Intuitionistic Linear Logic

Filip Smola

November 28, 2024

Contents

1 Intuitionistic Linear Logic

theory *ILL*
  imports
    Main
    HOL−Combinatorics.Permutations
begin

Note that in this theory we often use procedural proofs rather than structured ones. We find these to be more informative about how the basic rules of the logic are used when compared to collecting all the rules in one call of an automated method.

1.1 Deep Embedding of Propositions

We formalise ILL propositions as a datatype, parameterised by the type of propositional variables. The propositions are:

- Propositional variables
- Times of two terms, with unit **1**
- With of two terms, with unit ⊤
- Plus of two terms, with unit **0**
- Linear implication, with no unit
- Exponential of a term

datatype *'a ill-prop* =
  Prop 'a
  | *Times 'a ill-prop 'a ill-prop* (**infixr** ⊗ *90*) | *One* (**1**)
  | *With 'a ill-prop 'a ill-prop* (**infixr** & *90*) | *Top* (⊤)
  | *Plus 'a ill-prop 'a ill-prop* (**infixr** ⊕ *90*) | *Zero* (**0**)
  | *LImp 'a ill-prop 'a ill-prop* (**infixr** ▷ *90*)
  — Note that Isabelle font does not include ⊸, so we use ▷ instead
  | *Exp 'a ill-prop* (! *1000*)

1.2 Shallow Embedding of Deductions

See Bierman [1] or Kalvala and de Paiva [2] for an overview of valid sequents in ILL.

We first formalise ILL deductions as a relation between a list of propositions (anteceents) and a single proposition (consequent). This constitutes a shallow embedding of deductions (with a deep embedding to follow).

In using a list, as opposed to a multiset, we make the exchange rule explicit. Furthermore, we take as primitive a rule exchanging two propositions and later derive both the corresponding rule for lists of propositions as well as for multisets.

The specific formulation of rules we use here includes lists in more positions than is traditionally done when presenting ILL. This is inspired by the recommendations of Kalvala and de Paiva, intended to improve pattern matching and automation.

inductive *sequent* :: *'a ill-prop list* ⇒ *'a ill-prop* ⇒ *bool* (**infix** ⊢ *60*)
 where
 identity: $[a] \vdash a$
| *exchange*: $[\![G\ @\ [a]\ @\ [b]\ @\ D \vdash c]\!] \Longrightarrow G\ @\ [b]\ @\ [a]\ @\ D \vdash c$
| *cut*: $[\![G \vdash b;\ D\ @\ [b]\ @\ E \vdash c]\!] \Longrightarrow D\ @\ G\ @\ E \vdash c$
| *timesL*: $G\ @\ [a]\ @\ [b]\ @\ D \vdash c \Longrightarrow G\ @\ [a \otimes b]\ @\ D \vdash c$

| *timesR*: $[\![G \vdash a;\ D \vdash b]\!] \Longrightarrow G\ @\ D \vdash a \otimes b$
| *oneL*: $G\ @\ D \vdash c \Longrightarrow G\ @\ [\mathbf{1}]\ @\ D \vdash c$
| *oneR*: $[] \vdash \mathbf{1}$
| *limpL*: $[\![G \vdash a;\ D\ @\ [b]\ @\ E \vdash c]\!] \Longrightarrow G\ @\ D\ @\ [a \rhd b]\ @\ E \vdash c$
| *limpR*: $G\ @\ [a]\ @\ D \vdash b \Longrightarrow G\ @\ D \vdash a \rhd b$
| *withL1*: $G\ @\ [a]\ @\ D \vdash c \Longrightarrow G\ @\ [a\ \&\ b]\ @\ D \vdash c$
| *withL2*: $G\ @\ [b]\ @\ D \vdash c \Longrightarrow G\ @\ [a\ \&\ b]\ @\ D \vdash c$
| *withR*: $[\![G \vdash a;\ G \vdash b]\!] \Longrightarrow G \vdash a\ \&\ b$
| *topR*: $G \vdash \top$
| *plusL*: $[\![G\ @\ [a]\ @\ D \vdash c;\ G\ @\ [b]\ @\ D \vdash c]\!] \Longrightarrow G\ @\ [a \oplus b]\ @\ D \vdash c$
| *plusR1*: $G \vdash a \Longrightarrow G \vdash a \oplus b$
| *plusR2*: $G \vdash b \Longrightarrow G \vdash a \oplus b$
| *zeroL*: $G\ @\ [\mathbf{0}]\ @\ D \vdash c$
| *weaken*: $G\ @\ D \vdash b \Longrightarrow G\ @\ [!a]\ @\ D \vdash b$
| *contract*: $G\ @\ [!a]\ @\ [!a]\ @\ D \vdash b \Longrightarrow G\ @\ [!a]\ @\ D \vdash b$
| *derelict*: $G\ @\ [a]\ @\ D \vdash b \Longrightarrow G\ @\ [!a]\ @\ D \vdash b$
| *promote*: $\mathit{map}\ \mathit{Exp}\ G \vdash a \Longrightarrow \mathit{map}\ \mathit{Exp}\ G \vdash\ !a$

lemmas [*simp*] = *sequent.identity*

1.3 Proposition Equivalence

Two propositions are equivalent when each can be derived from the other

definition *ill-eq* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ *bool* (**infix** $\dashv\vdash$ *60*)
where $a \dashv\vdash b = ([a] \vdash b \wedge [b] \vdash a)$

We show that this is an equivalence relation

lemma *ill-eq-refl* [*simp*]:
$a \dashv\vdash a$
by (*simp add*: *ill-eq-def*)

lemma *ill-eq-sym* [*sym*]:
$a \dashv\vdash b \Longrightarrow b \dashv\vdash a$
by (*smt ill-eq-def*)

lemma *ill-eq-tran* [*trans*]:
$[\![a \dashv\vdash b;\ b \dashv\vdash c]\!] \Longrightarrow a \dashv\vdash c$
using *cut*[*of - - Nil Nil*] **by** (*simp add*: *ill-eq-def*) *blast*

lemma *equivp ill-eq*
by (*metis equivpI ill-eq-refl ill-eq-sym ill-eq-tran reflp-def sympI transp-def*)

lemma *ill-eqI* [*intro*]:
$[a] \vdash b \Longrightarrow [b] \vdash a \Longrightarrow a \dashv\vdash b$
using *ill-eq-def* **by** *blast*

lemma *ill-eqE* [*elim*]:
$a \dashv\vdash b \Longrightarrow ([a] \vdash b \Longrightarrow [b] \vdash a \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *ill-eq-def*)

lemma *ill-eq-lr*: $a \dashv\vdash b \Longrightarrow [a] \vdash b$
  and *ill-eq-rl*: $a \dashv\vdash b \Longrightarrow [b] \vdash a$
  by (*simp-all add*: *ill-eq-def*)

1.4 Useful Rules

We can derive a number of useful rules from the defining ones, especially their specific instantiations.

Particularly useful is an instantiation of the Cut rule that makes it transitive, allowing us to use equational reasoning (**also** and **finally**) to build derivations using single propositions

lemma *simple-cut* [*trans*]:
  $[\![G \vdash b; [b] \vdash c]\!] \Longrightarrow G \vdash c$
  using *cut*[*of - - Nil Nil*] **by** *simp*

lemma
  shows *sequent-Nil-left*: $[] @ G \vdash c \Longrightarrow G \vdash c$
    and *sequent-Nil-right*: $G @ [] \vdash c \Longrightarrow G \vdash c$
  by *simp-all*

lemma *simple-exchange*:
  $[\![[a, b] \vdash c]\!] \Longrightarrow [b, a] \vdash c$
  using *exchange*[*of Nil - - Nil*] **by** *simp*

lemma *simple-timesL*:
  $[\![[a] @ [b] \vdash c]\!] \Longrightarrow [a \otimes b] \vdash c$
  using *timesL*[*of Nil*] **by** *simp*

lemma *simple-withL1*: $[\![[a] \vdash c]\!] \Longrightarrow [a \,\&\, b] \vdash c$
  and *simple-withL2*: $[\![[b] \vdash c]\!] \Longrightarrow [a \,\&\, b] \vdash c$
  using *withL1*[*of Nil*] *withL2*[*of Nil*] **by** *simp-all*

lemma *simple-plusL*:
  $[\![[a] \vdash c; [b] \vdash c]\!] \Longrightarrow [a \oplus b] \vdash c$
  using *plusL*[*of Nil*] **by** *simp*

lemma *simple-weaken*:
  $[!a] \vdash \mathbf{1}$
  using *weaken*[*of Nil*] *oneR* **by** *simp*

lemma *simple-derelict*:
  $[\![[a] \vdash b]\!] \Longrightarrow [!a] \vdash b$
  using *derelict*[*of Nil*] **by** *simp*

lemmas *simple-promote* = *promote*[*of* [-], *unfolded list.map*]

lemma *promote-and-derelict*:

assumes $G \vdash c$
shows *map Exp* $G \vdash\ !c$
proof –
have *ind*: *map Exp* (*take n G*) @ *drop n G* $\vdash c$ **if** *n*: $n \leq$ *length G* **for** *n*
using *n*
proof (*induct n*)
case *0*
then show *?case* **using** *assms* **by** *simp*
next
case (*Suc m*)
moreover have *nth G m* # *drop* (*Suc m*) *G* = *drop m G*
using *Suc Cons-nth-drop-Suc Suc-le-lessD* **by** *blast*
moreover have *map Exp* (*take m G*) @ [! (*nth G m*)] = *map Exp* (*take* (*Suc m*) *G*)
by (*simp add*: *Suc Suc-le-lessD take-Suc-conv-app-nth*)
ultimately show *?case*
using *derelict*[*of map Exp* (*take m G*) *nth G m drop* (*Suc m*) *G c*]
by *simp* (*metis append.assoc append-Cons append-Nil*)
qed

have *map Exp* $G \vdash c$
using *ind*[*of length G*] **by** *simp*
then show *?thesis*
by (*rule promote*)
qed

lemmas *dereliction* = *simple-derelict*[*OF identity*]

lemma *simple-contract*:
$[\![[!a]$ @ $[!a] \vdash b]\!] \Longrightarrow [!a] \vdash b$
using *contract*[*of Nil*] **by** *simp*

lemma *duplicate*:
$[!a] \vdash\ !a \otimes !a$
using *identity simple-contract timesR* **by** *blast*

lemma *unary-promote*:
$[\![[!g] \vdash a]\!] \Longrightarrow [!g] \vdash\ !a$
by (*metis* (*mono-tags*, *opaque-lifting*) *promote list.simps*(*8*) *list.simps*(*9*))

lemma *tensor*:
$[\![[a] \vdash b; [c] \vdash d]\!] \Longrightarrow [a \otimes c] \vdash b \otimes d$
using *simple-timesL timesR* **by** *blast*

lemma *ill-eq-tensor*:
$a \dashv\vdash b \Longrightarrow x \dashv\vdash y \Longrightarrow a \otimes x \dashv\vdash b \otimes y$
by (*simp add*: *ill-eq-def tensor*)

lemma *times-assoc*:

$[(a \otimes b) \otimes c] \vdash a \otimes (b \otimes c)$
proof −
have $[a]$ @ $[b]$ @ $[c] \vdash a \otimes (b \otimes c)$
by (*rule timesR*[*OF identity timesR*, *OF identity identity*])
then have $[a \otimes b]$ @ $[c] \vdash a \otimes (b \otimes c)$
by (*metis timesL append-self-conv2*)
then show *?thesis*
by (*simp add*: *simple-timesL*)
qed

lemma *times-assoc'*:
$[a \otimes (b \otimes c)] \vdash (a \otimes b) \otimes c$
proof −
have ($[a]$ @ $[b]$) @ $[c] \vdash (a \otimes b) \otimes c$
by (*rule timesR*[*OF timesR identity*, *OF identity identity*])
then have $[a]$ @ $[b]$ @ $[c] \vdash (a \otimes b) \otimes c$
by *simp*
then show *?thesis*
using *timesL*[*of* $[a]$ *b c Nil*] **by** (*simp add*: *simple-timesL*)
qed

lemma *simple-limpR*:
$[a] \vdash b \Longrightarrow [\mathbf{1}] \vdash a \rhd b$
using *limpR*[*of Nil* - $[\mathbf{1}]$] *oneL*[*of* $[a]$ *Nil b*] **by** *simp*

lemma *simple-limpR-exp*:
$[a] \vdash b \Longrightarrow [\mathbf{1}] \vdash\ !(a \rhd b)$
proof −
assume $[a] \vdash b$
then have $[] \vdash a \rhd b$
by (*rule simple-cut*[*of Nil* $\mathbf{1}$ $a \rhd b$, *OF oneR simple-limpR*])
then have $[] \vdash\ !(a \rhd b)$
using *promote*[*of Nil* $a \rhd b$] **by** *simp*
then show *?thesis*
using *oneL*[*of Nil*] **by** *simp*
qed

lemma *limp-eval*:
$[a \otimes a \rhd b] \vdash b$
using *limpL*[*of* $[a]$ *a Nil*] *simple-timesL*[*of a*] **by** *simp*

lemma *timesR-intro*:
$[\![G \vdash a;\ D \vdash b;\ G\ @\ D = X]\!] \Longrightarrow X \vdash a \otimes b$
using *timesR* **by** *metis*

lemma *explimp-eval*:
$[a \otimes\ !(a \rhd b)] \vdash b \otimes\ !(a \rhd b)$
apply (*rule simple-timesL*)
apply (*subst* (*2*) *append-Nil2*[*symmetric*], *subst append-assoc*)

apply (*rule contract*)
apply (*subst append-Nil2*, *subst append-assoc*[*symmetric*])
apply (*rule timesR*)

apply (*subst* (*2*) *append-Nil2*[*symmetric*], *subst append-assoc*)
apply (*rule derelict*)
apply (*subst* (*2*) *append-Nil*[*symmetric*], *subst append-assoc*)
apply (*rule limpL*)
apply (*rule identity*)
apply (*subst append-Nil2*, *subst append-Nil*)
apply (*rule identity*)

apply (*rule identity*)
done

lemma *plus-progress*:
$[\![[a] \vdash b; [c] \vdash d]\!] \Longrightarrow [a \oplus c] \vdash b \oplus d$
using *plusR1 plusR2 simple-plusL* **by** *blast*

The following set of rules are based on Proposition 1 of Bierman [1]. Where there is a direct correspondence, we include a comment indicating the specific item in the proposition.

lemma *swap*: — Item 1
$[a \otimes b] \vdash b \otimes a$
proof –
have $[b]$ @ $[a] \vdash b \otimes a$
by (*rule timesR*[*OF identity identity*])
then have $[a]$ @ $[b] \vdash b \otimes a$
using *simple-exchange* **by** *force*
then show *?thesis*
using *simple-timesL* **by** *simp*
qed

lemma *unit*: — Item 2
$[a \otimes \mathbf{1}] \vdash a$
using *oneL*[*of* $[a]$] **by** (*simp add*: *simple-timesL*)

lemma *unit'*: — Item 2
$[a] \vdash a \otimes \mathbf{1}$
using *timesR*[*of* $[a]$ *a Nil* **1**] *oneR* **by** *simp*

lemma *with-swap*: — Item 3
$[a \,\&\, b] \vdash b \,\&\, a$
using *withL2*[*of Nil b*] *withL1*[*of Nil a*] **by** (*simp add*: *withR*)

lemma *with-top*: — Item 4
$a \dashv\vdash a \,\&\, \top$
proof
show $[a \,\&\, \top] \vdash a$

```
    by (simp add: simple-withL1)
next
  show [a] ⊢ a & ⊤
    by (rule withR[OF identity topR])
qed

lemma plus-swap: — Item 5
  [a ⊕ b] ⊢ b ⊕ a
  using plusL[of Nil a] by (simp add: plusR1 plusR2)

lemma plus-zero: — Item 6
  a ⊣⊢ a ⊕ 0
proof
  show [a ⊕ 0] ⊢ a
    using plusL[of Nil a] zeroL[of Nil - a] by simp
next
  show [a] ⊢ a ⊕ 0
    by (simp add: plusR1)
qed

lemma with-distrib: — Item 7
  [a ⊗ (b & c)] ⊢ (a ⊗ b) & (a ⊗ c)
  by (intro withR tensor identity simple-withL1 simple-withL2)

lemma plus-distrib: — Item 8
  [a ⊗ (b ⊕ c)] ⊢ (a ⊗ b) ⊕ (a ⊗ c)
  using timesR[OF identity identity] plusL[of [a] b Nil - c]
  by (metis append-Cons append-Nil plusR1 plusR2 simple-timesL)

lemma plus-distrib': — Item 9
  [(a ⊗ b) ⊕ (a ⊗ c)] ⊢ a ⊗ (b ⊕ c)
  by (simp add: simple-plusL tensor plusR1 plusR2)

lemma times-exp: — Item 10
  [!a ⊗ !b] ⊢ !(a ⊗ b)
proof –
  have [a, b] ⊢ a ⊗ b
    using timesR[of [a]] by simp
  then have [!a, !b] ⊢ a ⊗ b
    by (metis derelict append-Cons append-Nil)
  then have [!a, !b] ⊢ !(a ⊗ b)
    by (metis (mono-tags, opaque-lifting) promote list.simps(8) list.simps(9))
  then show ?thesis
    by (simp add: simple-timesL)
qed

lemma one-exp: — Item 10
  1 ⊣⊢ !(1)
  by (meson ill-eq-def simple-cut simple-limpR-exp simple-weaken unary-promote)
```

lemma — Item 11
  [!*a*] ⊢ **1** & *a* & (!*a* ⊗ !*a*)
  by (*metis identity withR simple-weaken simple-derelict simple-contract timesR*)

lemma — Item 12
  !*a* ⊗ !*b* ⊣⊢ !(*a* & *b*)
proof
  show [!*a* ⊗ !*b*] ⊢ !(*a* & *b*)
  proof −
    have [!*a*, !*b*] ⊢ *a* & *b*
    proof (*rule withR*)
      show [! *a*, ! *b*] ⊢ *a*
        using *weaken*[*of* [!*a*]] *dereliction*[*of a*] **by** *simp*
    next
      show [! *a*, ! *b*] ⊢ *b*
        using *weaken*[*of* [!*b*]] *dereliction*[*of b*] *simple-exchange*[*of* !*b* !*a*] **by** *simp*
    qed
    then show *?thesis*
      using *promote simple-timesL*
       by (*metis* (*mono-tags*, *opaque-lifting*) *append-Cons append-Nil list.simps*(*8*) *list.simps*(*9*))
  qed
next
  show [!(*a* & *b*)] ⊢ !*a* ⊗ !*b*
  proof (*rule simple-contract*, *rule timesR*)
    show [! (*a* & *b*)] ⊢ ! *a*
      by (*simp add*: *unary-promote simple-derelict simple-withL1*)
  next
    show [! (*a* & *b*)] ⊢ ! *b*
      by (*simp add*: *unary-promote simple-derelict simple-withL2*)
  qed
qed

lemma — Item 13
  1 ⊣⊢ !(⊤)
proof
  show [**1**] ⊢ !(⊤)
    using *simple-cut simple-limpR-exp topR unary-promote* **by** *blast*
next
  show [!(⊤)] ⊢ **1**
    by (*rule simple-weaken*)
qed

1.5 Compacting Lists of Propositions

Compacting transforms a list of propositions into a single proposition using the (⊗) operator, taking care to not expand the size when given a list with only one element. This operation allows us to link the meta-level antecedent concatenation with the object-level (⊗) operator, turning a list of antecedents into a single proposition with the same power in proofs.

```
function compact :: 'a ill-prop list ⇒ 'a ill-prop
  where
    xs ≠ [] ⟹ compact (x # xs) = x ⊗ compact xs
  | xs = [] ⟹ compact (x # xs) = x
  | compact [] = 1
  by (metis list.exhaust) simp-all
termination by (relation measure length, auto)
```

For code generation we use an if statement

```
lemma compact-code [code]:
  compact [] = 1
  compact (x # xs) = (if xs = [] then x else x ⊗ compact xs)
  by simp-all
```

Two lists of propositions that compact to the same result must be equal if they do not include any (⊗) or **1** elements. We show first that they must be equally long and then that they must be equal.

```
lemma compact-eq-length:
  assumes ⋀a. a ∈ set xs ⟹ a ≠ 1
      and ⋀a. a ∈ set ys ⟹ a ≠ 1
      and ⋀a u v. a ∈ set xs ⟹ a ≠ u ⊗ v
      and ⋀a u v. a ∈ set ys ⟹ a ≠ u ⊗ v
      and compact xs = compact ys
    shows length xs = length ys
  using assms
proof (induct xs arbitrary: ys)
  case Nil
  then show ?case
   by simp (metis ill-prop.simps(24) list.set-intros(1) compact.elims compact.simps(2))
next
  case xs: (Cons a xs)
  then show ?case
  proof (cases ys)
    case Nil
    then have False
      using xs by simp (metis compact.simps(1,2) ill-prop.distinct(17))
    then show ?thesis
      by metis
  next
    case (Cons a list)
    then show ?thesis
      using xs by simp (metis ill-prop.inject(2) compact.simps(1,2))
  qed
qed

lemma compact-eq:
```

```
  assumes ⋀a. a ∈ set xs ⟹ a ≠ 1
      and ⋀a. a ∈ set ys ⟹ a ≠ 1
      and ⋀a u v. a ∈ set xs ⟹ a ≠ u ⊗ v
      and ⋀a u v. a ∈ set ys ⟹ a ≠ u ⊗ v
      and compact xs = compact ys
    shows xs = ys
proof −
  have length xs = length ys
    using assms by (rule compact-eq-length)
  then show ?thesis
    using assms
  proof (induct xs arbitrary: ys)
    case Nil
    then show ?case by simp
  next
    case xs: (Cons a xs)
    then show ?case
    proof (cases ys)
      case Nil
      then show ?thesis using xs by simp
    next
      case (Cons a list)
      then show ?thesis
        using xs by simp (metis ill-prop.inject(2) compact.simps(1,2))
    qed
  qed
qed
```

Compacting to **1** means the list of propositions was either empty or just that

```
lemma compact-eq-oneE:
  assumes compact xs = 1
  obtains xs = [] | xs = [1]
  using assms
proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by simp (metis compact.simps(1,2) ill-prop.distinct(17))
qed
```

Compacting to (⊗) means the list of propositions was either just that or started with the left-hand proposition and the rest compacts to the right-hand proposition

```
lemma compact-eq-timesE:
  assumes compact xs = x ⊗ y
  obtains xs = [x ⊗ y] | ys where xs = x # ys and compact ys = y
  using assms
```

proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** *simp* (*metis compact.simps*(*1*,*2*) *ill-prop.inject*(*2*))
qed

Compacting to anything but **1** or (⊗) means the list was just that

lemma *compact-eq-otherD*:
  assumes $compact\ xs = a$
    and $\bigwedge x\ y.\ a \neq x \otimes y$
    and $a \neq \mathbf{1}$
  shows $xs = [a]$
  using *assms*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** *simp* (*metis compact-code*(*2*))
qed

For any list of propositions, we can derive its compacted form from it

lemma *identity-list*:
  $G \vdash (compact\ G)$
proof (*induction G rule*: *induct-list012*)
  case *1* **then show** *?case* **by** (*simp add*: *oneR*)
next case (*2 a*) **then show** *?case* **by** *simp*
next case (*3 a b G*) **then show** *?case* **using** *timesR*[*OF identity*] **by** *simp*
qed

For any valid sequent, we can compact any sublist of its antecedents without invalidating it

lemma *compact-split-antecedents*:
  assumes $X\ @\ G\ @\ Y \vdash c$
  shows $n \leq length\ G \Longrightarrow X\ @\ take\ (length\ G - n)\ G\ @\ [compact\ (drop\ (length\ G - n)\ G)]\ @\ Y \vdash c$
proof (*induct n*)
  case *0*
  then show *?case*
    using *oneL*[*of X* @ *G*] *assms* **by** *simp*
next
  case (*Suc n*)
  then obtain *as x bs* **where** G: $G = as\ @\ [x]\ @\ bs$ **and** bs: $length\ bs = n$
   by (*metis Suc-length-conv append-Cons append-Nil append-take-drop-id diff-diff-cancel length-drop*)

  have $X\ @\ take\ (length\ G - n)\ G\ @\ [compact\ (drop\ (length\ G - n)\ G)]\ @\ Y \vdash c$

using *Suc* **by** *simp*
then show *?case*
using *timesL*[*of X* @ *as x compact bs Y c*, *simplified*] *G Suc.prems assms bs*
using *Suc-diff-le Suc-leD Suc-le-D append.assoc append-Cons append-Nil append-eq-append-conv*
append-take-drop-id butlast-snoc diff-Suc-Suc diff-diff-cancel diff-less length-drop
take-hd-drop compact.simps(*1*) *compact.simps*(*2*) *zero-less-Suc*
by (*smt* (*verit*, *ccfv-threshold*))
qed

More generally, compacting a sublist of antecedents does not affect sequent validity

lemma *compact-antecedents*:
(X @ [*compact* G] @ $Y \vdash c$) = (X @ G @ $Y \vdash c$)
proof
assume X @ [*compact* G] @ $Y \vdash c$
then show X @ G @ $Y \vdash c$
using *identity-list cut* **by** *blast*
next
assume X @ G @ $Y \vdash c$
then show X @ [*compact* G] @ $Y \vdash c$
using *compact-split-antecedents*[**where** n = *length* G] **by** *fastforce*
qed

Times with a single proposition can be absorbed into compacting up to proposition equivalence

lemma *times-equivalent-cons*:
$a \otimes$ *compact* b ⊣⊢ *compact* (a # b)
proof (*cases* b)
case *Nil* **then show** *?thesis* **by** (*simp add*: *ill-eq-def unit unit′*)
next
case (*Cons a list*) **then show** *?thesis* **by** *simp*
qed

Times of compacted lists is equivalent to compacting the appended lists

lemma *times-equivalent-append*:
compact $a \otimes$ *compact* b ⊣⊢ *compact* (a @ b)
proof (*induct* a)
case *Nil*
then show *?case*
using *simple-cut*[*OF swap unit*] *simple-cut*[*OF unit′ swap*] *ill-eqI* **by** (*simp*, *blast*)
next
case *assm*: (*Cons a1 a2*)
have *compact* (*a1* # *a2*) $\otimes$ *compact* b ⊣⊢ (*a1* $\otimes$ *compact a2*) $\otimes$ *compact* b
by (*simp add*: *times-equivalent-cons ill-eq-sym ill-eq-tensor*)
also have ... ⊣⊢ *a1* $\otimes$ (*compact a2* $\otimes$ *compact* b)

by (*simp add*: *times-assoc times-assoc′ ill-eqI*)
also have ... ⊣⊢ *a1* ⊗ *compact* (*a2* @ *b*)
using *ill-eq-tensor*[*OF* - *assm*] **by** *simp*
finally show *?case*
by (*simp add*: *ill-eq-tran times-equivalent-cons*)
qed

Any number of single-antecedent sequents can be compacted with the rule ⟦[*?a*] ⊢ *?b*; [*?c*] ⊢ *?d*⟧ ⟹ [*?a* ⊗ *?c*] ⊢ *?b* ⊗ *?d*

lemma *compact-sequent*:
∀ *x* ∈ *set xs*. [*f x*] ⊢ *g x* ⟹ [*compact* (*map f xs*)] ⊢ *compact* (*map g xs*)
proof (*induct xs rule*: *induct-list012*)
case *1* **then show** *?case* **by** *simp*
next case (*2 x*) **then show** *?case* **by** *simp*
next case (*3 x y zs*) **then show** *?case* **by** (*simp add*: *tensor*)
qed

Any number of equivalences can be compacted together

lemma *compact-equivalent*:
∀ *x* ∈ *set xs*. *f x* ⊣⊢ *g x* ⟹ *compact* (*map f xs*) ⊣⊢ *compact* (*map g xs*)
by (*simp add*: *ill-eqI*[*OF compact-sequent compact-sequent*] *ill-eq-lr ill-eq-rl*)

1.6 Multiset Exchange

Recall that our (⊢) definition uses explicit single-proposition exchange. We now derive a rule for exchanging lists of propositions and then a rule that uses multisets to disregard the antecedent order entirely.

We can exchange lists of propositions by stepping through *compact*

lemma *exchange-list*:
G @ *A* @ *B* @ *D* ⊢ *c* ⟹ *G* @ *B* @ *A* @ *D* ⊢ *c*
proof −
assume *G* @ *A* @ *B* @ *D* ⊢ *c*
then have *G* @ [*compact A*] @ *B* @ *D* ⊢ *c*
using *compact-antecedents* **by** *force*
then have *G* @ [*compact A*] @ [*compact B*] @ *D* ⊢ *c*
using *compact-antecedents*[**where** *X* = *G* @ [*compact A*] **and** *G* = *B*] **by** *force*
then have *G* @ [*compact B*] @ [*compact A*] @ *D* ⊢ *c*
using *exchange* **by** *simp*
then have *G* @ [*compact B*] @ *A* @ *D* ⊢ *c*
using *compact-antecedents*[**where** *X* = *G* @ [*compact B*] **and** *G* = *A*] **by** *force*
then show *?thesis*
using *compact-antecedents* **by** *force*
qed

lemma *simple-exchange-list*:
⟦*A* @ *B* ⊢ *c*⟧ ⟹ *B* @ *A* ⊢ *c*
using *exchange-list*[*of Nil* - - *Nil*] **by** *simp*

By applying the list exchange rule multiple times, the lists do not need to be adjacent

```
lemma exchange-separated:
  G @ A @ X @ B @ D ⊢ c ⟹ G @ B @ X @ A @ D ⊢ c
  by (metis append.assoc exchange-list)
```

Single transposition in the antecedents does not invalidate a sequent

```
lemma exchange-transpose:
  assumes G ⊢ c
      and a ∈ {..<length G}
      and b ∈ {..<length G}
    shows permute-list (transpose a b) G ⊢ c
proof −
  consider a < b | a = b | b < a
    using not-less-iff-gr-or-eq by blast
  moreover { fix x y
    assume x-in [simp]: x ∈ {..<length G}
       and y-in [simp]: y ∈ {..<length G}
       and xy [arith]: x < y

    have G = take x G @ drop x G
      by simp
    also have ... = take x G @ nth G x # drop (Suc x) G
      by simp (metis x-in id-take-nth-drop lessThan-iff)
    also have ... = take x G @ nth G x # take (y − Suc x) (drop (Suc x) G) @
drop y G
    by simp (metis Suc-leI add.commute append-take-drop-id drop-drop le-add-diff-inverse
xy)
    also have
      ... = take x G @ nth G x # take (y − Suc x) (drop (Suc x) G) @ nth G y #
drop (Suc y) G
      by simp (metis Cons-nth-drop-Suc y-in lessThan-iff)
    finally have G:
      G = take x G @ nth G x # take (y − Suc x) (drop (Suc x) G) @ nth G y #
drop (Suc y) G .

    have take x G @ [nth G y] @ take (y − Suc x) (drop (Suc x) G) @ [nth G x]
@ drop (Suc y) G ⊢ c
      by (rule exchange-separated, simp add: G[symmetric] assms(1))
    moreover have
       permute-list (transpose x y) G
       = take x G @ nth G y # take (y − Suc x) (drop (Suc x) G) @ nth G x #
drop (Suc y) G
      unfolding list-eq-iff-nth-eq drop-Suc
    proof safe
      show
         length (permute-list (Transposition.transpose x y) G)
         = length (take x G @ nth G y # take (y − Suc x) (drop x (tl G)) @ nth G
x # drop y (tl G))
```

using *y-in* **by** *simp*
next
fix *i*
assume $i < length\ (permute\text{-}list\ (Transposition.transpose\ x\ y)\ G)$
then show $nth\ (permute\text{-}list\ (Transposition.transpose\ x\ y)\ G)\ i =$
$nth\ (take\ x\ G\ @\ nth\ G\ y\ \#\ take\ (y - Suc\ x)\ (drop\ x\ (tl\ G))\ @\ nth\ G\ x\ \#\ drop\ y\ (tl\ G))\ i$
by (*simp add*: *permute-list-def transpose-def nth-append min-diff nth-tl*)
qed
ultimately have $permute\text{-}list\ (transpose\ x\ y)\ G \vdash c$
by *simp*
}
ultimately show *?thesis*
using *assms* **by** (*metis permute-list-id transpose-commute transpose-same*)
qed

More generally, by transposition being involutive, a single antecedent transposition does not affect sequent validity

lemma *exchange-permute-eq*:
assumes $a \in \{..<length\ G\}$
and $b \in \{..<length\ G\}$
shows $permute\text{-}list\ (transpose\ a\ b)\ G \vdash c = G \vdash c$
using *assms exchange-transpose transpose-comp-involutory*
by (*metis length-permute-list permute-list-compose permute-list-id permutes-swap-id*)

Validity of a sequent is not affected by replacing any antecedent sublist with a list that represents the same multiset. This is because lists representing equal multisets are connected by a permutation, which is a sequence of transpositions and as such does not affect validity.

lemma *exchange-mset*:
$mset\ A = mset\ B \Longrightarrow G\ @\ A\ @\ D \vdash c = G\ @\ B\ @\ D \vdash c$
proof −
{ fix $X\ Y$ :: *'a ill-prop list*
assume $X \vdash c$ **and** $mset\ X = mset\ Y$
then have $Y \vdash c$
proof (*elim mset-eq-permutation*)
fix *p*
assume *p permutes* $\{..<length\ Y\}$
moreover have *finite* $\{..<length\ Y\}$
by *simp*
moreover assume $X \vdash c$ **and** $permute\text{-}list\ p\ Y = X$
ultimately show $Y \vdash c$
proof (*induct arbitrary*: *X rule*: *permutes-rev-induct*)
case *id* **then show** *?case* **by** *simp*
next
case (*swap a b p*)
then show *?case*
by (*metis permute-list-compose permutes-swap-id length-permute-list exchange-permute-eq*)

```
    qed
  qed
} note base = this

show mset A = mset B ⟹ G @ A @ D ⊢ c = G @ B @ D ⊢ c
  by (standard ; simp add: base)
qed
```

1.7 Additional Lemmas

These rules are based on Figure 2 of Kalvala and de Paiva [2], labelled by them as "additional rules for proof search". We present them out of order because we use some in the proofs of the others, but annotate them with the original labels as comments.

```
lemma ill-mp1: — mp₁
  assumes A @ [b] @ B @ C ⊢ c
    shows A @ [a] @ B @ [a ▷ b] @ C ⊢ c
proof –
  have [a] @ [a ▷ b] ⊢ b
    using limpL[of [a] a Nil] by simp
  then have A @ [a] @ [a ▷ b] @ B @ C ⊢ c
    using assms cut[of - b A B @ C c] by force
  then show ?thesis
    using exchange-list[of A @ [a] [a ▷ b]] by simp
qed

lemmas simple-mp1 = ill-mp1[of Nil - Nil Nil, simplified, OF identity]

lemma — raa₁
  G @ [!b] @ D @ [!b ▷ 0] @ E ⊢ a
  using zeroL ill-mp1 by blast

lemma ill-mp2: — mp₂
  assumes A @ [b] @ B @ C ⊢ c
    shows A @ [a ▷ b] @ B @ [a] @ C ⊢ c
  using ill-mp1[OF assms] exchange-list by (metis append.assoc)

lemmas simple-mp2 = ill-mp2[of Nil - Nil Nil, simplified, OF identity]

lemma — raa₂
  G @ [!b ▷ 0] @ D @ [!b] @ P ⊢ A
  using zeroL ill-mp2 by blast

lemma — ⊗-&
  assumes G @ [(!a ▷ 0) & (!b ▷ 0)] @ D ⊢ c
    shows G @ [!(!(a ⊕ b) ▷ 0)] @ D ⊢ c
proof –
  note exp-injL = unary-promote[OF simple-derelict, OF plusR1[OF identity, of a
```

b]]
  and *exp-injR* = *unary-promote*[*OF simple-derelict*, *OF plusR2*[*OF identity*, *of b a*]]
 have [!(!($a \oplus b$) $\triangleright$ **0**)] $\vdash$ (!a $\triangleright$ **0**) & (!b $\triangleright$ **0**)
  apply (*rule withR* ; *rule simple-derelict* , *rule limpR*[*of Nil*, *simplified*])
   apply (*rule cut*[*OF exp-injL*, *of Nil*, *simplified*], *rule simple-mp1*)
   apply (*rule cut*[*OF exp-injR*, *of Nil*, *simplified*], *rule simple-mp1*)
  done
 then show *?thesis*
  using *assms cut* **by** *blast*
qed

lemma — &-*lemma*
 assumes *G* @ [!*a*, !*b*] @ *D* $\vdash$ *c*
  shows *G* @ [!(*a* & *b*)] @ *D* $\vdash$ *c*
proof –
 have *as*: [!(*a* & *b*)] $\vdash$!*a*
  apply (*rule unary-promote*)
  apply (*rule simple-derelict*)
  by (*rule simple-withL1*[*OF identity*])
 have *bs*: [!(*a* & *b*)] $\vdash$!*b*
  apply (*rule unary-promote*)
  apply (*rule simple-derelict*)
  by (*rule simple-withL2*[*OF identity*])
 show *?thesis*
  apply (*rule contract*)
  using *cut*[*OF as*, *of G* [!*b*] @ *D c*] *cut*[*OF bs*, *of G* @ [!(*a* & *b*)] *D c*] *assms*
  by *simp*
qed

lemma — $\multimap_L$-*lemma*
 assumes *G* @ *D* $\vdash$ *a*
 shows *G* @ [!(*a* $\triangleright$ *b*)] @ *D* $\vdash$ *b*
 apply (*rule derelict*)
 using *exchange-list*[*of G D* [*a* $\triangleright$ *b*] *Nil b*, *simplified*]
    limpL[*OF assms*, *of Nil b Nil b*, *simplified*]
 by *simp*

lemma — $\multimap_R$-*lemma*
 assumes [*a*, !*a*] @ *G* $\vdash$ *b*
 shows *G* $\vdash$!*a* $\triangleright$ *b*
 apply (*rule limpR*[*of - - Nil*, *simplified*])
 apply (*rule exchange-list*[*of Nil* [!*a*] - *Nil*, *simplified*])
 apply (*rule contract*[*of Nil*, *simplified*])
 apply (*rule derelict*[*of Nil*, *simplified*])
 using *assms* **by** *simp*

lemma — *a-not-a*
 assumes *G* @ [!*a* $\triangleright$ **0**] @ *D* $\vdash$ *b*

```
  shows G @ [!a ⊳ (!a ⊳ 0)] @ D ⊢ b
proof –
  have [!a ⊳ (!a ⊳ 0)] ⊢ !a ⊳ 0
    apply (rule limpR[of - - Nil, simplified])
    apply (rule contract[of - - Nil, simplified])
    apply simp
    apply (rule ill-mp2[of Nil -  Nil [!a], simplified])
    by (rule simple-mp2)
  then show ?thesis
    using cut[OF - assms] by blast
qed

end
theory Proof
  imports ILL
begin
```

1.8 Deep Embedding of Deductions

To directly manipulate ILL deductions themselves we deeply embed them as a datatype. This datatype has a constructor to represent each introduction rule of (⊢), with the ILL propositions and further deductions those rules use as arguments. Additionally, it has a constructor to represent premises (sequents assumed to be valid) which allow us to represent contingent deductions.

The datatype is parameterised by two type variables:

- *'a* represents the propositional variables for the contained ILL propositions, and
- *'l* represents labels we associate with premises.

```
datatype ('a, 'l) ill-deduct =
    Premise 'a ill-prop list 'a ill-prop 'l
  | Identity 'a ill-prop
  | Exchange 'a ill-prop list 'a ill-prop 'a ill-prop 'a ill-prop list 'a ill-prop
      ('a, 'l) ill-deduct
  | Cut 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop list 'a ill-prop
      ('a, 'l) ill-deduct ('a, 'l) ill-deduct
  | TimesL 'a ill-prop list 'a ill-prop 'a ill-prop 'a ill-prop list 'a ill-prop
      ('a, 'l) ill-deduct
  | TimesR 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct
      ('a, 'l) ill-deduct
  | OneL 'a ill-prop list 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct
  | OneR
  | LimpL 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop 'a ill-prop list
      'a ill-prop ('a, 'l) ill-deduct ('a, 'l) ill-deduct
  | LimpR 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct
```

| *WithL1 'a ill-prop list 'a ill-prop 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct*
| *WithL2 'a ill-prop list 'a ill-prop 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct*
| *WithR 'a ill-prop list 'a ill-prop 'a ill-prop ('a, 'l) ill-deduct ('a, 'l) ill-deduct*
| *TopR 'a ill-prop list*
| *PlusL 'a ill-prop list 'a ill-prop 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct ('a, 'l) ill-deduct*
| *PlusR1 'a ill-prop list 'a ill-prop 'a ill-prop ('a, 'l) ill-deduct*
| *PlusR2 'a ill-prop list 'a ill-prop 'a ill-prop ('a, 'l) ill-deduct*
| *ZeroL 'a ill-prop list 'a ill-prop list 'a ill-prop*
| *Weaken 'a ill-prop list 'a ill-prop list 'a ill-prop 'a ill-prop ('a, 'l) ill-deduct*
| *Contract 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct*
| *Derelict 'a ill-prop list 'a ill-prop 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct*
| *Promote 'a ill-prop list 'a ill-prop ('a, 'l) ill-deduct*

1.8.1 Semantics

With every deduction we associate the antecedents and consequent of its conclusion sequent

primrec *antecedents* :: (*'a, 'l*) *ill-deduct* ⇒ *'a ill-prop list*
where
antecedents (*Premise G c l*) = *G*
| *antecedents* (*Identity a*) = [*a*]
| *antecedents* (*Exchange G a b D c P*) = *G* @ [*b*] @ [*a*] @ *D*
| *antecedents* (*Cut G b D E c P Q*) = *D* @ *G* @ *E*
| *antecedents* (*TimesL G a b D c P*) = *G* @ [*a* ⊗ *b*] @ *D*
| *antecedents* (*TimesR G a D b P Q*) = *G* @ *D*
| *antecedents* (*OneL G D c P*) = *G* @ [**1**] @ *D*
| *antecedents* (*OneR*) = []
| *antecedents* (*LimpL G a D b E c P Q*) = *G* @ *D* @ [*a* ▷ *b*] @ *E*
| *antecedents* (*LimpR G a D b P*) = *G* @ *D*
| *antecedents* (*WithL1 G a b D c P*) = *G* @ [*a* & *b*] @ *D*
| *antecedents* (*WithL2 G a b D c P*) = *G* @ [*a* & *b*] @ *D*
| *antecedents* (*WithR G a b P Q*) = *G*
| *antecedents* (*TopR G*) = *G*
| *antecedents* (*PlusL G a b D c P Q*) = *G* @ [*a* ⊕ *b*] @ *D*
| *antecedents* (*PlusR1 G a b P*) = *G*
| *antecedents* (*PlusR2 G a b P*) = *G*
| *antecedents* (*ZeroL G D c*) = *G* @ [**0**] @ *D*
| *antecedents* (*Weaken G D b a P*) = *G* @ [!*a*] @ *D*
| *antecedents* (*Contract G a D b P*) = *G* @ [!*a*] @ *D*
| *antecedents* (*Derelict G a D b P*) = *G* @ [!*a*] @ *D*
| *antecedents* (*Promote G a P*) = *map Exp G*

primrec *consequent* :: (*'a, 'l*) *ill-deduct* ⇒ *'a ill-prop*
where
consequent (*Premise G c l*) = *c*
| *consequent* (*Identity a*) = *a*

| *consequent* (*Exchange G a b D c P*) = *c*
| *consequent* (*Cut G b D E c P Q*) = *c*
| *consequent* (*TimesL G a b D c P*) = *c*
| *consequent* (*TimesR G a D b P Q*) = *a* ⊗ *b*
| *consequent* (*OneL G D c P*) = *c*
| *consequent* (*OneR*) = **1**
| *consequent* (*LimpL G a D b E c P Q*) = *c*
| *consequent* (*LimpR G a D b P*) = *a* ▷ *b*
| *consequent* (*WithL1 G a b D c P*) = *c*
| *consequent* (*WithL2 G a b D c P*) = *c*
| *consequent* (*WithR G a b P Q*) = *a* & *b*
| *consequent* (*TopR G*) = ⊤
| *consequent* (*PlusL G a b D c P Q*) = *c*
| *consequent* (*PlusR1 G a b P*) = *a* ⊕ *b*
| *consequent* (*PlusR2 G a b P*) = *a* ⊕ *b*
| *consequent* (*ZeroL G D c*) = *c*
| *consequent* (*Weaken G D b a P*) = *b*
| *consequent* (*Contract G a D b P*) = *b*
| *consequent* (*Derelict G a D b P*) = *b*
| *consequent* (*Promote G a P*) = !*a*

We define a sequent datatype for presenting deduction tree conclusions, deeply embedding (possibly invalid) sequents themselves.

Note: these are not used everywhere, separate antecedents and consequent tend to work better for proof automation. For instance, the full conclusion cannot be derived where only facts about antecedents are known.

datatype *'a ill-sequent* = *Sequent 'a ill-prop list 'a ill-prop*

Validity of deeply embedded sequents is defined by the shallow (⊢) relation

primrec *ill-sequent-valid* :: *'a ill-sequent* ⇒ *bool*
  where *ill-sequent-valid* (*Sequent a c*) = *a* ⊢ *c*

We set up a notation bundle to have infix ⊢ for stand for the sequent datatype and not the relation

bundle *deep-sequent*
begin
no-notation *sequent* (**infix** ⊢ *60*)
notation *Sequent* (**infix** ⊢ *60*)
end

context
  includes *deep-sequent*
begin

With deeply embedded sequents we can define the conclusion of every deduction

primrec *ill-conclusion* :: (*'a*, *'l*) *ill-deduct* ⇒ *'a ill-sequent*

where
ill-conclusion (*Premise G c l*) = $G \vdash c$
| *ill-conclusion* (*Identity a*) = $[a] \vdash a$
| *ill-conclusion* (*Exchange G a b D c P*) = G @ $[b]$ @ $[a]$ @ $D \vdash c$
| *ill-conclusion* (*Cut G b D E c P Q*) = D @ G @ $E \vdash c$
| *ill-conclusion* (*TimesL G a b D c P*) = G @ $[a \otimes b]$ @ $D \vdash c$
| *ill-conclusion* (*TimesR G a D b P Q*) = G @ $D \vdash a \otimes b$
| *ill-conclusion* (*OneL G D c P*) = G @ $[\mathbf{1}]$ @ $D \vdash c$
| *ill-conclusion* (*OneR*) = $[] \vdash \mathbf{1}$
| *ill-conclusion* (*LimpL G a D b E c P Q*) = G @ D @ $[a \rhd b]$ @ $E \vdash c$
| *ill-conclusion* (*LimpR G a D b P*) = G @ $D \vdash a \rhd b$
| *ill-conclusion* (*WithL1 G a b D c P*) = G @ $[a \mathbin{\&} b]$ @ $D \vdash c$
| *ill-conclusion* (*WithL2 G a b D c P*) = G @ $[a \mathbin{\&} b]$ @ $D \vdash c$
| *ill-conclusion* (*WithR G a b P Q*) = $G \vdash a \mathbin{\&} b$
| *ill-conclusion* (*TopR G*) = $G \vdash \top$
| *ill-conclusion* (*PlusL G a b D c P Q*) = G @ $[a \oplus b]$ @ $D \vdash c$
| *ill-conclusion* (*PlusR1 G a b P*) = $G \vdash a \oplus b$
| *ill-conclusion* (*PlusR2 G a b P*) = $G \vdash a \oplus b$
| *ill-conclusion* (*ZeroL G D c*) = G @ $[\mathbf{0}]$ @ $D \vdash c$
| *ill-conclusion* (*Weaken G D b a P*) = G @ $[!a]$ @ $D \vdash b$
| *ill-conclusion* (*Contract G a D b P*) = G @ $[!a]$ @ $D \vdash b$
| *ill-conclusion* (*Derelict G a D b P*) = G @ $[!a]$ @ $D \vdash b$
| *ill-conclusion* (*Promote G a P*) = *map Exp* $G \vdash !a$

This conclusion is the same as what *antecedents* and *consequent* express

lemma *ill-conclusionI* [*intro*!]:
assumes *antecedents P* = *G*
and *consequent P* = *c*
shows *ill-conclusion P* = $G \vdash c$
using *assms* **by** (*induction P*) *simp-all*

lemma *ill-conclusionE* [*elim*!]:
assumes *ill-conclusion P* = $G \vdash c$
obtains *antecedents P* = *G*
and *consequent P* = *c*
using *assms* **by** (*induction P*) *simp-all*

lemma *ill-conclusion-alt*:
(*ill-conclusion P* = $G \vdash c$) = (*antecedents P* = $G \wedge$ *consequent P* = *c*)
by *blast*

lemma *ill-conclusion-antecedents*: *ill-conclusion P* = $G \vdash c \Longrightarrow$ *antecedents P* = *G*
and *ill-conclusion-consequent*: *ill-conclusion P* = $G \vdash c \Longrightarrow$ *consequent P* = *c*
by *blast+*

Every deduction is well-formed if all deductions it relies on are well-formed and have the form required by the corresponding *sequent* rule.

primrec *ill-deduct-wf* :: ('*a*, '*l*) *ill-deduct* ⇒ *bool*

where
ill-deduct-wf (*Premise G c l*) = *True*
| *ill-deduct-wf* (*Identity a*) = *True*
| *ill-deduct-wf* (*Exchange G a b D c P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ [*b*] @ *D* ⊢ *c*)
| *ill-deduct-wf* (*Cut G b D E c P Q*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *b* ∧
ill-deduct-wf Q ∧ *ill-conclusion Q* = *D* @ [*b*] @ *E* ⊢ *c*)
| *ill-deduct-wf* (*TimesL G a b D c P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ [*b*] @ *D* ⊢ *c*)
| *ill-deduct-wf* (*TimesR G a D b P Q*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *a* ∧
ill-deduct-wf Q ∧ *ill-conclusion Q* = *D* ⊢ *b*)
| *ill-deduct-wf* (*OneL G D c P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ *D* ⊢ *c*)
| *ill-deduct-wf* (*OneR*) = *True*
| *ill-deduct-wf* (*LimpL G a D b E c P Q*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *a* ∧
ill-deduct-wf Q ∧ *ill-conclusion Q* = *D* @ [*b*] @ *E* ⊢ *c*)
| *ill-deduct-wf* (*LimpR G a D b P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ *D* ⊢ *b*)
| *ill-deduct-wf* (*WithL1 G a b D c P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ *D* ⊢ *c*)
| *ill-deduct-wf* (*WithL2 G a b D c P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*b*] @ *D* ⊢ *c*)
| *ill-deduct-wf* (*WithR G a b P Q*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *a* ∧
ill-deduct-wf Q ∧ *ill-conclusion Q* = *G* ⊢ *b*)
| *ill-deduct-wf* (*TopR G*) = *True*
| *ill-deduct-wf* (*PlusL G a b D c P Q*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ *D* ⊢ *c* ∧
ill-deduct-wf Q ∧ *ill-conclusion Q* = *G* @ [*b*] @ *D* ⊢ *c*)
| *ill-deduct-wf* (*PlusR1 G a b P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *a*)
| *ill-deduct-wf* (*PlusR2 G a b P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* ⊢ *b*)
| *ill-deduct-wf* (*ZeroL G D c*) = *True*
| *ill-deduct-wf* (*Weaken G D b a P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ *D* ⊢ *b*)
| *ill-deduct-wf* (*Contract G a D b P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [!*a*] @ [!*a*] @ *D* ⊢ *b*)
| *ill-deduct-wf* (*Derelict G a D b P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *G* @ [*a*] @ *D* ⊢ *b*)
| *ill-deduct-wf* (*Promote G a P*) =
(*ill-deduct-wf P* ∧ *ill-conclusion P* = *map Exp G* ⊢ *a*)

In some proofs phasing well-formedness in terms of *antecedents* and *consequent* is more useful.

lemmas *ill-deduct-wf-alt* = *ill-deduct-wf.simps*[*unfolded ill-conclusion-alt*]

end

Premises of a deduction can be gathered recursively. Because every element of the result is an instance of *Premise*, we represent them with the relevant three parameters (antecedents, consequent, label).

primrec *ill-deduct-premises*
  :: ($'a$, $'l$) *ill-deduct* ⇒ ($'a$ *ill-prop list* × $'a$ *ill-prop* × $'l$) *list*
 where
  ill-deduct-premises (*Premise G c l*) = [(*G*, *c*, *l*)]
 | *ill-deduct-premises* (*Identity a*) = []
 | *ill-deduct-premises* (*Exchange G a b D c P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*Cut G b D E c P Q*) =
   (*ill-deduct-premises P* @ *ill-deduct-premises Q*)
 | *ill-deduct-premises* (*TimesL G a b D c P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*TimesR G a D b P Q*) =
   (*ill-deduct-premises P* @ *ill-deduct-premises Q*)
 | *ill-deduct-premises* (*OneL G D c P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*OneR*) = []
 | *ill-deduct-premises* (*LimpL G a D b E c P Q*) =
   (*ill-deduct-premises P* @ *ill-deduct-premises Q*)
 | *ill-deduct-premises* (*LimpR G a D b P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*WithL1 G a b D c P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*WithL2 G a b D c P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*WithR G a b P Q*) =
   (*ill-deduct-premises P* @ *ill-deduct-premises Q*)
 | *ill-deduct-premises* (*TopR G*) = []
 | *ill-deduct-premises* (*PlusL G a b D c P Q*) =
   (*ill-deduct-premises P* @ *ill-deduct-premises Q*)
 | *ill-deduct-premises* (*PlusR1 G a b P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*PlusR2 G a b P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*ZeroL G D c*) = []
 | *ill-deduct-premises* (*Weaken G D b a P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*Contract G a D b P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*Derelict G a D b P*) = *ill-deduct-premises P*
 | *ill-deduct-premises* (*Promote G a P*) = *ill-deduct-premises P*

1.8.2 Soundness

Deeply embedded deductions are sound with respect to (⊢) in the sense that the conclusion of any well-formed deduction is a valid sequent if all of its premises are assumed to be valid sequents. This is proven easily, because our definitions stem from the (⊢) relation.

lemma *ill-deduct-sound*:
 assumes *ill-deduct-wf P*
  and ⋀*a c l*. (*a*, *c*, *l*) ∈ *set* (*ill-deduct-premises P*) ⟹ *ill-sequent-valid* (*Sequent a c*)
  shows *ill-sequent-valid* (*ill-conclusion P*)

using *assms*
proof (*induct P*)
case (*Premise G c l*) **then show** *?case* **by** *simp* **next**
case (*Identity x*) **then show** *?case* **by** *simp* **next**
case (*Exchange x1a x2 x3 x4 x5 x6*) **then show** *?case* **using** *exchange* **by** *simp blast* **next**
case (*Cut x1a x2 x3 x4 x5 x6 x7*) **then show** *?case* **using** *cut* **by** *simp blast* **next**
case (*TimesL x1a x2 x3 x4 x5 x6*) **then show** *?case* **using** *timesL* **by** *simp blast* **next**
case (*TimesR x1a x2 x3 x4 x5 x6*) **then show** *?case* **using** *timesR* **by** *simp blast* **next**
case (*OneL x1a x1b x2 x3*) **then show** *?case* **using** *oneL* **by** *simp blast* **next**
case *OneR* **then show** *?case* **using** *oneR* **by** *simp* **next**
case (*LimpL x1a x2 x3 x4 x5 x6 x7*) **then show** *?case* **using** *limpL* **by** *simp blast* **next**
case (*LimpR x1a x2 x3 x4 x5*) **then show** *?case* **using** *limpR* **by** *simp blast* **next**
case (*WithL1 x1a x2 x3 x4 x5 x6*) **then show** *?case* **using** *withL1* **by** *simp blast* **next**
case (*WithL2 x1a x2 x3 x4 x5 x6*) **then show** *?case* **using** *withL2* **by** *simp blast* **next**
case (*WithR x1a x2 x3 x4 x5*) **then show** *?case* **using** *withR* **by** *simp blast* **next**
case (*TopR x*) **then show** *?case* **using** *topR* **by** *simp blast* **next**
case (*PlusL x1a x2 x3 x4 x5 x6 x7*) **then show** *?case* **using** *plusL* **by** *simp blast* **next**
case (*PlusR1 x1a x2 x3 x4*) **then show** *?case* **using** *plusR1* **by** *simp blast* **next**
case (*PlusR2 x1a x2 x3 x4*) **then show** *?case* **using** *plusR2* **by** *simp blast* **next**
case (*ZeroL x1a x2 x3*) **then show** *?case* **using** *zeroL* **by** *simp blast* **next**
case (*Weaken x1a x2 x3 x4 x5*) **then show** *?case* **using** *weaken* **by** *simp blast* **next**
case (*Contract x1a x2 x3 x4 x5*) **then show** *?case* **using** *contract* **by** *simp blast* **next**
case (*Derelict x1a x2 x3 x4 x5*) **then show** *?case* **using** *derelict* **by** *simp blast* **next**
case (*Promote x1a x2 x3*) **then show** *?case* **using** *promote* **by** *simp blast*
qed

1.8.3 Completeness

Deeply embedded deductions are complete with respect to (⊢) in the sense that for any valid sequent there exists a well-formed deduction with no premises that has it as its conclusion. This is proven easily, because the deduction nodes map directly onto the rules of the (⊢) relation.

lemma *ill-deduct-complete*:
assumes $G \vdash c$
shows $\exists P.$ *ill-conclusion* $P =$ *Sequent* $G\ c \wedge$ *ill-deduct-wf* $P \wedge$ *ill-deduct-premises* $P = []$

using *assms*
proof (*induction rule*: *sequent.induct*)
case (*identity a*)
then show *?case*
using *ill-conclusion.simps*(*2*) **by** *fastforce*
next
case (*exchange G a b D c*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ [*b*] @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*Exchange G a b D c P*) **and** *ill-deduct-premises* (*Exchange G a b D c P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*3*))
next
case (*cut G b D E c*)
then obtain *P Q* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent G b* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
and *ill-conclusion Q* = *Sequent* (*D* @ [*b*] @ *E*) *c* ∧ *ill-deduct-wf Q* ∧ *ill-deduct-premises Q* = []
by *blast*
then have *ill-deduct-wf* (*Cut G b D E c P Q*) **and** *ill-deduct-premises* (*Cut G b D E c P Q*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*4*))
next
case (*timesL G a b D c*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ [*b*] @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*TimesL G a b D c P*) **and** *ill-deduct-premises* (*TimesL G a b D c P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*5*))
next
case (*timesR G a D b*)
then obtain *P Q* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent G a* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
and *ill-conclusion Q* = *Sequent D b* ∧ *ill-deduct-wf Q* ∧ *ill-deduct-premises Q* = []
by *blast*
then have *ill-deduct-wf* (*TimesR G a D b P Q*) **and** *ill-deduct-premises* (*TimesR

G a D b P Q) = []
    by *simp-all*
  then show *?case*
    by (*meson ill-conclusion.simps*(*6*))
next
  case (*oneL G D c*)
  then obtain *P* :: (*'a*, *'b*) *ill-deduct*
   where *ill-conclusion P* = *Sequent* (*G* @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
    by *blast*
  then have *ill-deduct-wf* (*OneL G D c P*) **and** *ill-deduct-premises* (*OneL G D c P*) = []
    by *simp-all*
  then show *?case*
    by (*meson ill-conclusion.simps*(*7*))
next
  case *oneR*
  then show *?case*
    using *ill-conclusion.simps*(*8*) **by** *fastforce*
next
  case (*limpL G a D b E c*)
  then obtain *P Q* :: (*'a*, *'b*) *ill-deduct*
    where *ill-conclusion P* = *Sequent G a* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
      and *ill-conclusion Q* = *Sequent* (*D* @ [*b*] @ *E*) *c* ∧ *ill-deduct-wf Q* ∧ *ill-deduct-premises Q* = []
    by *blast*
  then have *ill-deduct-wf* (*LimpL G a D b E c P Q*) **and** *ill-deduct-premises* (*LimpL G a D b E c P Q*) = []
    by *simp-all*
  then show *?case*
    by (*meson ill-conclusion.simps*(*9*))
next
  case (*limpR G a D b*)
  then obtain *P* :: (*'a*, *'b*) *ill-deduct*
    where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ *D*) *b* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
    by *blast*
  then have *ill-deduct-wf* (*LimpR G a D b P*) **and** *ill-deduct-premises* (*LimpR G a D b P*) = []
    by *simp-all*
  then show *?case*
    by (*meson ill-conclusion.simps*(*10*))
next
  case (*withL1 G a D c b*)
  then obtain *P* :: (*'a*, *'b*) *ill-deduct*
    where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
    by *blast*

then have *ill-deduct-wf* (*WithL1 G a b D c P*) **and** *ill-deduct-premises* (*WithL1 G a b D c P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*11*))
next
case (*withL2 G b D c a*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [*b*] @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*WithL2 G a b D c P*) **and** *ill-deduct-premises* (*WithL2 G a b D c P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*12*))
next
case (*withR G a b*)
then obtain *P Q* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent G a* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
and *ill-conclusion Q* = *Sequent G b* ∧ *ill-deduct-wf Q* ∧ *ill-deduct-premises Q* = []
by *blast*
then have *ill-deduct-wf* (*WithR G a b P Q*) **and** *ill-deduct-premises* (*WithR G a b P Q*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*13*))
next
case (*topR G*)
then show *?case*
using *ill-conclusion.simps*(*14*) **by** *fastforce*
next
case (*plusL G a D c b*)
then obtain *P Q* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ *D*) *c* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
and *ill-conclusion Q* = *Sequent* (*G* @ [*b*] @ *D*) *c* ∧ *ill-deduct-wf Q* ∧ *ill-deduct-premises Q* = []
by *blast*
then have *ill-deduct-wf* (*PlusL G a b D c P Q*) **and** *ill-deduct-premises* (*PlusL G a b D c P Q*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*15*))
next
case (*plusR1 G a b*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*

where *ill-conclusion P* = *Sequent G a* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*PlusR1 G a b P*) **and** *ill-deduct-premises* (*PlusR1 G a b P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*16*))
next
case (*plusR2 G b a*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent G b* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*PlusR2 G a b P*) **and** *ill-deduct-premises* (*PlusR2 G a b P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*17*))
next
case (*zeroL G D c*)
then show *?case*
using *ill-conclusion.simps*(*18*) **by** *fastforce*
next
case (*weaken G D b a*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ *D*) *b* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*Weaken G D b a P*) **and** *ill-deduct-premises* (*Weaken G D b a P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*19*))
next
case (*contract G a D b*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [! *a*] @ [! *a*] @ *D*) *b* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
by *blast*
then have *ill-deduct-wf* (*Contract G a D b P*) **and** *ill-deduct-premises* (*Contract G a D b P*) = []
by *simp-all*
then show *?case*
by (*meson ill-conclusion.simps*(*20*))
next
case (*derelict G a D b*)
then obtain *P* :: (*'a*, *'b*) *ill-deduct*
where *ill-conclusion P* = *Sequent* (*G* @ [*a*] @ *D*) *b* ∧ *ill-deduct-wf P* ∧

ill-deduct-premises P = []
  by *blast*
 then have *ill-deduct-wf* (*Derelict G a D b P*) **and** *ill-deduct-premises* (*Derelict G a D b P*) = []
  by *simp-all*
 then show *?case*
  by (*meson ill-conclusion.simps*(*21*))
next
 case (*promote G a*)
 then obtain *P* :: (*'a*, *'b*) *ill-deduct*
   where *ill-conclusion P* = *Sequent* (*map Exp G*) *a* ∧ *ill-deduct-wf P* ∧ *ill-deduct-premises P* = []
  by *blast*
 then have *ill-deduct-wf* (*Promote G a P*) **and** *ill-deduct-premises* (*Promote G a P*) = []
  by *simp-all*
 then show *?case*
  by (*meson ill-conclusion.simps*(*22*))
qed

1.8.4 Derived Deductions

We define a number of useful deduction patterns as (potentially recursive) functions. In each case we verify the well-formedness, conclusion and premises.

Swap order in a times proposition: $[a \otimes b] \vdash b \otimes a$:

fun *ill-deduct-swap* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*) *ill-deduct*
 where *ill-deduct-swap a b* =
  TimesL [] *a b* [] (*b* ⊗ *a*)
  (*Exchange* [] *b a* [] (*b* ⊗ *a*)
   (*TimesR* [*b*] *b* [*a*] *a* (*Identity b*) (*Identity a*)))

lemma *ill-deduct-swap* [*simp*]:
 ill-deduct-wf (*ill-deduct-swap a b*)
 ill-conclusion (*ill-deduct-swap a b*) = *Sequent* [*a* ⊗ *b*] (*b* ⊗ *a*)
 ill-deduct-premises (*ill-deduct-swap a b*) = []
 by *simp-all*

Simplified cut rule: $[\![G \vdash b; [b] \vdash c]\!] \Longrightarrow G \vdash c$:

fun *ill-deduct-simple-cut* :: (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct*
 where *ill-deduct-simple-cut P Q* = *Cut* (*antecedents P*) (*consequent P*) [] [] (*consequent Q*) *P Q*

lemma *ill-deduct-simple-cut* [*simp*]:
 ⟦[*consequent P*] = *antecedents Q*; *ill-deduct-wf P*; *ill-deduct-wf Q*⟧ ⟹
  ill-deduct-wf (*ill-deduct-simple-cut P Q*)
 [*consequent P*] = *antecedents Q* ⟹

ill-conclusion (*ill-deduct-simple-cut P Q*) = *Sequent* (*antecedents P*) (*consequent Q*)
ill-deduct-premises (*ill-deduct-simple-cut P Q*) = *ill-deduct-premises P* @ *ill-deduct-premises Q*
by *simp-all blast*

Combine two deductions with times: ⟦[a] ⊢ b; [c] ⊢ d⟧ ⟹ [a ⊗ c] ⊢ b ⊗ d:

fun *ill-deduct-tensor* :: ('a, 'l) *ill-deduct* ⇒ ('a, 'l) *ill-deduct* ⇒ ('a, 'l) *ill-deduct*
where *ill-deduct-tensor p q* =
TimesL [] (*hd* (*antecedents p*)) (*hd* (*antecedents q*)) [] (*consequent p* ⊗ *consequent q*)
(*TimesR* (*antecedents p*) (*consequent p*) (*antecedents q*) (*consequent q*) *p q*)

lemma *ill-deduct-tensor* [*simp*]:
⟦*antecedents P* = [a]; *antecedents Q* = [c]; *ill-deduct-wf P*; *ill-deduct-wf Q*⟧ ⟹
ill-deduct-wf (*ill-deduct-tensor P Q*)
⟦*antecedents P* = [a]; *antecedents Q* = [c]⟧ ⟹
ill-conclusion (*ill-deduct-tensor P Q*) = *Sequent* [a ⊗ c] (*consequent P* ⊗ *consequent Q*)
ill-deduct-premises (*ill-deduct-tensor P Q*) = *ill-deduct-premises P* @ *ill-deduct-premises Q*
by *simp-all blast*

Associate times proposition to right: [(a ⊗ b) ⊗ c] ⊢ a ⊗ b ⊗ c:

fun *ill-deduct-assoc* :: 'a *ill-prop* ⇒ 'a *ill-prop* ⇒ 'a *ill-prop* ⇒ ('a, 'l) *ill-deduct*
where *ill-deduct-assoc a b c* =
TimesL [] (a ⊗ b) c [] (a ⊗ (b ⊗ c))
(*Exchange* [] c (a ⊗ b) [] (a ⊗ (b ⊗ c))
(*TimesL* [c] a b [] (a ⊗ (b ⊗ c))
(*Exchange* [] a c [b] (a ⊗ (b ⊗ c))
(*TimesR* [a] a [c, b] (b ⊗ c)
(*Identity* a)
(*Exchange* [] b c [] (b ⊗ c)
(*TimesR* [b] b [c] c
(*Identity* b)
(*Identity* c)))))))

lemma *ill-deduct-assoc* [*simp*]:
ill-deduct-wf (*ill-deduct-assoc a b c*)
ill-conclusion (*ill-deduct-assoc a b c*) = *Sequent* [(a ⊗ b) ⊗ c] (a ⊗ (b ⊗ c))
ill-deduct-premises (*ill-deduct-assoc a b c*) = []
by *simp-all*

Associate times proposition to left: [a ⊗ b ⊗ c] ⊢ (a ⊗ b) ⊗ c:

fun *ill-deduct-assoc'* :: 'a *ill-prop* ⇒ 'a *ill-prop* ⇒ 'a *ill-prop* ⇒ ('a, 'l) *ill-deduct*
where *ill-deduct-assoc' a b c* =
TimesL [] a (b ⊗ c) [] ((a ⊗ b) ⊗ c)
(*TimesL* [a] b c [] ((a ⊗ b) ⊗ c)
(*TimesR* [a, b] (a ⊗ b) [c] c

(*TimesR* [*a*] *a* [*b*] *b*
  (*Identity a*)
  (*Identity b*))
(*Identity c*)))

lemma *ill-deduct-assoc′* [*simp*]:
ill-deduct-wf (*ill-deduct-assoc′ a b c*)
ill-conclusion (*ill-deduct-assoc′ a b c*) = *Sequent* [$a \otimes (b \otimes c)$] (($a \otimes b) \otimes c$)
ill-deduct-premises (*ill-deduct-assoc′ a b c*) = []
by *simp-all*

Eliminate times unit a proposition: $[a \otimes \mathbf{1}] \vdash a$:

fun *ill-deduct-unit* :: *′a ill-prop* ⇒ (*′a*, *′l*) *ill-deduct*
where *ill-deduct-unit a* = *TimesL* [] *a* (**1**) [] *a* (*OneL* [*a*] [] *a* (*Identity a*))

lemma *ill-deduct-unit* [*simp*]:
ill-deduct-wf (*ill-deduct-unit a*)
ill-conclusion (*ill-deduct-unit a*) = *Sequent* [$a \otimes \mathbf{1}$] *a*
ill-deduct-premises (*ill-deduct-unit a*) = []
by *simp-all*

Introduce times unit into a proposition $[a] \vdash a \otimes \mathbf{1}$:

fun *ill-deduct-unit′* :: *′a ill-prop* ⇒ (*′a*, *′l*) *ill-deduct*
where *ill-deduct-unit′ a* = *TimesR* [*a*] *a* [] (**1**) (*Identity a*) *OneR*

lemma *ill-deduct-unit′* [*simp*]:
ill-deduct-wf (*ill-deduct-unit′ a*)
ill-conclusion (*ill-deduct-unit′ a*) = *Sequent* [*a*] ($a \otimes \mathbf{1}$)
ill-deduct-premises (*ill-deduct-unit′ a*) = []
by *simp-all*

Simplified weakening: $[!\ a] \vdash \mathbf{1}$:

fun *ill-deduct-simple-weaken* :: *′a ill-prop* ⇒ (*′a*, *′l*) *ill-deduct*
where *ill-deduct-simple-weaken a* = *Weaken* [] [] (**1**) *a OneR*

lemma *ill-deduct-simple-weaken* [*simp*]:
ill-deduct-wf (*ill-deduct-simple-weaken a*)
ill-conclusion (*ill-deduct-simple-weaken a*) = *Sequent* [!*a*] **1**
ill-deduct-premises (*ill-deduct-simple-weaken a*) = []
by *simp-all*

Simplified dereliction: $[!\ a] \vdash a$:

fun *ill-deduct-dereliction* :: *′a ill-prop* ⇒ (*′a*, *′l*) *ill-deduct*
where *ill-deduct-dereliction a* = *Derelict* [] *a* [] *a* (*Identity a*)

lemma *ill-deduct-dereliction* [*simp*]:
ill-deduct-wf (*ill-deduct-dereliction a*)

ill-conclusion (*ill-deduct-dereliction* *a*) = *Sequent* [!*a*] *a*
ill-deduct-premises (*ill-deduct-dereliction* *a*) = []
by *simp-all*

Duplicate exponentiated proposition: $[!\ a] \vdash !\ a \otimes !\ a$:

fun *ill-deduct-duplicate* :: *'a ill-prop* ⇒ (*'a*, *'l*) *ill-deduct*
where *ill-deduct-duplicate a* =
Contract [] *a* [] (!*a* ⊗ !*a*) (*TimesR* [!*a*] (!*a*) [!*a*] (!*a*) (*Identity* (!*a*)) (*Identity* (!*a*)))

lemma *ill-deduct-duplicate* [*simp*]:
ill-deduct-wf (*ill-deduct-duplicate* *a*)
ill-conclusion (*ill-deduct-duplicate* *a*) = *Sequent* [!*a*] (!*a* ⊗ !*a*)
ill-deduct-premises (*ill-deduct-duplicate* *a*) = []
by *simp-all*

Simplified plus elimination: $[\![[a] \vdash c; [b] \vdash c]\!] \Longrightarrow [a \oplus b] \vdash c$:

fun *ill-deduct-simple-plusL* :: (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct*
where *ill-deduct-simple-plusL p q* =
PlusL [] (*hd* (*antecedents p*)) (*hd* (*antecedents q*)) [] (*consequent p*) *p q*

lemma *ill-deduct-simple-plusL* [*simp*]:
⟦ *antecedents P* = [*a*]; *antecedents Q* = [*b*]; *ill-deduct-wf P*
; *ill-deduct-wf Q*; *consequent P* = *consequent Q*⟧ ⟹
ill-deduct-wf (*ill-deduct-simple-plusL P Q*)
⟦*antecedents P* = [*a*]; *antecedents Q* = [*b*]⟧ ⟹
ill-conclusion (*ill-deduct-simple-plusL P Q*) = *Sequent* [*a* ⊕ *b*] (*consequent P*)
ill-deduct-premises (*ill-deduct-simple-plusL P Q*)
= *ill-deduct-premises P* @ *ill-deduct-premises Q*
by *simp-all blast*

Simplified left plus introduction: $[a] \vdash a \oplus b$:

fun *ill-deduct-plusR1* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*) *ill-deduct*
where *ill-deduct-plusR1 a b* = *PlusR1* [*a*] *a b* (*Identity a*)

lemma *ill-deduct-plusR1* [*simp*]:
ill-deduct-wf (*ill-deduct-plusR1 a b*)
ill-conclusion (*ill-deduct-plusR1 a b*) = *Sequent* [*a*] (*a* ⊕ *b*)
ill-deduct-premises (*ill-deduct-plusR1 a b*) = []
by *simp-all*

Simplified right plus introduction: $[b] \vdash a \oplus b$:

fun *ill-deduct-plusR2* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*) *ill-deduct*
where *ill-deduct-plusR2 a b* = *PlusR2* [*b*] *a b* (*Identity b*)

lemma *ill-deduct-plusR2* [*simp*]:
ill-deduct-wf (*ill-deduct-plusR2 a b*)

ill-conclusion (*ill-deduct-plusR2 a b*) = *Sequent* [*b*] (*a* ⊕ *b*)
ill-deduct-premises (*ill-deduct-plusR2 a b*) = []
by *simp-all*

Simplified linear implication introduction: $[a] \vdash b \Longrightarrow [\mathbf{1}] \vdash a \rhd b$:

fun *ill-deduct-simple-limpR* :: ('*a*, '*l*) *ill-deduct* ⇒ ('*a*, '*l*) *ill-deduct*
where *ill-deduct-simple-limpR p* =
LimpR [] (*hd* (*antecedents p*)) [**1**] (*consequent p*)
(*OneL* [*hd* (*antecedents p*)] [] (*consequent p*) *p*)

lemma *ill-deduct-simple-limpR* [*simp*]:
⟦*antecedents P* = [*a*]; *consequent P* = *b*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-simple-limpR P*)
⟦*antecedents P* = [*a*]; *consequent P* = *b*⟧ ⟹
ill-conclusion (*ill-deduct-simple-limpR P*) = *Sequent* [**1**] (*a* ▷ *b*)
ill-deduct-premises (*ill-deduct-simple-limpR P*)
= *ill-deduct-premises P*
by *simp-all blast*

Simplified introduction of exponentiated impliciation: $[a] \vdash b \Longrightarrow [\mathbf{1}] \vdash !\ (a \rhd b)$:

fun *ill-deduct-simple-limpR-exp* :: ('*a*, '*l*) *ill-deduct* ⇒ ('*a*, '*l*) *ill-deduct*
where *ill-deduct-simple-limpR-exp p* =
OneL [] [] (!((*hd* (*antecedents p*)) ▷ (*consequent p*)))
(*Promote* [] ((*hd* (*antecedents p*)) ▷ (*consequent p*))
(*ill-deduct-simple-cut*
OneR
(*ill-deduct-simple-limpR p*)))

lemma *ill-deduct-simple-limpR-exp* [*simp*]:
⟦*antecedents P* = [*a*]; *consequent P* = *b*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-simple-limpR-exp P*)
⟦*antecedents P* = [*a*]; *consequent P* = *b*⟧ ⟹
ill-conclusion (*ill-deduct-simple-limpR-exp P*) = *Sequent* [**1**] (!(*a* ▷ *b*))
ill-deduct-premises (*ill-deduct-simple-limpR-exp P*) = *ill-deduct-premises P*
by *simp-all blast*

Linear implication elimination with times: $[a \otimes a \rhd b] \vdash b$:

fun *ill-deduct-limp-eval* :: '*a ill-prop* ⇒ '*a ill-prop* ⇒ ('*a*, '*l*) *ill-deduct*
where *ill-deduct-limp-eval a b* =
TimesL [] *a* (*a* ▷ *b*) [] *b* (*LimpL* [*a*] *a* [] *b* [] *b* (*Identity a*) (*Identity b*))

lemma *ill-deduct-limp-eval* [*simp*]:
ill-deduct-wf (*ill-deduct-limp-eval a b*)
ill-conclusion (*ill-deduct-limp-eval a b*) = *Sequent* [*a* ⊗ *a* ▷ *b*] *b*
ill-deduct-premises (*ill-deduct-limp-eval a b*) = []
by *simp-all*

Exponential implication elimination with times: $[a \otimes !\ (a \rhd b)] \vdash b \otimes !\ (a$

$\rhd$ b):

fun *ill-deduct-explimp-eval* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*) *ill-deduct*
  where *ill-deduct-explimp-eval a b* =
    TimesL [] *a* (!(*a* ▷ *b*)) [] (*b* ⊗ !(*a* ▷ *b*)) (
    Contract [*a*] (*a* ▷ *b*) [] (*b* ⊗ !(*a* ▷ *b*)) (
    TimesR [*a*, !(*a* ▷ *b*)] *b* [!(*a* ▷ *b*)] (!(*a* ▷ *b*))
    (*Derelict* [*a*] (*a* ▷ *b*) [] *b* (
      LimpL [*a*] *a* [] *b* [] *b*
      (*Identity a*)
      (*Identity b*)))
    (*Identity* (!(*a* ▷ *b*)))))

lemma *ill-deduct-explimp-eval* [*simp*]:
  ill-deduct-wf (*ill-deduct-explimp-eval a b*)
  ill-conclusion (*ill-deduct-explimp-eval a b*) = *Sequent* [*a* ⊗ !(*a* ▷ *b*)] (*b* ⊗ !(*a* ▷
b))
  ill-deduct-premises (*ill-deduct-explimp-eval a b*) = []
  by *simp-all*

Distributing times over plus: $[a \otimes b \oplus c] \vdash (a \otimes b) \oplus a \otimes c$:

fun *ill-deduct-distrib-plus* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*)
ill-deduct
  where *ill-deduct-distrib-plus a b c* =
  TimesL [] *a* (*b* ⊕ *c*) [] ((*a* ⊗ *b*) ⊕ (*a* ⊗ *c*))
  (*PlusL* [*a*] *b c* [] ((*a* ⊗ *b*) ⊕ (*a* ⊗ *c*))
    (*PlusR1* [*a*, *b*] (*a* ⊗ *b*) (*a* ⊗ *c*)
      (*TimesR* [*a*] *a* [*b*] *b*
        (*Identity a*)
        (*Identity b*)))
    (*PlusR2* [*a*, *c*] (*a* ⊗ *b*) (*a* ⊗ *c*)
      (*TimesR* [*a*] *a* [*c*] *c*
        (*Identity a*)
        (*Identity c*))))

lemma *ill-deduct-distrib-plus* [*simp*]:
  ill-deduct-wf (*ill-deduct-distrib-plus a b c*)
  ill-conclusion (*ill-deduct-distrib-plus a b c*) = *Sequent* [*a* ⊗ (*b* ⊕ *c*)] ((*a* ⊗ *b*) ⊕
(*a* ⊗ *c*))
  ill-deduct-premises (*ill-deduct-distrib-plus a b c*) = []
  by *simp-all*

Distributing times out of plus: $[(a \otimes b) \oplus a \otimes c] \vdash a \otimes b \oplus c$:

fun *ill-deduct-distrib-plus'* :: *'a ill-prop* ⇒ *'a ill-prop* ⇒ *'a ill-prop* ⇒ (*'a*, *'l*)
ill-deduct
  where *ill-deduct-distrib-plus' a b c* =
  PlusL [] (*a* ⊗ *b*) (*a* ⊗ *c*) [] (*a* ⊗ (*b* ⊕ *c*))
  (*ill-deduct-tensor*
    (*Identity a*)
    (*ill-deduct-plusR1 b c*))

(*ill-deduct-tensor*
(*Identity a*)
(*ill-deduct-plusR2 b c*))

lemma *ill-deduct-distrib-plus′* [*simp*]:
ill-deduct-wf (*ill-deduct-distrib-plus′ a b c*)
ill-conclusion (*ill-deduct-distrib-plus′ a b c*) = *Sequent* [($a \otimes b$) $\oplus$ ($a \otimes c$)] ($a \otimes$ ($b \oplus c$))
ill-deduct-premises (*ill-deduct-distrib-plus′ a b c*) = []
by *simp-all*

Combining two deductions with plus: $[\![[a] \vdash b; [c] \vdash d]\!] \Longrightarrow [a \oplus c] \vdash b \oplus d$:

fun *ill-deduct-plus-progress* :: ($'a$, $'l$) *ill-deduct* $\Rightarrow$ ($'a$, $'l$) *ill-deduct* $\Rightarrow$ ($'a$, $'l$) *ill-deduct*
where *ill-deduct-plus-progress p q* =
ill-deduct-simple-plusL
(*ill-deduct-simple-cut p* (*ill-deduct-plusR1* (*consequent p*) (*consequent q*)))
(*ill-deduct-simple-cut q* (*ill-deduct-plusR2* (*consequent p*) (*consequent q*)))

lemma *ill-deduct-plus-progress* [*simp*]:
$[\![$*antecedents P* = [*a*]; *antecedents Q* = [*c*]; *ill-deduct-wf P*; *ill-deduct-wf Q*$]\!] \Longrightarrow$
ill-deduct-wf (*ill-deduct-plus-progress P Q*)
$[\![$*antecedents P* = [*a*]; *antecedents Q* = [*c*]$]\!] \Longrightarrow$
ill-conclusion (*ill-deduct-plus-progress P Q*) = *Sequent* [$a \oplus c$] (*consequent P* $\oplus$ *consequent Q*)
ill-deduct-premises (*ill-deduct-plus-progress P Q*)
= *ill-deduct-premises P* @ *ill-deduct-premises Q*
by *simp-all blast*

Simplified with introduction: $[\![[a] \vdash b; [a] \vdash c]\!] \Longrightarrow [a] \vdash b \,\&\, c$:

fun *ill-deduct-with* :: ($'a$, $'l$) *ill-deduct* $\Rightarrow$ ($'a$, $'l$) *ill-deduct* $\Rightarrow$ ($'a$, $'l$) *ill-deduct*
where *ill-deduct-with p q* = *WithR* [*hd* (*antecedents p*)] (*consequent p*) (*consequent q*) *p q*

lemma *ill-deduct-with* [*simp*]:
$[\![$ *antecedents P* = [*a*]; *antecedents Q* = [*a*]; *consequent P* = *b*
; *consequent Q* = *c*; *ill-deduct-wf P*; *ill-deduct-wf Q*$]\!] \Longrightarrow$
ill-deduct-wf (*ill-deduct-with P Q*)
$[\![$*antecedents P* = [*a*]; *antecedents Q* = [*a*]; *consequent P* = *b*; *consequent Q* = *c*$]\!]$ $\Longrightarrow$
ill-conclusion (*ill-deduct-with P Q*) = *Sequent* [*a*] (*consequent P* & *consequent Q*)
ill-deduct-premises (*ill-deduct-with P Q*) = *ill-deduct-premises P* @ *ill-deduct-premises Q*
by *simp-all blast*

Simplified with left projection: $[a \,\&\, b] \vdash a$:

fun *ill-deduct-projectL* :: $'a$ *ill-prop* $\Rightarrow$ $'a$ *ill-prop* $\Rightarrow$ ($'a$, $'l$) *ill-deduct*
where *ill-deduct-projectL a b* = *WithL1* [] *a b* [] *a* (*Identity a*)

lemma *ill-deduct-projectL* [*simp*]:
ill-deduct-wf (*ill-deduct-projectL a b*)
ill-conclusion (*ill-deduct-projectL a b*) = *Sequent* [*a* & *b*] *a*
ill-deduct-premises (*ill-deduct-projectL a b*) = []
by *simp-all*

Simplified with right projection: $[a \ \& \ b] \vdash b$:

fun *ill-deduct-projectR* :: ′*a ill-prop* ⇒ ′*a ill-prop* ⇒ (′*a*, ′*l*) *ill-deduct*
where *ill-deduct-projectR a b* = *WithL2* [] *a b* [] *b* (*Identity b*)

lemma *ill-deduct-projectR* [*simp*]:
ill-deduct-wf (*ill-deduct-projectR a b*)
ill-conclusion (*ill-deduct-projectR a b*) = *Sequent* [*a* & *b*] *b*
ill-deduct-premises (*ill-deduct-projectR a b*) = []
by *simp-all*

Distributing times over with: $[a \otimes b \ \& \ c] \vdash (a \otimes b) \ \& \ a \otimes c$:

fun *ill-deduct-distrib-with* :: ′*a ill-prop* ⇒ ′*a ill-prop* ⇒ ′*a ill-prop* ⇒ (′*a*, ′*l*) *ill-deduct*
where *ill-deduct-distrib-with a b c* =
WithR [*a* ⊗ (*b* & *c*)] (*a* ⊗ *b*) (*a* ⊗ *c*)
(*ill-deduct-tensor*
(*Identity a*)
(*ill-deduct-projectL b c*))
(*ill-deduct-tensor*
(*Identity a*)
(*ill-deduct-projectR b c*))

lemma *ill-deduct-distrib-with* [*simp*]:
ill-deduct-wf (*ill-deduct-distrib-with a b c*)
ill-conclusion (*ill-deduct-distrib-with a b c*) = *Sequent* [*a* ⊗ (*b* & *c*)] ((*a* ⊗ *b*) & (*a* ⊗ *c*))
ill-deduct-premises (*ill-deduct-distrib-with a b c*) = []
by *simp-all*

Weakening a list of propositions: $G \ @ \ D \vdash b \Longrightarrow G \ @ \ map \ ! \ xs \ @ \ D \vdash b$:

fun *ill-deduct-weaken-list*
:: ′*a ill-prop list* ⇒ ′*a ill-prop list* ⇒ ′*a ill-prop list* ⇒ (′*a*, ′*l*) *ill-deduct*
⇒ (′*a*, ′*l*) *ill-deduct*
where
ill-deduct-weaken-list G D [] *P* = *P*
| *ill-deduct-weaken-list G D* (*x*#*xs*) *P* =
Weaken G (*map Exp xs* @ *D*) (*consequent P*) *x* (*ill-deduct-weaken-list G D xs P*)

lemma *ill-deduct-weaken-list* [*simp*]:
⟦*antecedents P* = *G* @ *D*; *ill-deduct-wf P*⟧ ⟹ *ill-deduct-wf* (*ill-deduct-weaken-list G D xs P*)

antecedents P = *G* @ *D* ∨ *xs* ≠ [] ⟹
antecedents (*ill-deduct-weaken-list G D xs P*) = *G* @ (*map Exp xs*) @ *D*
consequent (*ill-deduct-weaken-list G D xs P*) = *consequent P*
ill-deduct-premises (*ill-deduct-weaken-list G D xs P*) = *ill-deduct-premises P*
proof –
have [*simp*]: *antecedents* (*ill-deduct-weaken-list G D xs P*) = *G* @ (*map Exp xs*) @ *D*
if *antecedents P* = *G* @ *D* ∨ *xs* ≠ []
for *G D* :: *'c ill-prop list* **and** *xs* :: *'c ill-prop list* **and** *P* :: (*'c*, *'d*) *ill-deduct*
using *that* **by** (*induct xs*) *simp-all*
then show *antecedents P* = *G* @ *D* ∨ *xs* ≠ [] ⟹
antecedents (*ill-deduct-weaken-list G D xs P*) = *G* @ (*map Exp xs*) @ *D* **.**

have [*simp*]: *consequent* (*ill-deduct-weaken-list G D xs P*) = *consequent P*
for *G D* :: *'c ill-prop list* **and** *xs* **and** *P* :: (*'c*, *'d*) *ill-deduct*
by (*induct xs*) *simp-all*
then show *consequent* (*ill-deduct-weaken-list G D xs P*) = *consequent P* **.**

show ⟦*antecedents P* = *G* @ *D*; *ill-deduct-wf P*⟧ ⟹ *ill-deduct-wf* (*ill-deduct-weaken-list G D xs P*)
by (*induct xs*) (*simp-all add*: *ill-conclusion-alt*)

show *ill-deduct-premises* (*ill-deduct-weaken-list G D xs P*) = *ill-deduct-premises P*
by (*induct xs*) *simp-all*
qed

Exponentiating a deduction: *G* ⊢ *b* ⟹ *map* ! *G* ⊢ ! *b*

fun *ill-deduct-exp-helper* :: *nat* ⇒ (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct*
— Helper function to apply *Derelict* to first *n* antecedents
where
ill-deduct-exp-helper 0 P = *P*
| *ill-deduct-exp-helper* (*Suc n*) *P* =
Derelict
(*map Exp* (*take n* (*antecedents P*)))
(*nth* (*antecedents P*) *n*)
(*drop* (*Suc n*) (*antecedents P*))
(*consequent P*)
(*ill-deduct-exp-helper n P*)

lemma *ill-deduct-exp-helper*:
n ≤ *length* (*antecedents P*) ⟹
antecedents (*ill-deduct-exp-helper n P*)
= *map Exp* (*take n* (*antecedents P*)) @ *drop n* (*antecedents P*)
consequent (*ill-deduct-exp-helper n P*) = *consequent P*
n ≤ *length* (*antecedents P*) ⟹ *ill-deduct-wf* (*ill-deduct-exp-helper n P*) = *ill-deduct-wf P*
ill-deduct-premises (*ill-deduct-exp-helper n P*) = *ill-deduct-premises P*
proof –

have [*simp*]:
antecedents (*ill-deduct-exp-helper n P*)
= *map Exp* (*take n* (*antecedents P*)) @ *drop n* (*antecedents P*)
if $n \leq$ *length* (*antecedents P*) **for** *n*
using *that* **by** (*induct n*) (*simp-all add*: *take-Suc-conv-app-nth*)
then show $n \leq$ *length* (*antecedents P*) $\Longrightarrow$
antecedents (*ill-deduct-exp-helper n P*)
= *map Exp* (*take n* (*antecedents P*)) @ *drop n* (*antecedents P*) **.**

have [*simp*]: *consequent* (*ill-deduct-exp-helper n P*) = *consequent P* **for** *n*
by (*induct n*) *simp-all*
then show *consequent* (*ill-deduct-exp-helper n P*) = *consequent P* **.**

show $n \leq$ *length* (*antecedents P*) $\Longrightarrow$ *ill-deduct-wf* (*ill-deduct-exp-helper n P*) = *ill-deduct-wf P*
by (*induct n*) (*simp-all add*: *ill-conclusion-alt Cons-nth-drop-Suc*)

show *ill-deduct-premises* (*ill-deduct-exp-helper n P*) = *ill-deduct-premises P*
by (*induct n*) *simp-all*
qed

fun *ill-deduct-exp* :: ('*a*, '*l*) *ill-deduct* ⇒ ('*a*, '*l*) *ill-deduct*
where *ill-deduct-exp P* =
Promote (*antecedents P*) (*consequent P*) (*ill-deduct-exp-helper* (*length* (*antecedents P*)) *P*)

lemma *ill-deduct-exp* [*simp*]:
ill-conclusion (*ill-deduct-exp P*) = *Sequent* (*map Exp* (*antecedents P*)) (!(*consequent P*))
ill-deduct-wf (*ill-deduct-exp P*) = *ill-deduct-wf P*
ill-deduct-premises (*ill-deduct-exp P*) = *ill-deduct-premises P*
by (*simp-all add*: *ill-conclusion-alt ill-deduct-exp-helper*)

1.8.5 Compacting Equivalences

Compacting cons equivalence: $a \otimes$ *compact b* ⊣⊢ *compact* (*a* # *b*):

primrec *ill-deduct-times-to-compact-cons* :: '*a ill-prop* ⇒ '*a ill-prop list* ⇒ ('*a*, '*l*) *ill-deduct*
— [$a \otimes$ *compact b*] ⊢ *compact* (*a* # *b*)
where
ill-deduct-times-to-compact-cons a [] = *ill-deduct-unit a*
| *ill-deduct-times-to-compact-cons a* (*b*#*bs*) = *Identity* ($a \otimes$ *compact* (*b*#*bs*))

lemma *ill-deduct-times-to-compact-cons* [*simp*]:
ill-deduct-wf (*ill-deduct-times-to-compact-cons a b*)
ill-conclusion (*ill-deduct-times-to-compact-cons a b*)
= *Sequent* [$a \otimes$ *compact b*] (*compact* (*a* # *b*))
ill-deduct-premises (*ill-deduct-times-to-compact-cons a b*) = []
by (*cases b*, *simp-all*)+

primrec *ill-deduct-compact-cons-to-times* :: *'a ill-prop* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
— [*compact* (*a* # *b*)] ⊢ *a* ⊗ *compact b*
where
ill-deduct-compact-cons-to-times a [] = *ill-deduct-unit' a*
| *ill-deduct-compact-cons-to-times a* (*b*#*bs*) = *Identity* (*a* ⊗ *compact* (*b*#*bs*))

lemma *ill-deduct-compact-cons-to-times* [*simp*]:
ill-deduct-wf (*ill-deduct-compact-cons-to-times a b*)
ill-conclusion (*ill-deduct-compact-cons-to-times a b*)
= *Sequent* [*compact* (*a* # *b*)] (*a* ⊗ *compact b*)
ill-deduct-premises (*ill-deduct-compact-cons-to-times a b*) = []
by (*cases b*, *simp*, *simp*)+

Compacting append equivalence: *compact a* ⊗ *compact b* ⊣⊢ *compact* (*a* @ *b*):

primrec *ill-deduct-times-to-compact-append*
:: *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
— [*compact a* ⊗ *compact b*] ⊢ *compact* (*a* @ *b*)
where
ill-deduct-times-to-compact-append [] *b* =
ill-deduct-simple-cut (*ill-deduct-swap* (**1**) (*compact b*)) (*ill-deduct-unit* (*compact b*))
| *ill-deduct-times-to-compact-append* (*a*#*as*) *b* =
ill-deduct-simple-cut
(*ill-deduct-simple-cut*
(*ill-deduct-simple-cut*
(*ill-deduct-tensor*
(*ill-deduct-compact-cons-to-times a as*)
(*Identity* (*compact b*)))
(*ill-deduct-assoc a* (*compact as*) (*compact b*)))
(*ill-deduct-tensor*
(*Identity a*)
(*ill-deduct-times-to-compact-append as b*)))
(*ill-deduct-times-to-compact-cons a* (*as* @ *b*))

lemma *ill-deduct-times-to-compact-append* [*simp*]:
ill-deduct-wf (*ill-deduct-times-to-compact-append a b* :: (*'a*, *'l*) *ill-deduct*)
ill-conclusion (*ill-deduct-times-to-compact-append a b* :: (*'a*, *'l*) *ill-deduct*)
= *Sequent* [*compact a* ⊗ *compact b*] (*compact* (*a* @ *b*))
ill-deduct-premises (*ill-deduct-times-to-compact-append a b*) = []
by (*induct a*) (*simp-all add*: *ill-conclusion-antecedents ill-conclusion-consequent*)

primrec *ill-deduct-compact-append-to-times*
:: *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
— [*compact* (*a* @ *b*)] ⊢ *compact a* ⊗ *compact b*
where
ill-deduct-compact-append-to-times [] *b* =

ill-deduct-simple-cut
(*ill-deduct-unit′* (*compact b*))
(*ill-deduct-swap* (*compact b*) (**1**))
| *ill-deduct-compact-append-to-times* (*a#as*) *b* =
ill-deduct-simple-cut
(*ill-deduct-compact-cons-to-times a* (*as* @ *b*))
(*ill-deduct-simple-cut*
(*ill-deduct-tensor*
(*Identity a*)
(*ill-deduct-compact-append-to-times as b*))
(*ill-deduct-simple-cut*
(*ill-deduct-assoc′ a* (*compact as*) (*compact b*))
(*ill-deduct-tensor*
(*ill-deduct-times-to-compact-cons a as*)
(*Identity* (*compact b*)))))

lemma *ill-deduct-compact-append-to-times* [*simp*]:
ill-deduct-wf (*ill-deduct-compact-append-to-times a b* :: (′*a*, ′*l*) *ill-deduct*)
ill-conclusion (*ill-deduct-compact-append-to-times a b* :: (′*a*, ′*l*) *ill-deduct*)
= *Sequent* [*compact* (*a* @ *b*)] (*compact a* ⊗ *compact b*)
ill-deduct-premises (*ill-deduct-compact-append-to-times a b*) = []
by (*induct a*) (*simp-all add*: *ill-conclusion-antecedents ill-conclusion-consequent*)

Combine a list of deductions with times using *ill-deduct-tensor*, representing a generalised version of the following theorem of the shallow embedding: ∀ *x*∈*set ?xs*. [*?f x*] ⊢ *?g x* ⟹ [*compact* (*map ?f ?xs*)] ⊢ *compact* (*map ?g ?xs*)

primrec *ill-deduct-tensor-list* :: (′*a*, ′*l*) *ill-deduct list* ⇒ (′*a*, ′*l*) *ill-deduct*
where
ill-deduct-tensor-list [] = *Identity* (**1**)
| *ill-deduct-tensor-list* (*x#xs*) =
(*if xs* = [] *then x else ill-deduct-tensor x* (*ill-deduct-tensor-list xs*))

lemma *ill-deduct-tensor-list* [*simp*]:
fixes *xs* :: (′*a*, ′*l*) *ill-deduct list*
assumes ⋀*x*. *x* ∈ *set xs* ⟹ ∃ *a*. *antecedents x* = [*a*]
shows *ill-conclusion* (*ill-deduct-tensor-list xs*)
= *Sequent* [*compact* (*map* (*hd* ∘ *antecedents*) *xs*)] (*compact* (*map consequent xs*))
and (⋀*x*. *x* ∈ *set xs* ⟹ *ill-deduct-wf x*) ⟹ *ill-deduct-wf* (*ill-deduct-tensor-list xs*)
and *ill-deduct-premises* (*ill-deduct-tensor-list xs*) = *concat* (*map ill-deduct-premises xs*)
proof −
have *x* [*simp*]:
ill-conclusion (*ill-deduct-tensor-list xs*)
= *Sequent* [*compact* (*map* (*hd* ∘ *antecedents*) *xs*)] (*compact* (*map consequent xs*))
if ⋀*x*. *x* ∈ *set xs* ⟹ ∃ *a*. *antecedents x* = [*a*] **for** *xs* :: (′*a*, ′*l*) *ill-deduct list*

using *that*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
using *that* **by** (*simp add*: *ill-conclusion-antecedents ill-conclusion-consequent*) *fastforce*
qed
then show
ill-conclusion (*ill-deduct-tensor-list xs*)
= *Sequent* [*compact* (*map* (*hd* ∘ *antecedents*) *xs*)] (*compact* (*map consequent xs*))
using *assms* **.**

show ($\bigwedge$*x*. *x* ∈ *set xs* ⟹ *ill-deduct-wf x*) ⟹ *ill-deduct-wf* (*ill-deduct-tensor-list xs*)
using *assms*
by (*induct xs*) (*fastforce simp add*: *ill-conclusion-antecedents ill-conclusion-consequent*)+

show *ill-deduct-premises* (*ill-deduct-tensor-list xs*) = *concat* (*map ill-deduct-premises xs*)
using *assms* **by** (*induct xs*) *simp-all*
qed

1.8.6 Premise Substitution

Premise substitution replaces certain premises in a deduction with other deductions. The target premises are specified with a predicate on the three arguments of the *Premise* constructor: antecedents, consequent and label. The replacement for each is specified as a function of those three arguments. In this way, the substitution can replace a whole class of premises in a single pass.

primrec *ill-deduct-subst* ::
('*a ill-prop list* ⇒ '*a ill-prop* ⇒ '*l* ⇒ *bool*) ⇒
('*a ill-prop list* ⇒ '*a ill-prop* ⇒ '*l* ⇒ ('*a*, '*l*) *ill-deduct*) ⇒
('*a*, '*l*) *ill-deduct* ⇒ ('*a*, '*l*) *ill-deduct*
where
ill-deduct-subst p f (*Premise G c l*) = (*if p G c l then f G c l else Premise G c l*)
| *ill-deduct-subst p f* (*Identity a*) = *Identity a*
| *ill-deduct-subst p f* (*Exchange G a b D c P*) = *Exchange G a b D c* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*Cut G b D E c P Q*) =
Cut G b D E c (*ill-deduct-subst p f P*) (*ill-deduct-subst p f Q*)
| *ill-deduct-subst p f* (*TimesL G a b D c P*) = *TimesL G a b D c* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*TimesR G a D b P Q*) =

TimesR G a D b (*ill-deduct-subst p f P*) (*ill-deduct-subst p f Q*)
| *ill-deduct-subst p f* (*OneL G D c P*) = *OneL G D c* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*OneR*) = *OneR*
| *ill-deduct-subst p f* (*LimpL G a D b E c P Q*) =
LimpL G a D b E c (*ill-deduct-subst p f P*) (*ill-deduct-subst p f Q*)
| *ill-deduct-subst p f* (*LimpR G a D b P*) = *LimpR G a D b* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*WithL1 G a b D c P*) = *WithL1 G a b D c* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*WithL2 G a b D c P*) = *WithL2 G a b D c* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*WithR G a b P Q*) =
WithR G a b (*ill-deduct-subst p f P*) (*ill-deduct-subst p f Q*)
| *ill-deduct-subst p f* (*TopR G*) = *TopR G*
| *ill-deduct-subst p f* (*PlusL G a b D c P Q*) =
PlusL G a b D c (*ill-deduct-subst p f P*) (*ill-deduct-subst p f Q*)
| *ill-deduct-subst p f* (*PlusR1 G a b P*) = *PlusR1 G a b* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*PlusR2 G a b P*) = *PlusR2 G a b* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*ZeroL G D c*) = *ZeroL G D c*
| *ill-deduct-subst p f* (*Weaken G D b a P*) = *Weaken G D b a* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*Contract G a D b P*) = *Contract G a D b* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*Derelict G a D b P*) = *Derelict G a D b* (*ill-deduct-subst p f P*)
| *ill-deduct-subst p f* (*Promote G a P*) = *Promote G a* (*ill-deduct-subst p f P*)

If the target premise is not present, then substitution does nothing

lemma *ill-deduct-subst-no-target*:
(⋀*G c l.* (*G*, *c*, *l*) ∈ *set* (*ill-deduct-premises x*) ⟹ ¬ *p G c l*) ⟹ *ill-deduct-subst p f x* = *x*
by (*induct x*) *simp-all*

If a deduction has no premise, then substitution does nothing

lemma *ill-deduct-subst-no-prems*:
ill-deduct-premises x = [] ⟹ *ill-deduct-subst p f x* = *x*
using *ill-deduct-subst-no-target empty-set emptyE* **by** *metis*

If we substitute the target, then the substitution does nothing

lemma *ill-deduct-subst-of-target* [*simp*]:
f = *Premise* ⟹ *ill-deduct-subst p f x* = *x*
by (*induct x*) *simp-all*

Substitution matching the target's antecedents preserves overall deduction antecedents

lemma *ill-deduct-subst-antecedents* [*simp*]:
assumes (⋀*G c l. p G c l* ⟹ *antecedents* (*f G c l*) = *G*)
shows *antecedents* (*ill-deduct-subst p f x*) = *antecedents x*

using *assms* **by** (*induct x*) *simp-all*

Substitution matching the target's consequent preserves overall deduction consequent

lemma *ill-deduct-subst-consequent* [*simp*]:
  assumes $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *consequent* (*f G c l*) = *c*
    shows *consequent* (*ill-deduct-subst p f x*) = *consequent x*
  by (*induct x*) (*simp-all add*: *assms*)

Substitution matching target's antecedent, consequent and well-formedness preserves overall well-formedness

lemma *ill-deduct-subst-wf* [*simp*]:
  assumes $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *antecedents* (*f G c l*) = *G*
      and $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *consequent* (*f G c l*) = *c*
      and $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *ill-deduct-wf* (*f G c l*)
    shows *ill-deduct-wf x* = *ill-deduct-wf* (*ill-deduct-subst p f x*)
  using *assms* **by** (*induct x*) (*simp-all add*: *ill-conclusion-alt*)

Premises after substitution are those that didn't satisfy the predicate and anything that was introduced by the function applied on satisfying premises' parameters.

lemma *ill-deduct-subst-ill-deduct-premises*:
   ill-deduct-premises (*ill-deduct-subst p f x*)
  = *concat* (*map* (λ(*G*, *c*, *l*).
      if p G c l then ill-deduct-premises (*f G c l*) *else* [(*G*, *c*, *l*)])
    (*ill-deduct-premises x*))
  by (*induct x*) (*simp-all*)

This substitution commutes with many operations on deductions

lemma
  assumes $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *antecedents* (*f G c l*) = *G*
      and $\bigwedge G\ c\ l.\ p\ G\ c\ l \Longrightarrow$ *consequent* (*f G c l*) = *c*
    shows *ill-deduct-subst-simple-cut* [*simp*]:
       ill-deduct-subst p f (*ill-deduct-simple-cut X Y*)
      = *ill-deduct-simple-cut* (*ill-deduct-subst p f X*) (*ill-deduct-subst p f Y*)
      and *ill-deduct-subst′-tensor* [*simp*]:
       ill-deduct-subst p f (*ill-deduct-tensor X Y*) =
        ill-deduct-tensor (*ill-deduct-subst p f X*) (*ill-deduct-subst p f Y*)
      and *ill-deduct-subst-simple-plusL* [*simp*]:
       ill-deduct-subst p f (*ill-deduct-simple-plusL X Y*) =
        ill-deduct-simple-plusL (*ill-deduct-subst p f X*) (*ill-deduct-subst p f Y*)
      and *ill-deduct-subst-with* [*simp*]:
       ill-deduct-subst p f (*ill-deduct-with X Y*) =
        ill-deduct-with (*ill-deduct-subst p f X*) (*ill-deduct-subst p f Y*)
      and *ill-deduct-subst-simple-limpR* [*simp*]:
       ill-deduct-subst p f (*ill-deduct-simple-limpR X*) =
        ill-deduct-simple-limpR (*ill-deduct-subst p f X*)
      and *ill-deduct-subst-simple-limpR-exp* [*simp*]:

ill-deduct-subst p f (*ill-deduct-simple-limpR-exp X*) =
ill-deduct-simple-limpR-exp (*ill-deduct-subst p f X*)
using *assms* **by** (*simp-all add*: *ill-conclusion-alt*)

1.8.7 List-Based Exchange

To expand the applicability of the exchange rule to lists of propositions, we first need to establish that the well-formedness of a deduction is not affected by compacting a sublist of the antecedents of its conclusions. This corresponds to the following equality in the shallow embedding of deductions: *?X* @ [*compact ?G*] @ *?Y* ⊢ *?c* = *?X* @ *?G* @ *?Y* ⊢ *?c*.

For one direction of the equality we need to use *TimesL* to recursively add one proposition at a time into the compacted part of the antecedents. Note that, just like *compact*, the recursion terminates in the singleton case.

primrec *ill-deduct-compact-antecedents-split*
:: *nat* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
⇒ (*'a*, *'l*) *ill-deduct*
where
ill-deduct-compact-antecedents-split 0 X G Y P = *OneL* (*X* @ *G*) *Y* (*consequent P*) *P*
| *ill-deduct-compact-antecedents-split* (*Suc n*) *X G Y P* = (*if n* = *0 then P else*
TimesL
(*X* @ *take* (*length G* − (*Suc n*)) *G*)
(*hd* (*drop* (*length G* − (*Suc n*)) *G*))
(*compact* (*drop* (*length G* − *n*) *G*))
Y
(*consequent P*)
(*ill-deduct-compact-antecedents-split n X G Y P*))

lemma *ill-deduct-compact-antecedents-split* [*simp*]:
assumes *n* ≤ *length G*
shows *antecedents P* = *X* @ *G* @ *Y* ⟹
antecedents (*ill-deduct-compact-antecedents-split n X G Y P*)
= *X* @ *take* (*length G* − *n*) *G* @ [*compact* (*drop* (*length G* − *n*) *G*)] @ *Y*
and *consequent* (*ill-deduct-compact-antecedents-split n X G Y P*) = *consequent P*
and ⟦*antecedents P* = *X* @ *G* @ *Y*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-compact-antecedents-split n X G Y P*)
and *ill-deduct-premises* (*ill-deduct-compact-antecedents-split n X G Y P*)
= *ill-deduct-premises P*
proof −
have [*simp*]:
antecedents (*ill-deduct-compact-antecedents-split n X G Y P*)
= *X* @ *take* (*length G* − *n*) *G* @ [*compact* (*drop* (*length G* − *n*) *G*)] @ *Y*
if *antecedents P* = *X* @ *G* @ *Y* **and** *n* ≤ *length G* **for** *n X G Y* **and** *P* :: (*'c*, *'d*) *ill-deduct*
proof −

have *tol-hd-tl*: $\bigwedge$*xs ys*. $[\![$*ys* = *tl xs*; *ys* $\neq$ []$]\!]$ $\Longrightarrow$ *hd xs* $\otimes$ *compact ys* = *compact xs*
by (*metis list.collapse compact.simps*(*1*) *tl-Nil*)

show *?thesis*
using *that*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case *m*: (*Suc m*)
then show *?case*
proof (*cases m*)
case *0*
then have *drop* (*length G* − *1*) *G* = [*last G*]
using *m*
by (*metis Suc-le-lessD append-butlast-last-id append-eq-conv-conj length-butlast length-greater-0-conv*)
then show *?thesis*
using *m 0* **by** *simp* (*metis append-take-drop-id*)
next
case (*Suc m′*)
have *tl* (*drop* (*length G* − *Suc* (*Suc m′*)) *G*) = *drop* (*length G* − *Suc m′*) *G*
using *m.prems*(*2*) **by** (*metis Suc Suc-diff-Suc Suc-le-lessD drop-Suc tl-drop*)
then have
drop (*length G* − *Suc* (*Suc m′*)) *G*
= *hd* (*drop* (*length G* − *Suc* (*Suc m′*)) *G*) # *drop* (*length G* − *Suc m′*) *G*
using *m.prems*(*2*)
by (*metis Suc diff-diff-cancel diff-is-0-eq′ drop-eq-Nil hd-Cons-tl nat.distinct*(*1*))
moreover have *drop* (*length G* − *Suc m′*) *G* $\neq$ []
using *m.prems*(*2*) **by** *simp*
ultimately have
hd (*drop* (*length G* − *Suc* (*Suc m′*)) *G*) $\otimes$ *compact* (*drop* (*length G* − *Suc m′*) *G*)
= *compact* (*drop* (*length G* − *Suc* (*Suc m′*)) *G*)
by (*metis compact.simps*(*1*))
then show *?thesis*
using *Suc* **by** *simp*
qed
qed
qed
then show *antecedents P* = *X* @ *G* @ *Y* $\Longrightarrow$
antecedents (*ill-deduct-compact-antecedents-split n X G Y P*)
= *X* @ *take* (*length G* − *n*) *G* @ [*compact* (*drop* (*length G* − *n*) *G*)] @ *Y*
using *assms* **by** *simp*

have [*simp*]: *consequent* (*ill-deduct-compact-antecedents-split n X G Y P*) = *consequent P*
if *n* $\leq$ *length G* **for** *n X G Y* **and** *P* :: (′*a*, ′*l*) *ill-deduct*
by (*induct n*) *simp-all*

then show *consequent* (*ill-deduct-compact-antecedents-split n X G Y P*) = *consequent P*
using *assms* **.**

show ⟦*antecedents P* = *X* @ *G* @ *Y*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-compact-antecedents-split n X G Y P*)
using *assms* **by** (*induct n*) (*simp-all add*: *Suc-diff-Suc take-hd-drop ill-conclusion-alt*)
show
ill-deduct-premises (*ill-deduct-compact-antecedents-split n X G Y P*)
= *ill-deduct-premises P*
by (*induct n*) *simp-all*
qed

Implication in the uncompacted-to-compacted direction

fun *ill-deduct-antecedents-to-times*
:: *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
⇒ (*'a*, *'l*) *ill-deduct*
— *X* @ *G* @ *Y* ⊢ *c* ⟹ *X* @ [*compact G*] @ *Y* ⊢ *c*
where *ill-deduct-antecedents-to-times X G Y P* =
ill-deduct-compact-antecedents-split (*length G*) *X G Y P*

lemma *ill-deduct-antecedents-to-times* [*simp*]:
antecedents P = *X* @ *G* @ *Y* ⟹
antecedents (*ill-deduct-antecedents-to-times X G Y P*) = *X* @ [*compact G*] @ *Y*
consequent (*ill-deduct-antecedents-to-times X G Y P*) = *consequent P*
⟦*antecedents P* = *X* @ *G* @ *Y*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-antecedents-to-times X G Y P*)
ill-deduct-premises (*ill-deduct-antecedents-to-times X G Y P*) = *ill-deduct-premises P*
by *simp-all*

For the other direction we only need to derive the compacted propositions from the original list. This corresponds to the following valid sequent in the shallow embedding of deductions: *?G* ⊢ *compact ?G*.

fun *ill-deduct-identity-compact* :: *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
where
ill-deduct-identity-compact [] = *OneR*
| *ill-deduct-identity-compact* [*x*] = *Identity x*
| *ill-deduct-identity-compact* (*x*#*xs*) =
TimesR [*x*] *x xs* (*compact xs*) (*Identity x*) (*ill-deduct-identity-compact xs*)

lemma *ill-deduct-identity-compact* [*simp*]:
ill-conclusion (*ill-deduct-identity-compact G*) = *Sequent G* (*compact G*)
ill-deduct-wf (*ill-deduct-identity-compact G*)
ill-deduct-premises (*ill-deduct-identity-compact G*) = []
proof −
have [*simp*]: *ill-conclusion* (*ill-deduct-identity-compact G*) = *Sequent G* (*compact G*)
for *G* :: *'a ill-prop list*

by (*induct G rule*: *induct-list012*) *simp-all*
then show *ill-conclusion* (*ill-deduct-identity-compact G*) = *Sequent G* (*compact G*) **.**
show *ill-deduct-wf* (*ill-deduct-identity-compact G*)
by (*induct G rule*: *induct-list012*) (*simp-all add*: *ill-conclusion-alt*)
show *ill-deduct-premises* (*ill-deduct-identity-compact G*) = []
by (*induct G rule*: *induct-list012*) *simp-all*
qed

Implication in the compacted-to-uncompacted direction

fun *ill-deduct-antecedents-from-times*
:: *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ (*'a*, *'l*) *ill-deduct*
⇒ (*'a*, *'l*) *ill-deduct*
— *X* @ [*compact G*] @ *Y* ⊢ *c* ⟹ *X* @ *G* @ *Y* ⊢ *c*
where *ill-deduct-antecedents-from-times X G Y P* =
Cut G (*compact G*) *X Y* (*consequent P*) (*ill-deduct-identity-compact G*) *P*

lemma *ill-deduct-antecedents-from-times* [*simp*]:
ill-conclusion (*ill-deduct-antecedents-from-times X G Y P*) =
Sequent (*X* @ *G* @ *Y*) (*consequent P*)
⟦*antecedents P* = *X* @ [*compact G*] @ *Y*; *ill-deduct-wf P*⟧ ⟹
ill-deduct-wf (*ill-deduct-antecedents-from-times X G Y P*)
ill-deduct-premises (*ill-deduct-antecedents-from-times X G Y P*)
= *ill-deduct-premises P*
by (*simp-all add*: *ill-conclusion-alt*)

Finally, we establish the deep embedding of list-based exchange. This corresponds to the following theorem in the shallow embedding of deductions: *?G* @ *?A* @ *?B* @ *?D* ⊢ *?c* ⟹ *?G* @ *?B* @ *?A* @ *?D* ⊢ *?c*.

fun *ill-deduct-exchange-list*
:: *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop list* ⇒ *'a ill-prop*
⇒ (*'a*, *'l*) *ill-deduct* ⇒ (*'a*, *'l*) *ill-deduct*
where *ill-deduct-exchange-list G A B D c P* =
ill-deduct-antecedents-from-times G B (*A* @ *D*)
(*ill-deduct-antecedents-from-times* (*G* @ [*compact B*]) *A D*
(*Exchange G* (*compact A*) (*compact B*) *D c*
(*ill-deduct-antecedents-to-times* (*G* @ [*compact A*]) *B D*
(*ill-deduct-antecedents-to-times G A* (*B* @ *D*) *P*))))

lemma *ill-deduct-exchange-list* [*simp*]:
ill-conclusion (*ill-deduct-exchange-list G A B D c P*) = *Sequent* (*G* @ *B* @ *A* @ *D*) *c*
⟦*ill-deduct-wf P*; *antecedents P* = *G* @ *A* @ *B* @ *D*; *consequent P* = *c*⟧ ⟹
ill-deduct-wf (*ill-deduct-exchange-list G A B D c P*)
ill-deduct-premises (*ill-deduct-exchange-list G A B D c P*) = *ill-deduct-premises P*
by (*simp-all add*: *ill-conclusion-alt*)

end

References

[1] G. M. Bierman. On intuitionistic linear logic. Technical Report UCAM-CL-TR-346, University of Cambridge, Computer Laboratory, Aug. 1994.

[2] S. Kalvala and V. De Paiva. Mechanizing linear logic in Isabelle. In *In 10th International Congress of Logic, Philosophy and Methodology of Science*, volume 24. Citeseer, 1995.