the Algorithm

C.1 Internal Data Structure

Internal data structures of sequents are as follows:

$$\begin{array}{rcl} \langle\text{Sequent}\rangle & := & \langle\text{Formula List}\rangle\ \texttt{-->}\ \langle\text{Formula List}\rangle \\ \langle\text{Formula List}\rangle & := & \texttt{[}\ \langle\text{Formula}\rangle\ \texttt{,}\ \ldots\ \langle\text{Formula}\rangle\ \texttt{]} \end{array}$$

Internal data structures of proofs are as follows. In the definition of ⟨Proof⟩, ⟨Sequent⟩ means the lower sequent, and ⟨Proof⟩ means the proof of the upper sequents. ⟨Position⟩ `r(`n`)` (or `l(`n`)`) means the n-th position in the formula list in the sequent (left most position if $n = 0$).

```
⟨Proof⟩      :=  [ ⟨Rule Name⟩ , ⟨Formula Position⟩ , ⟨Sequent⟩ , ⟨Proof⟩ , . . . ⟨Proof⟩ ]
⟨Rule Name⟩  :=  ax | axiom | cut | l(~) | r(~) | l(/\,1) | l(/\,2) | r(/\) | l(*) | r(*) |
                 l(\/) | r(\/,1) | r(\/,2) | l(+) | r(+) | l(->) | r(->) |
                 l(1) | r(1) | r(top) | l(0) | r(0) | l(bot) |
                 l(all) | r(all) | r(exists) | l(exists) |
                 c(!) | w(!) | l(!) | r(!) | w(?) | c(?) | l(?) | r(?)
```

$$\begin{aligned}
\langle\text{Formula Position}\rangle &:= \texttt{[}\,\langle\text{Principal Formula Pos}\rangle\,\texttt{,}\,\langle\text{Sub-Formula Pos}\rangle\,\texttt{,}\ldots\langle\text{Sub-Formula Pos}\rangle\,\texttt{]}\\
\langle\text{Principal Formula Pos}\rangle &:= \texttt{[]} \mid \langle\text{Position}\rangle\\
\langle\text{Sub-Formula Pos}\rangle &:= \texttt{[]} \mid \langle\text{Position}\rangle \mid \texttt{[}\,\langle\text{Position}\rangle\,\texttt{,}\,\langle\text{Position}\rangle\,\texttt{]}\\
\langle\text{Position}\rangle &:= \texttt{r(}\,\langle\text{Natural Number}\rangle\,\texttt{)} \mid \texttt{l(}\,\langle\text{Natural Number}\rangle\,\texttt{)}
\end{aligned}$$

For example,

```
              ------- Ax  ------- Ax
              b --> b     c --> c
------- Ax    ------------------- L->
a --> a          b,b->c --> c
--------------------------- L->
     a,b,a->b->c --> c
     ------------------ R->
     b,a->b->c --> a->c
     ------------------- R->
     a->b->c --> b->a->c
```

the internal data structure of the above proof is as follows:

```
[r(->), [r(0),[l(0),r(0)]], ([a->b->c]-->[b->a->c]),
  [r(->), [r(0),[l(0),r(0)]], ([b,a->b->c]-->[a->c]),
    [l(->), [l(2),r(0),l(1)], ([a,b,a->b->c]-->[c]),
      [ax, [[]], ([a]-->[a])],
      [l(->), [l(1),r(0),l(0)], ([b,b->c]-->[c]),
        [ax, [[]], ([b]-->[b])],
        [ax, [[]], ([c]-->[c])]
      ]
    ]
  ]
]
```

C.2 Outline of the Algorithm

- Predicate `prove(`$+S$`,` $-P$`)`
  Predicate `prove(`S`,`P`)` searches a proof P which uses cuts and contractions less or equal to the threshold value for a given sequent S. If one proof is found, other proofs are not searched.

  (1) Erase internal database `proved/3` and `not_proved/2`.

  (2) Invoke `prove(`S`,`P`,`N`)` by incrementing N from 0 to the threshold value.

- Predicate `prove(`$+S$`,` $-P$`,` $+N$`)`
  Predicate `prove(`S`,`P`)` searches a proof P which uses cuts and contractions less or equal to the threshold value N for a given sequent S. If one proof is found, other proofs are not searched. If `prove(`S`,`P`,`N`)` fails and S is ground, register it to the internal database `not_proved(`S`,`N`)`.

  (1) If `not_proved(`S`,`M`)` is registered in the internal database and $N \leq M$, fail.

  (2) If `proved(`S`,`P`,`M`)` is registered in the internal database and $M \leq N$, succeed with the result proof P.

  (3) If S matches with user defined axioms, succeed.

  (4) If the system is **ILZ** or **ILL** and there are two or more formulas in the right-hand side of S, fail.

  (5) Select an applicable rule R for S and get the list of upper sequents Ss and the positions of principal formula and sub-formulas in Pos by calling `select_rule(`R`,`S`,`Ss`,`Pos`)`. If there are no applicable rules, `prove(`S`,`P`,`N`)` fail.

(6) If R is `cut`, add a constraint that the left upper sequent is a user's axiom. That is, the rule of the left upper sequent is unified with `axiom`.

(7) If R is `c(!)`, add a constraint that the applicable rules for the upper sequent is either `r(*)`, `l(+)`, `l(->)`, `c(!)`, `c(?)`, or `l(!)`. Currently, we use `freeze` predicate of SICStus Prolog to add the constraint.

(8) If R is `c(?)`, add a constraint that the applicable rules for the upper sequent is either `r(*)`, `l(+)`, `l(->)`, `c(!)`, `c(?)`, or `r(?)`. Currently, we use `freeze` predicate of SICStus Prolog to add the constraint.

(9) If R is either `cut`, `c(!)`, or `c(?)`, decrement N by 1. If $N \leq 0$, backtrack to `select_rule`.

(10) Call `prove(`S_i`,`P_i`,`N`)` recursively for each sequent S_i in Ss. If any call fails, backtrack to `select_rule`.

(11) Let $P = [R, Pos, S, P_1, \ldots, P_n]$, and register `proved(`$S$`,`$P$`,`$N$`)` to the internal database.

- Predicate `select_rule(`$-R$`, `$+S$`, `$-Ss$`, `$-Pos$`)`
  Predicate `select_rule(`R`,`S`,`Ss`,`Pos`)` selects an applicable rule R for the given sequent S and returns the list of upper sequents in Ss and the positions of the principal formula and sub-formulas in Pos. By backtracking, it returns all possible Ss and R. However, if one of invertible rules (that is, rules other than `l(/\,`i`)`, `r(\/,`i`)`, `r(*)`, `l(+)`, `l(->)`, `l(all)`, `r(exists)`, `c(!)`, `w(!)`, `l(!)`, `w(?)`, `c(?)`, `r(?)`) is applicable, it is selected deterministically (other Ss and R are not searched). Eigen variable condition is processed by `eigen_variable`.

- Predicate `eigen_variable(`$+V$`, `$+X$`)`
  V is a variable y in R$\forall$ and L$\exists$ of CLL-X, and X is the lower sequent. Predicate `eigen_variable(`V`,`X`)` adds a constraint that all free variables in X is different from V by using `dif` predicate of SICStus Prolog. Also, V is protected from binding it with other terms (except variables) by using `freeze(`V`,fail)` (V might be instantiated in `ax`).

C.3 Using the Prover from Other Programs

The prover can be used from other Prolog programs as follows:

```
:- compile(llprover).

use_llprover :-
    ll_init,
    set_system(ill,0),
    set_threshold(0),
    set_axioms([]),
    read(S0),
    convert_to_seq(S0, S),
    prove(S, P),
    set_style(onesided),
    set_output_form(pretty),
    print_proof(P).
```

References

[1] J.-Y. Girard: Linear Logic. Theoretical Computer Science, 50:1–102, 1987.

[2] J.-Y. Girard, P. Taylor, Y. Lafont: Proofs and Types. Cambridge University Press, 1988.

[3] A. S. Troelstra: Lectures on Linear Logic. CSLI Lecture Notes No.29, 1992.