A Formally Verified Cut-Elimination Procedure for Linear Nested Sequents for Tense Logic

Caitlin D'Abrera[1], Jeremy Dawson[1(✉)], and Rajeev Goré[2]

[1] School of Computing, The Australian National University, Canberra, Australia
{caitlin.dabrera,Jeremy.Dawson}@anu.edu.au
[2] Vienna University of Technology, Vienna, Austria

Abstract. We port Dawson and Goré's general framework of deep embeddings of derivability from Isabelle to Coq. By using lists instead of multisets to encode sequents, we enable the encoding of genuinely substructural logics in which some combination of exchange, weakening and contraction are not admissible. We then show how to extend the framework to encode the linear nested sequent calculus $\mathsf{LNS}_{\mathsf{Kt}}$ of Goré and Lellmann for the tense logic Kt and prove cut-elimination and all required proof-theoretic theorems in Coq, based on their pen-and-paper proofs. Finally, we extract the proof of the cut-elimination theorem to obtain a formally verified Haskell program that produces cut-free derivations from those with cut. We believe it is the first published formally verified computer program for eliminating cuts in any proof calculus.

Keywords: Formalised proof theory · Cut-elimination · Linear nested sequent calculus · Tense logic · Coq · Extraction · Program synthesis

1 Introduction

Traditional styles of proof calculi for capturing the notion of logical derivations include Hilbert calculi, natural deduction [19], sequent calculi [19] and tableau calculi [7]. More recent and elaborate styles include display calculi [1,3], labelled sequents [14] and nested sequent calculi [10]. Each style has strengths and weaknesses: expressivity, complexity, ease of use, and philosophical motivations.

Consider the interesting case of systems for tense logics. After previously published failed attempts at providing sequent calculi that satisfy cut-elimination, more complex systems were produced in the forms of display calculi, nested sequents and labelled sequents. Goré and Lellmann [8] provide a simpler calculus, $\mathsf{LNS}_{\mathsf{Kt}}$, using linear nested sequents (LNS) for the minimal tense logic Kt that

C. D'Abrera—Supported by an Australian Government Research Training Program Scholarship.

R. Goré—Work supported by the FWF projects I 2982 and P 33548.

A. Das and S. Negri (Eds.): TABLEAUX 2021, LNAI 12842, pp. 281–298, 2021.
https://doi.org/10.1007/978-3-030-86059-2_17

does not require the heavy machinery found in the other more complex systems. Their LNS calculus is a cut-free system and they proved cut-admissibility.

But, as is well-known, proofs of cut-admissibility are notoriously "case heavy" with many similar cases which are often "left to the reader". The cut-admissibility proof of Goré and Lellmann [8] is no exception and involves a complicated simultaneous induction over four sub-cases. One way of verifying its correctness is to formalise it in a modern interactive proof assistant, as explained next.

Dawson and Goré [6] gave an elegant embedding of a general framework for derivability in Isabelle/HOL, applicable to many styles of proof calculi, including many extensions of the sequent calculus [4]. But they used Isabelle/HOL 2007 and it would require a complete rewrite of that material to use it in modern Isabelle, which may indeed not be possible. So we ported this work to Coq and adapted it to allow the handling of genuinely substructural notions such as exchange, which the aforementioned Isabelle/HOL formalisation lacked. We extended this Coq framework to encode linear nested sequents and $\mathsf{LNS}_{\mathsf{Kt}}$, and proved in Coq the standard structural proof-theoretic theorems up to and including cut-admissibility, based on the original pen-and-paper proofs [8].

Constructively proving cut-elimination in Coq permits us to extract the procedure into a Haskell program that computes cut-free derivations from those with cut. The full proof of cut-elimination in a pen-and-paper setting is already large because it involves so many cases, and adding the extra cases that emerge as part of the task of formalising has made this theorem a good candidate for verification, particularly as multiple mistakes were found. As far as we know, ours is the first published extracted program for performing cut-elimination.

Our Coq code was developed with Coq 8.8.2 (October 2018) and tested with Coq 8.10.2 (Nov 2019): https://github.com/caitlindabrera/LNS-for-Kt.

2 Preliminaries

Formulae of normal modal tense logics are built from a given set Atm of atomic formulae via the following BNF grammar where $p \in Atm$: $A := p \mid \bot \mid A \to A \mid \Box A \mid \blacksquare A$. We assume that the reader is familiar with their associated Kripke semantics or their standard Hilbert calculus [8].

We define linear nested sequents as in [8] before giving the full calculus.

2.1 A Linear Nested Sequent Calculus for Kt

Definition 1. *A* sequent *is an expression* $\Gamma \Rightarrow \Delta$ *where the* antecedent Γ *and the* succedent Δ *are finite, possibly empty, multisets of formulae. We write* ϵ *to stand for an empty antecedent or succedent to avoid confusion. A* linear nested sequent *is a sequence of sequents where each adjacent pair is connected by* $\nearrow$ *or* $\swarrow$*. The sequents that occur within such a linear nested sequent are* components.

$$\frac{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta,A\swarrow\Sigma\Rightarrow\Pi,\Box A\quad\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\swarrow\Sigma\Rightarrow\Pi,\Box A\nearrow\epsilon\Rightarrow A}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\swarrow\Sigma\Rightarrow\Pi,\Box A}\ \Box^1_R$$

$$\frac{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta,A\nearrow\Sigma\Rightarrow\Pi,\blacksquare A\quad\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\nearrow\Sigma\Rightarrow\Pi,\blacksquare A\swarrow\epsilon\Rightarrow A}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\nearrow\Sigma\Rightarrow\Pi,\blacksquare A}\ \blacksquare^1_R$$

$$\frac{\mathcal{G}\nearrow\Gamma\Rightarrow\Delta,\Box A\nearrow\epsilon\Rightarrow A}{\mathcal{G}\nearrow\Gamma\Rightarrow\Delta,\Box A}\ \Box^2_R\qquad\frac{\mathcal{G}\swarrow\Gamma\Rightarrow\Delta,\blacksquare A\swarrow\epsilon\Rightarrow A}{\mathcal{G}\swarrow\Gamma\Rightarrow\Delta,\blacksquare A}\ \blacksquare^2_R$$

$$\frac{\mathcal{G}\updownarrow\Gamma,\Box A\Rightarrow\Delta\nearrow\Sigma,A\Rightarrow\Pi}{\mathcal{G}\updownarrow\Gamma,\Box A\Rightarrow\Delta\nearrow\Sigma\Rightarrow\Pi}\ \Box^1_L\qquad\frac{\mathcal{G}\updownarrow\Gamma,\blacksquare A\Rightarrow\Delta\swarrow\Sigma,A\Rightarrow\Pi}{\mathcal{G}\updownarrow\Gamma,\blacksquare A\Rightarrow\Delta\swarrow\Sigma\Rightarrow\Pi}\ \blacksquare^1_L$$

$$\frac{\mathcal{G}\updownarrow\Gamma,A\Rightarrow\Delta}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\swarrow\Sigma,\Box A\Rightarrow\Pi}\ \Box^2_L\qquad\frac{\mathcal{G}\updownarrow\Gamma,A\Rightarrow\Delta}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta\nearrow\Sigma,\blacksquare A\Rightarrow\Pi}\ \blacksquare^2_L$$

$$\frac{}{\mathcal{G}\updownarrow\Gamma,p\Rightarrow p,\Delta}\ (id)\qquad\frac{}{\mathcal{G}\updownarrow\Gamma,\bot\Rightarrow\Delta}\ \bot_L\qquad\frac{\mathcal{G}}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta}\ \mathsf{EW}$$

$$\frac{\mathcal{G}\updownarrow\Gamma,A\Rightarrow\Delta,A\to B,B}{\mathcal{G}\updownarrow\Gamma\Rightarrow\Delta,A\to B}\ \to_R\qquad\frac{\mathcal{G}\updownarrow\Gamma,A\to B,B\Rightarrow\Delta\quad\mathcal{G}\updownarrow\Gamma,A\to B\Rightarrow\Delta,A}{\mathcal{G}\updownarrow\Gamma,A\to B\Rightarrow\Delta}\ \to_L$$

Fig. 1. The system $\mathsf{LNS_{Kt}}$ where $\updownarrow$ stands for either $\nearrow$ or $\swarrow$

We use lower case letters (p,q,r) for atomic formulae, upper case letters (A,B,C) for formulae, capital Greek letters $(\Gamma,\Delta,\Sigma,\Pi)$ for finite multisets of formulae and calligraphic capital letters $(\mathcal{G},\mathcal{H})$ for LNSs, unless otherwise stated. We often write $\mathcal{G}$ for a possibly empty *context*: e.g., $\mathcal{G}\nearrow\Gamma\Rightarrow\Delta$ stands for $\Gamma\Rightarrow\Delta$ if $\mathcal{G}$ is empty, and for $\Sigma\Rightarrow\Pi\swarrow\Omega\Rightarrow\Theta\nearrow\Gamma\Rightarrow\Delta$ if $\mathcal{G}$ is the LNS $\Sigma\Rightarrow\Pi\swarrow\Omega\Rightarrow\Theta$.

The rules of $\mathsf{LNS_{Kt}}$ [8] are in Fig. 1. Each rule has a number of *premisses* above the line and a single *conclusion* below it. The single formula in the conclusion is the *principal* formula and the formulae in the premisses are the *side-formulae*.

Intuitively, each component of a linear nested sequent corresponds to a world of a Kripke model, and the structural connectives $\nearrow$ and $\swarrow$ between components corresponds to the relations R and R^{-1} that connect these worlds. We can then think of a Kripke model forcing a LNS $\mathcal{G}$ if for every connected sequence of worlds corresponding to the structure of $\mathcal{G}$, one of those worlds forces its corresponding sequent component. See [8] for the formal detail.

Every instance of the (id) and $\bot_L$ rules is a *derivation* of depth 1. If (ρ) is an n-ary rule and there are n premiss derivations $\mathcal{D}_1,\cdots,\mathcal{D}_n$, each of depth $d_1,\cdots,d_n$, with respective conclusions $c_1,\cdots,c_n$, and $c_1,\cdots,c_n/c_0$ is an instance of (ρ) then $\mathcal{D}_1,\cdots,\mathcal{D}_n/c_0$ is a derivation of depth $1+max\{d_1,\cdots,d_n\}$. We use $\mathsf{dp}(\mathcal{D})$ for the depth of derivation $\mathcal{D}$.

We generalise "derivations" to allow for "unfinished leaves", by which we simply mean leaf sequents from a given set that are considered separate to any zero-premiss rules in the system. We write $\mathcal{D}\vdash^{\mathsf{prems}}_{\mathsf{rls}}\mathcal{G}$ to mean that $\mathcal{D}$ is a derivation of the LNS $\mathcal{G}$ in the calculus with rule set rls, allowing for unfinished leaves prems. We further write $\mathcal{D}\vdash_{\mathsf{rls}}\mathcal{G}$ when prems is empty and thus $\mathcal{D}$ must be a finished proof with no unfinished leaves. In both cases, we may omit the

$\mathcal{D}$ to mean that there exists a derivation $\mathcal{D}$ such that $\mathcal{D} \vdash_{\mathsf{rls}}^{\mathsf{prems}} \mathcal{G}$ or $\mathcal{D} \vdash_{\mathsf{rls}} \mathcal{G}$, respectively. We simply write $\mathcal{D} \vdash \mathcal{G}$ if $\mathcal{D}$ is a derivation in $\mathsf{LNS}_{\mathsf{Kt}}$ of the linear nested sequent $\mathcal{G}$, and $\vdash \mathcal{G}$ if there is a derivation $\mathcal{D}$ with $\mathcal{D} \vdash \mathcal{G}$.

Example 1. Consider the LNS $\epsilon \Rightarrow r \to \blacksquare\neg\Box\neg r$ where $r \to \blacksquare\neg\Box\neg r$ is the formula $r \to \blacksquare\Diamond r$ with the definition of $\Diamond$ expanded. The following is a non-trivial derivation that demonstrates the use of some of the box rules to move formulae between different components. Note that it uses the provably admissible rules $\neg_L$, $\neg_R$ and weakening.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r, \neg r \Rightarrow r, \blacksquare\neg\Box\neg r}\,(\mathrm{id})}{r, \neg r \Rightarrow \blacksquare\neg\Box\neg r}\,\neg_L}{r \Rightarrow \blacksquare\neg\Box\neg r \swarrow \Box\neg r \Rightarrow \neg\Box\neg r}\,\Box^2_L}{r \Rightarrow \blacksquare\neg\Box\neg r \swarrow \epsilon \Rightarrow \neg\Box\neg r}\,\neg_R}{r \Rightarrow \blacksquare\neg\Box\neg r}\,\blacksquare^2_R}{r \Rightarrow r \to \blacksquare\neg\Box\neg r, \blacksquare\neg\Box\neg r}\,\mathrm{weak}}{\epsilon \Rightarrow r \to \blacksquare\neg\Box\neg r}\,{\to_R}$$

3 Encoding Formulae, Sequents and LNSs

Having seen the pen-and-paper definition of the calculus, we turn to our Coq formalisation. We start with a set of proposition variables, `V : Set`, which corresponds to *Atm*, over which we build our formulae, `PropF V`:

```
Inductive PropF (V : Set) : Type :=
  | Bot  : PropF V
  | Var  : V -> PropF V
  | WBox : PropF V -> PropF V
  | BBox : PropF V -> PropF V
  | Imp  : PropF V -> PropF V -> PropF V.
```

Here we are creating a new type called `PropF`, and so we would encode, for example, the infix formula $\Box(p \to q) \to (\Box p \to \Box q)$ using prefix notation by the term `Imp (WBox (Imp (Var p) (Var q))) (Imp (WBox (Var p)) (WBox (Var q)))`.

Recall our sequents consist of multisets. To model this we chose lists and later proved exchange lemmas that enable us to move formulae around in any order without compromising derivability. See Sect. 6 for further details. Thus the multiset Γ of formulae is encoded as a term with type `list (PropF V)` and sequents, which have the form $\Gamma \Rightarrow \Delta$, have type `seq`:

```
Definition rel (W : Type) : Type := W * W. (* ie, prod W W *)
Definition seq {V : Set} := rel (list (PropF V)).
```

Here, `prod W W`, also written `W * W`, is the Cartesian product $W \times W$, and so an `s` of type `seq` is a pair of lists of formulae such as `pair` Γ Δ, also written (Γ, Δ).

In Coq, `list` comes pre-defined as expected, where `a :: b :: c :: nil` encodes the list $[a, b, c]$. We can append list `l1` to list `l2` by the `++` operator. For a function `f : A -> B`, and a list `l : list A`, the result of `map f l` is got

by applying `f` to each member of `l`. We use these extensively throughout our formalisation, as can be seen in the definitions for `nslclext` and `nslclrule` in Sect. 4.

Coq allows "implicit arguments" where certain arguments for some functions can usually be inferred and are not given. Typical examples are the first arguments of the list operators `++` and `::`, which state the type of the list members. The symbol `@` preceding a function name, as in `@seq V`, indicates that all arguments are given explicitly. In the code above, braces as in {`V : Set`}, as opposed to parentheses as in `(W : Type)`, indicates `V` is to be an implicit argument.

We defined the type `LNS` to encode LNSs as lists of pairs of sequents and directions, where the latter is defined to have two inhabitants corresponding to the $\nearrow$ and $\swarrow$ arrows:

```
Inductive dir : Type :=   | fwd : dir  | bac : dir.
Definition LNS {V : Set} := list ((@seq V) * dir).
```

There is an extra direction in the type for LNS: for n components there should only be $n-1$ directions. We ignore the first direction: so $\Gamma \Rightarrow \Delta \nearrow \Sigma \Rightarrow \Pi$ is encoded by [(Γ, Δ, `fwd`), (Σ, Π, `fwd`)] and [(Γ, Δ, `bac`), (Σ, Π, `fwd`)].

4 Encoding the $\mathsf{LNS_{Kt}}$ Calculus

A rule instance couples a list `ps` of premises with a conclusion `c`. We define a type `rlsT W := list W -> W -> Type` so that `rlsT ps c` is the type of all rule instances with list of premises `ps` and conclusion `c`. Our aim then is to encode a collection of permissible rule schemas – in our case the rules of $\mathsf{LNS_{Kt}}$ – by defining the type `LNSKt_rules` which has type `rlsT (@LNS V)`.

To do so, we encoded sub-collections of rules called `b2rrules`, `b1rrules`, `b2lrules`, `b1lrules`, `EW_rule` and `rs_prop` which correspond to the boxed rules, external weakening and the remaining propositional rules, respectively.

Consider `b2lrules` with `WBox2Ls` corresponding to $\Box^2_L$ and `BBox2Ls` to $\blacksquare^2_L$:

```
Inductive b2lrules (V : Set) : rlsT (@LNS V) :=
| WBox2Ls : forall A d Γ1 Γ2 Σ1 Σ2 Δ Π, b2lrules
  [ [((Γ1 ++ A :: Γ2), Δ, d)] ]
  [((Γ1 ++ Γ2), Δ, d) ; ((Σ1 ++ WBox A :: Σ2), Π, bac)]
| BBox2Ls : forall A d Γ1 Γ2 Σ1 Σ2 Δ Π, b2lrules
  [[((Γ1 ++ A :: Γ2), Δ, d) ]]
  [((Γ1 ++ Γ2), K1, d) ; ((Σ1 ++ BBox A :: Σ2), Π, fwd)].
```

The first `WBox2Ls` says that any rule instance that has one premise (encoded as a singleton list of LNSs) of the form $\Gamma_1, A, \Gamma_2 \Rightarrow \Delta$ and conclusion of the form $\Gamma_1, \Gamma_2 \Rightarrow \Delta \swarrow \Sigma_1, \Box A, \Sigma_2 \Rightarrow \Pi$ is permitted. Thus although the official rule has an antecedent multiset Γ, A, we present it as the list Γ_1, A, Γ_2 instead to align the pen-and-paper presentation and Coq presentation.

Note that we encode only those components containing principal or side formulae i.e. the last two components of the conclusion. So `WBox2Ls` encodes the rule instance below left. Ultimately we want the full context version below right.

$$\frac{\Gamma_1, A, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, \Gamma_2 \Rightarrow \Delta \swarrow \Sigma_1, \Box A, \Sigma_2 \Rightarrow \Pi} \qquad \frac{\mathcal{G} \updownarrow \Gamma_1, A, \Gamma_2 \Rightarrow \Delta}{\mathcal{G} \updownarrow \Gamma_1, \Gamma_2 \Rightarrow \Delta \swarrow \Sigma_1, \Box A, \Sigma_2 \Rightarrow \Pi}\ \Box^2_L$$

To obtain the right one from the left, we define `nslcext` and `nclcrule`, where `nslcext` extends an LNS `ls` with a given context `G` to the left. We use `nclcrule` to extend the collection of rule instances defined by `b2lrules` to allow for the uniform adding of contexts with `nslcext` into premises and conclusions.

```
Definition nslclext W (G ls : list W) := G ++ ls.

Inductive nslclrule W (sr : rlsT (list W)) : rlsT (list W) :=
| NSlclctxt : forall ps c G, sr ps c ->
  nslclrule sr (map (nslclext G) ps) (nslclext G c).
```

If `r` is of type `nslclrule W sr` then it must be obtained by adding a (possibly empty) context to all the premises and conclusion of a rule instance of `sr`.

So `nslclrule (@b2lrules V) ps c` captures that the premise list `ps` and conclusion `c` form an instance of an extended version of a rule from `b2lrules` via uniform context addition, giving the full version of $\Box_L^2$ on the right above.

We can then define the full $\mathsf{LNS}_{\mathsf{Kt}}$ rule set with `b2rrules` and the other corresponding subcollections by the following `LNSKt_rules` definition:

```
Inductive LNSKt_rules {V : Set} : rlsT (@LNS V) :=
| b2r  : forall ps c, nslclrule (@b2rrules V) ps c ->
  LNSKt_rules ps c
| b1r  : forall ps c, nslclrule (@b1rrules V) ps c ->
  LNSKt_rules ps c
| b2l  : forall ps c, nslclrule (@b2lrules V) ps c ->
  LNSKt_rules ps c
| b1l  : forall ps c, nslclrule (@b1lrules V) ps c ->
  LNSKt_rules ps c
| nEW  : forall ps c, nslclrule (@EW_rule V) ps c ->
  LNSKt_rules ps c
| prop : forall ps c, nslcrule (seqrule (@rs_prop V)) ps c ->
  LNSKt_rules ps c.
```

The `b2l` case reads: if the premise list `ps` and conclusion `c` form an extended instance of `b2lrules`, then `ps/c` is also a rule instance of `LNSKt_rules`. The other rules work in much the same way. The exceptions are the `prop` rules as `rs_prop` captures an even more refined rule skeleton for $\rightarrow_L$, $\rightarrow_R$, $\bot_L$ and (id) and requires the addition of a sequent level context (Γs and Δs) via `seqrule` [6]. We leave it to the reader to see the original code for details of the remaining rules.

5 Encoding Derivability

Our notion of derivability follows Dawson and Goré's formalisation in Isabelle [6]:

```
Inductive derrec X (rules : list X -> X -> Type) (prems : X -> Type) :
X -> Type :=
| dpI  : forall concl, prems concl -> derrec rules prems concl
| derI : forall ps concl, rules ps concl ->
         dersrec rules prems ps -> derrec rules prems concl
with dersrec X (rules : list X -> X -> Type) (prems : X -> Type) :
list X -> Type :=
| dlNil  : dersrec rules prems []
| dlCons : forall seq seqs, derrec rules prems seq ->
           dersrec rules prems seqs ->
           dersrec rules prems (seq :: seqs).
```

The `X` is the type of objects about which we are reasoning: formulae in natural deduction calculi, sequents in sequent calculi, etc. In our case, we will instantiate `X` with the type `LNS` of LNSs. Then `D : derrec X rules prems concl` encodes $\mathtt{D} \vdash^{\mathtt{prems}}_{\mathtt{rules}} \mathtt{concl}$. But, by a complication of Coq, `X` is an "implicit argument" and must be omitted in the line above. Think of `prems` as a characteristic function for set membership. Likewise `rules`, where the set is a set of pairs (each pair being a list of premises and a conclusion).

A conclusion `concl` is derivable from a set `prems` of premises if

`dpI`: `concl` is a member of the set `prems`, or
`derI`: there is a rule inferring `concl` from a list `ps`, and each `p` in `ps` is derivable

The notion `derrec` is defined mutually with `dersrec` which asserts that a list of conclusions is derivable instead of just one as in `derrec`. Thus `dersrec X rules prems concls`, using constructors `dlNil` and `dlCons`, holds if all members of `concls` are derivable via `derrec`.

In contrast to encodings that define both derivability and calculus rules in the one definition (for example [2,21]), `derrec` gives a modular framework where `X`, `rules` and `prems` can be arbitrary. Thus `derrec` can be used for a variety of calculi where we can prove lemmas which are generic to multiple calculi satisfing certain conditions, rather than having to prove the same lemmas over and over for different calculi.

Interlude: Doing this in Coq Versus Isabelle/HOL. We can contrast how this works out in Coq compared with our previous work in Isabelle/HOL. In Coq, as `derrec ... concl` is a `Type`, it represents the derivation tree showing that `concl` is derivable. This means we can define the height or size of a derivation tree, as needed to do proofs by induction on height or size of a derivation. (We note that doing this specifically requires using `Type`, not `Prop`, for `derrec`, `dersrec`, etc.)

By contrast, in Isabelle we had to define a separate data structure to describe a derivation tree (essentially a rose tree of sequents), specify the condition of its validity (that each node is a rule of the system), and prove that such a tree exists iff the endsequent `concl` satisfies `derrec ... concl`.

However the derivation tree that Coq gives us for free has problems not shared by the Isabelle/HOL derivation tree: namely, the derivation trees next up from the endsequent (ie the trees deriving the premises of the bottom rule of the tree) do not form a list, because they are not of the same type, as their conclusions are all different. Navigating around such difficulties was not easy.

In our cut-elimination context we were mostly working with Coq proofs that required all leaves to be obtained via the (id) or $\bot_L$ rules, and so we regularly instantiated `prems : X -> Type` with the empty characteristic function `(fun _ => False)`. The "wrappers" `pf` (proof) and `pfs` (proofs) encode such derivations:

```
Definition pf {X : Type} rules concl :=
  @derrec X rules (fun _ => False) concl.

Definition pfs {X : Type} rules concls :=
  @dersrec X rules (fun _ => False) concls.
```

Then, `pf_LNSKt` and `pfs_LNSKt` are the "proofs" with `rules` being `LNSKt_rules`:

```
Definition pf_LNSKt {V : Set} ns :=
  derrec (@LNSKt_rules V) (fun _ => False) ns.

Definition pfs_LNSKt {V : Set} lns :=
  dersrec (@LNSKt_rules V) (fun _ => False) lns.
```

6 Proof Theoretic Properties of LNSKt

We proved `LNSKt_exchL` and `LNSKt_exchR` as the admissibility of left and right internal exchange of formulae but show only left exchange for brevity:

Lemma 1 (Left internal exchange of $\mathsf{LNS_{Kt}}$). *If* $\vdash \mathcal{G} \updownarrow_1 \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \Rightarrow \Delta \updownarrow_2 \mathcal{K}$ *then* $\vdash \mathcal{G} \updownarrow_1 \Gamma_1, \Gamma_3, \Gamma_2, \Gamma_4 \Rightarrow \Delta \updownarrow_2 \mathcal{K}$.

```
Definition can_gen_swapL {V : Set} (rules : rlsT (@LNS V)) ns :=
  forall G K s d Γ1 Γ2 Γ3 Γ4 Δ,
  ns = G ++ (s, d) :: K ->
  s = pair (Γ1 ++ Γ2 ++ Γ3 ++ Γ4) Δ ->
  pf rules (G ++ (pair (Γ1 ++ Γ3 ++ Γ2 ++ Γ4) Δ, d) :: K).

Lemma LNSKt_exchL: forall (V : Set) ns (D : @pf_LNSKt V ns),
  can_gen_swapL (@LNSKt_rules V) ns.
```

Thus if LNS `ns` is derivable, then so is any LNS which permutes two adjacent sublists Γ`2` and Γ`3` on the left – all within $\mathsf{LNS_{Kt}}$ with no unfinished leaves.

On paper, Lemma 1 is immediate as the Γs and Δs are multisets so that Γ_2, Γ_3 is identical to Γ_3, Γ_2. In our Coq formalisation however, we chose to encode the Γs and Δs instead by lists and prove left and right internal exchange so `LNSKt_exchL` was not immediate.

Our proof is a standard induction on the structure of `D` using the inductive hypothesis automatically generated by Coq that the required result holds for the premises of the final (bottom) rule application. This is a weaker principle than induction on the depth of `D` with an inductive hypothesis that the required result holds for the conclusions of any derivation of lesser depth.

The remaining proof theoretic properties, internal weakening and contraction, are stated in the original paper as Lemma 13. As with left-exchange, we used a"`can_gen`" conclusion and a standard induction. See the code for details.

We are now ready to tackle cut-elimination.

7 Cut-Elimination via Cut-Admissibility

Goré and Lellmann [8] proved cut-admissibility in their cut-free calculus and we follow suit. Using cut-admissibility, we additionally prove cut-elimination. To state cut-admissibility for LNSs, we must merge two LNSs in the following way:

Definition 2. *The* merge *of two linear nested sequents is defined via the following, where we assume $\mathcal{G}$ and $\mathcal{H}$ to be non-empty:*

$$\begin{aligned}
(\Gamma \Rightarrow \Delta) \oplus (\Sigma \Rightarrow \Pi) &:= \Gamma, \Sigma \Rightarrow \Delta, \Pi \\
(\Gamma \Rightarrow \Delta) \oplus (\Sigma \Rightarrow \Pi \updownarrow \mathcal{H}) &:= \Gamma, \Sigma \Rightarrow \Delta, \Pi \updownarrow \mathcal{H} \\
(\Gamma \Rightarrow \Delta \updownarrow \mathcal{H}) \oplus (\Sigma \Rightarrow \Pi) &:= \Gamma, \Sigma \Rightarrow \Delta, \Pi \updownarrow \mathcal{H} \\
(\Gamma \Rightarrow \Delta \nearrow \mathcal{G}) \oplus (\Sigma \Rightarrow \Pi \nearrow \mathcal{H}) &:= \Gamma, \Sigma \Rightarrow \Delta, \Pi \nearrow (\mathcal{G} \oplus \mathcal{H}) \\
(\Gamma \Rightarrow \Delta \swarrow \mathcal{G}) \oplus (\Sigma \Rightarrow \Pi \swarrow \mathcal{H}) &:= \Gamma, \Sigma \Rightarrow \Delta, \Pi \swarrow (\mathcal{G} \oplus \mathcal{H}) \,.
\end{aligned}$$

Our Coq encoding of merge is as follows:

```
Inductive merge {V : Set} :
(@LNS V) -> (@LNS V) -> (@LNS V) -> Type :=
| merge_nilL ns1 ns2 ns3 : ns1 = [] -> ns3 = ns2 -> merge ns1 ns2 ns3
| merge_nilR ns1 ns2 ns3 : ns2 = [] -> ns3 = ns1 -> merge ns1 ns2 ns3
| merge_step Γ Δ Σ Π d ns1 ns2 ns3 ns4 ns5 ns6 s1 s2 s3 :
  s1 = (Γ,Δ,d) -> s2 = (Σ,Π,d) -> s3 = (Γ ++ Σ, Δ ++ Π,d) ->
  merge ns1 ns2 ns3 -> ns4 = s1 :: ns1 ->
  ns5 = s2 :: ns2 -> ns6 = s3 :: ns3 ->
  merge ns4 ns5 ns6.
```

Thus `merge ns1 ns2 ns3` encodes that `ns3` is the merge of `ns1` and `ns2`. Our Coq definition allows either the left or right LNS to be empty.

Note also that both pen-and-paper and Coq definitions are only well-defined for LNSs that are in some sense *structurally equivalent*, where the arrows of the two LNSs correspond. The original paper allowed LNSs of possibly differing lengths to be structurally equivalent. We instead used a strong version for only equal length LNSs as this was all that was needed. Doing so simplified some of the proofs. Alternatively we could define `merge` to hold only for structurally equivalent sequents. But where convenient, we conform to the original paper.

Definition 3. *Two LNSs $S_1 \updownarrow_1^S \ldots \updownarrow_{n-1}^S S_n$ and $T_1 \updownarrow_1^T \ldots \updownarrow_{n-1}^T T_n$ are* structurally equivalent *if we have $\updownarrow_i^S = \updownarrow_i^T$ for every i.*

```
Inductive struct_equiv_str {V : Set} : (@LNS V) -> (@LNS V) -> Type :=
| se_nil2 : struct_equiv_str [] []
| se_step2 Γ1 Δ1 d Γ2 Δ2 ns1 ns2 ns3 ns4 :
ns3 = ((Γ1, Δ1, d) :: ns1) -> ns4 = ((Γ2, Δ2, d) :: ns2) ->
struct_equiv_str ns1 ns2 -> struct_equiv_str ns3 ns4.
```

Our cut-admissibility theorem is called cut-elimination in the original paper.

7.1 Cut-Admissibility

The cut rule, where the premiss and conclusion LNSs are structurally equivalent:

$$\frac{\mathcal{G} \updownarrow \Gamma \Rightarrow \Delta, A \qquad \mathcal{H} \updownarrow A, \Sigma \Rightarrow \Pi}{\mathcal{G} \oplus \mathcal{H} \updownarrow \Gamma, \Sigma \Rightarrow \Delta, \Pi}\ \mathsf{Cut}$$

Theorem 1 (Cut-admissibility). *For structurally equivalent LNSs $\mathcal{G}$ and $\mathcal{H}$, if $\vdash \mathcal{G} \updownarrow \Gamma \Rightarrow \Delta, A$ and $\vdash \mathcal{H} \updownarrow A, \Sigma \Rightarrow \Pi$, then also $\vdash \mathcal{G} \oplus \mathcal{H} \updownarrow \Gamma, \Sigma \Rightarrow \Delta, \Pi$.*

```
Definition can_gen_cut {V : Set} (rules : rlsT (@LNS V)) ns1 ns2 :=
  forall G1 G2 G3 s1 s2 d Γ Δ1 Δ2 Σ1 Σ2 Π A,
  ns1 = G1 ++ [(s1, d)] -> s1 = pair Γ (Δ1++[A]++Δ2) ->
  ns2 = G2 ++ [(s2, d)] -> s2 = pair (Σ1++[A]++Σ2) Π ->
  merge G1 G2 G3 -> struct_equiv_str G1 G2 ->
  pf rules (G3 ++ [(Γ++Σ1++Σ2, Δ1++Δ2++Π, d)]).

Theorem LNSKt_cut_admissibility : forall (V : Set) ns1 ns2
  (D1 : pf_LNSKt ns1) (D2 : pf_LNSKt ns2),
  can_gen_cut (@LNSKt_rules V) ns1 ns2.
```

So `LNSKt_cut_admissibility` states that if two structurally-equivalent LNSs are (cut-free) derivable from `LNSKt_rules` then so is the structurally-equivalent LNS obtained by applying the cut rule to their conclusions (i.e. no cut in the rules).

Our Coq code follows the original paper where cut-admissibility is a corollary of the huge `Lemma_Sixteen`, thus transfering all heavy lifting to this mega lemma.

7.2 The Main Lemma: "Lemma Sixteen"

The so-called Lemma Sixteen is defined as follows.

Lemma 2. *The following statements hold for every n, m where we always assume that $\mathcal{G}$ and $\mathcal{H}$ are structurally equivalent:*

($\mathsf{SR}_\Box(n,m)$) *Suppose that all of the following hold:*
- $\mathcal{D}_1 \vdash \mathcal{G} \uparrow_1 \Gamma \Rightarrow \Delta, \Box A$ *with* $\Box A$ *principal in the last rule in* $\mathcal{D}_1$
- $\mathcal{D}_2 \vdash \mathcal{H} \uparrow_1 \Box A, \Sigma \Rightarrow \Pi \uparrow_2 \mathcal{I}$
- *there is a derivation of* $\mathcal{G} \oplus \mathcal{H} \uparrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \nearrow \epsilon \Rightarrow A$
- $\mathsf{dp}(\mathcal{D}_1) + \mathsf{dp}(\mathcal{D}_2) \leq m$
- $|\Box A| \leq n$.

Then there is a derivation of $\mathcal{G} \oplus \mathcal{H} \uparrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \uparrow_2 \mathcal{I}$.

($\mathsf{SR}_\blacksquare(n,m)$) *Suppose that all of the following hold:*
- $\mathcal{D}_1 \vdash \mathcal{G} \uparrow_1 \Gamma \Rightarrow \Delta, \blacksquare A$ *with* $\blacksquare A$ *principal in the last rule in* $\mathcal{D}_1$
- $\mathcal{D}_2 \vdash \mathcal{H} \uparrow_1 \blacksquare A, \Sigma \Rightarrow \Pi \uparrow_2 \mathcal{I}$
- $\mathsf{dp}(\mathcal{D}_1) + \mathsf{dp}(\mathcal{D}_2) \leq m$
- *there is a derivation of* $\mathcal{G} \oplus \mathcal{H} \uparrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \swarrow \epsilon \Rightarrow A$
- $|\blacksquare A| \leq n$.

Then there is a derivation of $\mathcal{G} \oplus \mathcal{H} \uparrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \uparrow_2 \mathcal{I}$.

($\mathsf{SR}_p(n,m)$) *Suppose that all of the following hold:*
- $\mathcal{D}_1 \vdash \mathcal{G} \uparrow_1 \Gamma \Rightarrow \Delta, A$ *with* A *principal in the last applied rule in* $\mathcal{D}_1$
- $\mathcal{D}_2 \vdash \mathcal{H} \uparrow_1 A, \Sigma \Rightarrow \Pi \uparrow_2 \mathcal{I}$
- $\mathsf{dp}(\mathcal{D}_1) + \mathsf{dp}(\mathcal{D}_2) \leq m$
- $|A| \leq n$
- A *not of the form* $\Box B$ *or* $\blacksquare B$.

Then there is a derivation of $\mathcal{G} \oplus \mathcal{H} \uparrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \uparrow_2 \mathcal{I}$.

($\mathsf{SL}(n,m)$) *Suppose that all of the following hold:*
- $\mathcal{D}_1 \vdash \mathcal{G} \uparrow_1 \Gamma \Rightarrow \Delta, A \uparrow_2 \mathcal{I}$
- $\mathcal{D}_2 \vdash \mathcal{H} \uparrow_1 A, \Sigma \Rightarrow \Pi$

– $\mathsf{dp}(\mathcal{D}_1) + \mathsf{dp}(\mathcal{D}_2) \leq m$
– $|A| \leq n$.

Then there is a derivation of $\mathcal{G} \oplus \mathcal{H} \updownarrow_1 \Gamma, \Sigma \Rightarrow \Delta, \Pi \updownarrow_2 \mathcal{I}$.

We show only the encoding corresponding to one of the parts, $\mathsf{SR}_{\Box}(n, m)$.

```
Definition SR_wb_pre (n m : nat) := forall {V : Set}
  G Γ Δ1 Δ2 H Σ1 Σ2 Π I GH (A : PropF V) d
  (D1 : pf_LNSKt (G ++ [(Γ, Δ1 ++ [WBox A] ++ Δ2,d)]))
  (D2 : pf_LNSKt (H ++ [(Σ1 ++ [WBox A] ++ Σ2, Π, d)] ++ I))
  (D3 : pf_LNSKt
          (GH ++ [(Γ ++ Σ1 ++ Σ2, Δ1 ++ Δ2 ++ Π, d)] ++
            [([],[A],fwd)])),
  principal_WBR D1 (WBox A) Γ Δ1 Δ2 ->
  ((dp D1) + (dp D2))%nat <= m ->
  struct_equiv_str G H ->
  merge G H GH ->
  fsize (WBox A) <= n ->
  pf_LNSKt (GH ++ [(Γ ++ Σ1 ++ Σ2, Δ1 ++ Δ2 ++ Π, d)] ++ I).

Definition SR_wb (nm : nat * nat) :=
  let (n,m) := nm in SR_wb_pre n m.

Lemma Lemma_Sixteen : forall (nm : nat * nat),
  SR_wb nm * SR_bb nm * SR_p nm * SL nm.
```

We had to state that $\Box A$ is principal in the last rule in $\mathcal{D}_1$. That is, although there may be other occurrences of $\Box A$ in $\mathcal{G} \updownarrow_1 \Gamma \Rightarrow \Delta, \Box A$, it is that particular displayed occurrence that is principal. To capture this, we defined `principal_WBR` which is specifically designed for white box formulae that occur on the right side of the sequent in the last component (hence `WBR` standing for White Box Right). Moreover, the statement `principal_WBR D1 (WBox A)` $\Gamma\, \Delta 1\, \Delta 2$ carries all required information of where the principal `WBox A` sits, in particular that it sits between $\Delta 1$ and $\Delta 2$. This is a requirement specific to our implementation based on lists. We have analogous definitions for the other cases: `principal_BBR` and `principal_not_box_R`. The full code contains these definitions and others we have omitted here, including depth of derivation `dp` and formula size `fsize`.

The Coq `Lemma_Sixteen` uses `*` as a `Type`-level conjunction as in the earlier example to enable extraction. All four parts of Lemma Sixteen are proved simultaneously by induction on the pair (n, m) in the lexicographic ordering, as in the original proof. Please refer to the code for our definitions of this lexicographic ordering `lt_lex_nat` and our induction principle `wf_lt_lex_nat_induction`.

The need to prove all four components of Lemma Sixteen simultaneously is of course because the different parts depend on induction hypotheses of the other components. The proof works through a lot of different cases, often multiple layers of cases going at once. Given that there are already four sublemmas to prove as well as the copious number of cases, the pen-and-paper proof is large and the Coq proof is, understandably, even larger.

Coq is useful to check the subtle details for each case, and it is unsurprising that our work highlighted multiple mistakes in the original proof. These ranged from incorrect arrow directions, incorrect rule applications, to omissions of conditions such as structural equivalence and same length of LNSs. Fortunately all were easily resolved. We have confirmed these with the original authors.

For example, in $\mathsf{SR}_{\square}(n,m)$ ($\mathsf{SR}_{\blacksquare}(n,m)$), where we have $\updownarrow_1$ the original paper had $\nearrow$ ($\swarrow$), but there were indeed cases which require our more general version.

Cut-admissibility follows from ($\mathsf{SL}(n,m)$), and leads easily to cut-elimination.

7.3 Cut-Elimination

We encoded the cut rule in Coq as follows where we don't encode the skeleton of cut and then fill in contexts using `nslclrule` because the premises do not share linear nested sequent level contexts *i.e.* $\mathcal{G}$ and $\mathcal{H}$ may differ:

```
Inductive Cut_rule {V : Set} : rlsT (@LNS V) :=
| Cut : forall G H GH s1 s2 s3 ns1 ns2 ns3 d Γ Δ1 Δ2 Σ1 Σ2 Π A,
  ns1 =  G ++ [(s1, d)] -> s1 = (Γ, (Δ1++[A]++Δ2)) ->
  ns2 =  H ++ [(s2, d)] -> s2 = ((Σ1++[A]++Σ2), Π) ->
  ns3 = GH ++ [(s3, d)] -> s3 = ((Γ++Σ1++Σ2), (Δ1++Δ2++Π)) ->
  merge G H GH -> struct_equiv_str G H ->
  Cut_rule [ns1 ; ns2] ns3.
```

We then defined the calculus $\mathsf{LNS}_{\mathsf{Kt}+\mathsf{Cut}}$ as $\mathsf{LNS}_{\mathsf{Kt}}$ plus Cut in Coq:

```
Inductive LNSKt_cut_rules {V : Set} : rlsT (@LNS V) :=
| LNSKt_rules_woc :
  forall ps c, LNSKt_rules ps c   -> LNSKt_cut_rules ps c
| LNSKt_rules_wc  :
  forall ps c, (@Cut_rule V) ps c -> LNSKt_cut_rules ps c.
```

The first constructor (`_woc` for "without cut") includes all `LNSKt_rules` and the second constructor (`_wc` for "with cut") adds the cut rule. We then specialised `LNSKt_cut_rules` to have no unfinished leaves as for `LNSKt_rules`:

```
Definition pf_LNSKt_cut {V : Set} ns :=
  derrec (@LNSKt_cut_rules V) (fun _ => False) ns.
```

The cut-elimination theorem allows us to eliminate Cut applications:

Theorem 2. *For every LNS* $\mathcal{G}$*, if* $\vdash_{\mathsf{LNS}_{\mathsf{Kt}+\mathsf{Cut}}} \mathcal{G}$ *then* $\vdash_{\mathsf{LNS}_{\mathsf{Kt}}} \mathcal{G}$*.*

```
Theorem LNSKt_cut_elimination :
  forall {V:Set} (ns:@LNS V), pf_LNSKt_cut ns -> pf_LNSKt ns.
```

The original paper stopped at cut-admissibility (though they called it cut-elimination), so we produced our own proof in the standard way by induction on the depth of the derivation with cut. As usual, we start with a derivation with cuts, eliminate the cut application with smallest depth using cut-admissibility, and repeat the procedure in the resulting (transformed) derivation!

8 Extraction

The form of the cut-elimination theorem in Coq enables us to utilise Coq's extraction facility to synthesise a Haskell program that computes cut-free derivations from those with cut. Specifically, we can use Coq to distill the algorithmic content of our cut-elimination theorem in order to automatically produce a Haskell function that computes cut-free derivation from those without cut.

The file cut_elimination_theorem.v imports the necessary libraries and sets the language to Haskell, after which we extract into separate Haskell modules:

```
Require Import cut.
Require Import Extraction.
Extraction Language Haskell.

Separate Extraction LNSKt_cut_elimination.
```

This process automatically produces 47 Haskell modules, with the final cut-elimination function, `coq_LNSKt_cut_elimination`, specified in the Cut.hs module.

To use `coq_LNSKt_cut_elimination`, we have to specify how the required objects should be printed, and so we hand-coded a simple printing module that can be easily checked, called Main_thm.hs, which imports Cut.hs. Once Main_thm.hs is loaded, one can then use the cut-elimination function by calling `coq_LNSKt_cut_elimination` with the appropriate input.

Given that `coq_LNSKt_cut_elimination` requires inputs that are fairly large and can be difficult to write, our preferred method is to encode the desired examples in Coq before extraction, and then use extraction on both the cut-elimination theorem as well as that example derivation. That way, we benefit from Coq's type checker on the example derivation as well as the cut-elimination function. Let us illustrate with an example.

Consider the following derivation that uses the cut rule:

$$\dfrac{\dfrac{\dfrac{}{\Box p \Rightarrow \epsilon \nearrow p \Rightarrow p}\,(\mathrm{id})}{\Box p \Rightarrow \epsilon \nearrow \epsilon \Rightarrow p}\,\Box^1_L \qquad \dfrac{\dfrac{}{\epsilon \Rightarrow \epsilon \nearrow p, q \Rightarrow p, q \rightarrow p}\,(\mathrm{id})}{\epsilon \Rightarrow \epsilon \nearrow p \Rightarrow q \rightarrow p}\,{\rightarrow_R}}{\Box p \Rightarrow \epsilon \nearrow \epsilon \Rightarrow q \rightarrow p}\,\mathrm{Cut}$$

In file cut_extraction_example_pre.v, we hand-coded this derivation in about 100 lines. Each rule instance requires an easy Coq proof to identify the rule, its principal and side-formulae and their locations in the premises and conclusion. For example, lemma `pf3_000` tells us *how* `concl3_000` follows from no premisses, where `concl3_000` corresponds to the conclusion of the left (*id*) instance above:

```
Definition concl3_000 :=
[ ([WBox (Var 0)], [], fwd) ; ([Var 0], [Var 0], fwd) ].

Lemma pf3_000 : LNSKt_cut_rules [] concl3_000.
```

This proof term is required by the cut-elimination function because it makes decisions depending on the form of `pf3_000`: thus we cannot elide it during extraction. The need for the user to specify these proof terms while specifying the full derivation is why we prefer to encode them on the Coq side. So Coq checks their type and the extraction mechanism converts everything to Haskell code.

Then we build up the final derivation, `example3`, by putting together these proof terms, premisses and conclusions. See the full code for details of how this is done. We then define `cut_example3` to perform cut-elimination on `example3`:

```
Definition cut_example3 := LNSKt_cut_elimination example3.
```

Extracting using cut_elimination_theorem.v extracts the cut-elimination function only. Instead, we ask users to compile cut_elimination_example.v, thereby extracting both the cut-elimination function and the example derivation `example3`:

```
Separate Extraction cut_example3.
```

This produces a Haskell module containing the example derivation code: extracting `example3` will not produce Haskell code for the cut-elimination procedure.

Beside the Main_thm.hs printing file, we have written Main_example.hs which is identical except for an import statement that gives access to the example derivation. Once this is loaded, we can call `cut_example3` (or the longer version `coq_LNSKt_cut_elimination example3`) which outputs the cut-free derivation:

```
derI
  (derI
    (derI (dlNil)
          ([[[[[.] p 0], []), fwd) ::
            (([p 0 :: p 1], [p 1 --> p 0 :: p 0]), fwd)])
          (Id))
    ([[[[[.] p 0], []), fwd) :: (([p 0], [p 1 --> p 0]), fwd)])
    (ImpR))
  ([[[[[.] p 0], []), fwd) :: (([], [p 1 --> p 0]), fwd)])
  (WBox1Ls)
```

Adding line breaks and indentation, the code above is the cut-free derivation:

$$\dfrac{\dfrac{\dfrac{}{\Box p \Rightarrow \epsilon \nearrow p, q \Rightarrow q \rightarrow p, p}\,(\mathrm{id})}{\Box p \Rightarrow \epsilon \nearrow p \Rightarrow q \rightarrow p}\,{\rightarrow_R}}{\Box p \Rightarrow \epsilon \nearrow \epsilon \Rightarrow q \rightarrow p}\,\Box^1_L$$

Each `derI` in the above code is a rule application that takes in three arguments: the subderivation of the premise, the conclusion and the name of the rule.

Clearly, the linear representation is not easy to read and a tree style representation would be better. See Sect. 10 for more discussion on this. Note that our printing instructions for the output derivation in the Main files exclude the printing of the proof terms like `pf3_000`. We did this because 1) logicians read this information off naturally from concrete derivations without it being explicitly stated; and 2) it gives a cleaner, easier to read derivation.

By consulting README.txt, readers can input their own derivations, extract, and behold the verified cut-free output derivations from `coq_LNSKt_cut_elimination`.

9 Related Work

Chaudhuri et al. [2] cover the related work well, so we concentrate only on work which formalises meta-theory, as opposed to formalised proof-search.

The work of Pfenning [16], Graham-Lengrand [9], Simmons [17] and Urban [20] all represent sequents as formulae of the meta-logic where, for example, a sequent $A, B \Rightarrow \varphi$ becomes the meta-logical formula `hyp A -> hyp A -> conc phi` [16]. Since the meta-logic is intuitionistic, the sequent calculi inherit exchange, weakening and contraction.

Arbitrary contexts follow by encoding rule skeletons using `->` to encode the horizontal line separating premises and conclusions. These methods cannot handle calculi that lack some combination of weakening, contraction and exchange, nor do they include extraction.

At the next level are encodings which build sequents out of multisets. Dawson and Goré [6] prove mix-elimination for the provability logic GL using Isabelle/HOL. Xavier et al. [21] prove cut-elimination and other meta-theoretic properties of (commutative) linear logic in Coq by extending the standard library for multisets with additional theorems and tactics. Multisets preclude non-commutativity. There is no extraction in either.

At the next level is work where sequents are built from lists, but with extra machinery added to regain commutativity. Tews [18] uses setoids, while Chaudhuri et al. [2] and Larchey-Wendling [11,12] build-in "permutability" explicitly. All these could be extended to handle non-commutative logics, but none do. Only Larchey-Wendling [11] allows extraction, though this has not been published.

Miller and Pimentel [13] explored embedding various object logics into linear logic, and gave "cut-coherence" conditions for the cut-admissibility of linear logic to carry over to an object logic. Olarte et al. [15] extended this work to allow object logics with modalities using a LNS presentation of SLL (a linear logic with subexponentials). A Coq encoding of this work would require us to first encode the syntax of (subexponential) linear logic, and then encode our object logic (sequents) as formulae of linear logic. Encoding into linear logic cannot handle non-commutative substructural logics but does allow us to omit weakening and contraction, which can then be regained via (sub)exponentials.

Our work has numerous advantages: (1) our notion of derivability is parametric on a set of objects `X` which could be formulae, sequents, or other structures; (2) using lists allows us to handle genuinely substructural logics in which some combination of weakening, contraction and exchange (commutativity) are missing, with Dawson and Goré's previous work [5] allowing us to even capture non-associativity if required; and (3) our use of `Type`, rather than `Prop`, in Coq allows us to extract a formally verified computer program to perform cut-elimination.

10 Conclusion and Future Work

We have transported and extended from Isabelle to Coq Dawson and Goré's [6] encodings of general notion of derivability which is usable for many different kinds of proof systems. We applied this to the linear nested sequent calculus $\mathsf{LNS}_{\mathsf{Kt}}$ that was given by Goré and Lellmann [8] for tense logic and formalised the calculus along with all structural proof theoretic properties up to and including their proof of cut-admissibility (called cut-elimination in the original paper).

We uncovered multiple small mistakes but none were major and all were easily amended. The original authors accepted the corrections.

We proved cut-elimination from cut-admissibility and extracted a formally verified Haskell program that computes cut-free derivations from those with cut. We hand-coded Haskell `Main` files to provide requisite printing instructions in order to display the outputs.

Our Coq encodings are modular and allow us to prove meta-theoretic lemmas for arbitrary `rules` that satisfy certain conditions, which can then be applied to specific calculi. For example, in the proof of left internal exchange, the case

where the last rule applied was a rule in `rs_prop` was proved not restricted to just `LNSKt_rules` but more generally for any `rules` provided that the last rule skeleton applied satisfied the `rules_L_oeT` condition, those for which every conclusion has at most one formula on the left. Thus our work has the potential to lead to a Coq library for proof theory that is applicable to a broad range of logics and calculi with results proved in the aforementioned generic way. Most formalisations that we have encountered do not enjoy such modularity (e.g. [2,21]).

This modularity of our encodings is in part due to our capturing not just derivability but derivations as first class citizens with corresponding proof terms. The deep embedding of derivations also enables extraction of the cut-elimination procedure into Haskell, and as such an alternative approach using a shallow embedding (e.g. [16]) would not suffice.

Recall that we encoded multisets in our context with lists and then proved exchange. First, our framework can then handle noncommutative logics and, secondly, gives us access to the libraries of basic reasoning about lists where other multiset libraries seemed to be lacking. The trade-off of this general framework is that in our specific $\mathsf{LNS}_{\mathsf{Kt}}$ context where exchange is admitted we faced needless difficulties relating to where a particular formula is located in a sequent. It is worth exploring a formalisation where `list` is replaced in `Definition seq := rel (list (PropF V))` to an encoding of multisets (e.g. [21]), or which utilises the Permutation library (e.g. [12]). In that case, we would expect to see simpler proofs and less complex and more efficient tactics. This may also translate to a more efficient extracted cut-elimination function which would not need to perform rearrangements on lists of formulae.

Related, we haven't yet performed any tests on efficiency of the extracted function nor attempted to make significant improvements in this area. While we consider this interesting, it is also part of a bigger picture of comparing efficiencies of verified and unverified implementations, which contributes to answering how realistic it is to prefer the former kind over the latter. We are still in early stages of program synthesis of formalised proof-theory and consider this an avenue worthy of further investigation.

Another aspect in this bigger picture of usability is the readability of the extracted function. The printing instructions in the Haskell Main files employ a linear representation to type-check and display the output, which is difficult to read. Alternatively, we could write tree-style printing instructions, import pre-existing Haskell libraries, use a format such as the LaTeX package bussproofs, or even develop a nice graphical user interface. The further down the spectrum of readability you move, the more you compromise on trustworthiness. In its current form, our work adopts a conservative approach on the trustworthy end of the spectrum but we do acknowledge there is scope here to improve usability.

We believe that our work has the potential to lead to a Coq library for deeply embedded and extractable proof theory for the huge number of truly substructural logics in the literature where some combination of weakening, contraction and exchange are not admissible.

References

1. Belnap, N.: Display logic. J. Philos. Log. **11**, 375–417 (1982)
2. Chaudhuri, K., Lima, L., Reis, G.: Formalized meta-theory of sequent calculi for linear logics. TCS **781**, 24–38 (2019)
3. Dawson, J.E., Brotherston, J., Goré, R.: Machine-checked interpolation theorems for substructural logics using display calculi. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 452–468. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_31
4. Dawson, J.E., Clouston, R., Goré, R., Tiu, A.: From display calculi to deep nested sequent calculi: formalised for full intuitionistic linear logic. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) TCS 2014. LNCS, vol. 8705, pp. 250–264. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_20
5. Dawson, J.E., Goré, R.: Formalised cut admissibility for display logic. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 131–147. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45685-6_10
6. Dawson, J.E., Goré, R.: Generic methods for formalising sequent calculi applied to provability logic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 263–277. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_19
7. Goré, R.: Tableau methods for modal and temporal logics. In: D'Agostino, M., Gabbay, D., Haehnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, Kluwer, pp. 297–396 (1999)
8. Goré, R., Lellmann, B.: Syntactic cut-elimination and backward proof-search for tense logic via linear nested sequents. In: Cerrito, S., Popescu, A. (eds.) TABLEAUX 2019. LNCS (LNAI), vol. 11714, pp. 185–202. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29026-9_11
9. Graham-Lengrand, S.: Polarities & focussing: a journey from realisability to automated reasoning, Habilitation Thesis, Université Paris-Sud (2014)
10. Kashima, R.: Cut-free sequent calculi for some tense logics. Stud. Log. **53**(1), 119–136 (1994)
11. Larchey-Wendling, D.: Semantic cut elimination. https://github.com/.DmxLarchey/Coq-Phase-Semantics/blob/master/coq.type/cut_elim.v
12. Larchey-Wendling, D.: Constructive decision via redundancy-free proof-search. J. Autom. Reason. **64**(7), 1197–1219 (2020)
13. Miller, D., Pimentel, E.: A formal framework for specifying sequent calculus proof systems. Theor. Comput. Sci. **474**, 98–116 (2013)
14. Negri, S.: Proof analysis in modal logic. J. Philos. Logic **34**(5–6), 507–544 (2005)
15. Olarte, C., Pimentel, E., Xavier, B.: A fresh view of linear logic as a logical framework. In: LSFA 2020. ENTCS, vol. 351, pp. 143–165. Elsevier (2020)
16. Pfenning, F.: Structural cut elimination. In: LICS 1995, pp. 156–166. IEEE Computer Society (1995)
17. Simmons, R.J.: Structural focalization. ACM Trans. Comput. Log. **15**(3), 21:1–21:33 (2014)
18. Tews, H.: Formalizing cut elimination of coalgebraic logics in Coq. In: Galmiche, D., Larchey-Wendling, D. (eds.) TABLEAUX 2013. LNCS (LNAI), vol. 8123, pp. 257–272. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40537-2_22
19. Troelstra, A., Schwichtenberg, H.: Basic Proof Theory. Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1996)

20. Urban, C., Zhu, B.: Revisiting cut-elimination: one difficult proof is really a proof. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 409–424. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70590-1_28
21. Xavier, B., Olarte, C., Reis, G., Nigam, V.: Mechanizing focused linear logic in Coq. In: LSFA 2017. ENTCS, vol. 338, pp. 219–236. Elsevier (2017)