A Reusable Isabelle/HOL Framework for Propositional Labelled Natural Deduction

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

June 25, 2026

Abstract

A reusable Isabelle/HOL framework presenting labelled natural deduction as a conservative refinement of ordinary natural deduction, with generic structural metatheory and several concrete label interpretations. We define ordinary intuitionistic propositional natural deduction, a parametric labelled refinement, prove erasure soundness (labelled derivations are ordinary derivations after erasing labels) and a controlled lifting theorem (ordinary derivations admit some labelling). Standard structural metatheory — weakening, exchange, substitution, cut admissibility — is established in both the ordinary and labelled systems.

The framework is instantiated three times. Unit labels recover ordinary ND as a special case via conservativity. Provenance labels carry sets of assumption indices and give both a sound over-approximation of assumption dependencies and a completeness result equating ordinary derivability with the existence of a provenance-labelled derivation whose filtered sub-context still derives the conclusion. A possible-world modal instance derives the K axiom schema and necessitation and is connected to Kripke validity; the modal fragment is presented as an example instantiation, not as a full labelled modal proof theory.

The development is fully constructive, contains no unfinished proof commands or oracle invocations, and is intended as a foundation other entries can cite.

Contents

1 What this entry proves

The main named results, with the theory in which each is established, are:

- `labelled_sound_under_erasure` (`Erasure`) — every labelled derivation erases to an ordinary derivation.
- `lifting_general`, `ordinary_derivations_admit_labelling` (`Label_Algebra`, `Lifting`) — ordinary derivations admit a labelling under the assumed lift-coherence law.
- `framework_adequacy`, `framework_adequacy_iff` (`Label_Algebra`) — the locale-level equivalence between labelled derivability up to a label and ordinary derivability after erasure.
- `unit_labels_conservative` (`Unit_Labels`) — the unit-labels instance recovers ordinary ND.
- `provenance_overapproximates_dependencies` (`Provenance_Labels`) — the provenance instance gives a sound over-approximation of assumption dependencies.
- `ordinary_admits_safe_provenance`, `ordinary_admits_tight_provenance`, `provenance_completeness` (`Provenance_Labels`) — the provenance instance is dependency-tracking complete: ordinary derivability is equivalent to the existence of a provenance-labelled derivation whose selected filtered context still derives the conclusion.
- `modal_K_schema`, `modal_necessitation`, `modal_K_kripke_valid` (`Modal_K_Schema`) — the worked modal instance, where the K axiom schema is derived uniformly and connected to Kripke validity.

2 Overview

Labelled deductive systems were introduced by Gabbay as a general way to attach structured information to formulae and rules [2]. The modal example in this entry follows the standard possible-world reading of modal labels, closely related to labelled proof-theoretic presentations such as Simpson's work on intuitionistic modal logic [3]. Labelled approaches to non-classical proof systems have been developed extensively by Viganò [4] and by Basin, Matthews, and Viganò [1]; our framework targets the simpler proof-theoretic core of propositional intuitionistic ND but recovers similar structural properties.

3 Label algebra and adequacy

The entry factors the parametric part of labelled derivability through a label-algebra locale. The locale records the proof obligations needed by any future label instance: that label constructors are uniform under erasure to the singleton type and that ordinary derivations can be lifted coherently over the labelled context. Its central theorem, `framework_adequacy`, states that labelled derivability up to a label is equivalent to ordinary derivability after erasing the context, with the chosen lift supplying the canonical labelled witness. The result is intentionally architectural rather than surprising: it packages the soundness/lifting obligations a new label instance must discharge, so that once those obligations are met the equivalence comes for free.

4 Modal K axiom and necessitation

The worked modal instance is presented as an example, not as a self-contained labelled modal proof theory, but it is strong enough to derive the K axiom schema uniformly in the formulae A, B, the accessibility relation R, and the world w. The theory `Modal_K_Schema.thy` records this as `modal_K_schema`, proves `modal_necessitation` under the standard side-condition that necessitation applies only to formulae derivable from the empty context at every world and relation, and connects the labelled derivation to Kripke semantics through `modal_K_kripke_valid`.

5 Provenance completeness

The provenance instance is the entry's main dependency-tracking result. It is adequate in a strong sense: every ordinary derivation lifts to a labelled derivation whose provenance is contained in the indices of the ordinary context and whose selected sub-context is still sufficient to re-derive the conclusion. The relevant named theorems are:

- `ordinary_admits_safe_provenance` — records the index-bound and well-formedness properties of the lifted provenance derivation.
- `ordinary_admits_tight_provenance` — strengthens this to a sufficient filtered sub-context.
- `provenance_completeness` — states the resulting equivalence between ordinary derivability and provenance-labelled derivability with a sufficient filtered context.

theory *Labelled-Formulas*
  imports *Main*
begin

This theory fixes the propositional formula language and the elementary operations that erase labels from labelled formulae, databases, and list contexts. Labels are deliberately represented by a product component, so the logical shape of a labelled formula is recovered by taking its second projection.

datatype *'a fm* =
    Atom 'a
  | *Bot*
  | *Imp 'a fm 'a fm*
  | *Con 'a fm 'a fm*
  | *Dis 'a fm 'a fm*

type-synonym *('l, 'a) lfm* = *'l* × *'a fm*

definition *erase-lfm* :: *('l, 'a) lfm* ⇒ *'a fm* **where**
  erase-lfm x = *snd x*

definition *erase-db* :: *('l, 'a) lfm set* ⇒ *'a fm set* **where**
  erase-db Γ = *snd* ‘ Γ

definition *erase-ctx* :: *('l, 'a) lfm list* ⇒ *'a fm list* **where**
  erase-ctx Γ = *map snd* Γ

lemma *erase-lfm-simps* [*simp*]:
  erase-lfm (*l*, *A*) = *A*
  by (*simp add*: *erase-lfm-def*)

lemma *erase-ctx-Nil* [*simp*]:
  erase-ctx [] = []
  by (*simp add*: *erase-ctx-def*)

lemma *erase-ctx-Cons* [*simp*]:
  erase-ctx (*x* # Γ) = *erase-lfm x* # *erase-ctx* Γ
  by (*cases x*) (*simp add*: *erase-ctx-def*)

lemma *erase-ctx-append* [*simp*]:
  erase-ctx (Γ @ Δ) = *erase-ctx* Γ @ *erase-ctx* Δ
  by (*simp add*: *erase-ctx-def*)

lemma *erase-db-insert* [*simp*]:
  erase-db (*insert x* Γ) = *insert* (*erase-lfm x*) (*erase-db* Γ)
  by (*cases x*) (*auto simp*: *erase-db-def erase-lfm-def*)

lemma *erase-db-Un* [*simp*]:
  erase-db (Γ ∪ Δ) = *erase-db* Γ ∪ *erase-db* Δ

by (*auto simp*: *erase-db-def*)

end

theory *Natural-Deduction*
imports *Labelled-Formulas*
begin

This theory defines ordinary intuitionistic propositional natural deduction over list contexts. The structural metatheory is stated in terms of the set of assumptions represented by a list, which keeps exchange and weakening explicit while avoiding dependence on a particular context order.

primrec *fm-subst* :: ('*a* ⇒ '*b fm*) ⇒ '*a fm* ⇒ '*b fm* **where**
fm-subst σ (*Atom p*) = σ *p*
| *fm-subst* σ *Bot* = *Bot*
| *fm-subst* σ (*Imp A B*) = *Imp* (*fm-subst* σ *A*) (*fm-subst* σ *B*)
| *fm-subst* σ (*Con A B*) = *Con* (*fm-subst* σ *A*) (*fm-subst* σ *B*)
| *fm-subst* σ (*Dis A B*) = *Dis* (*fm-subst* σ *A*) (*fm-subst* σ *B*)

inductive *derives* :: '*a fm list* ⇒ '*a fm* ⇒ *bool* (- ⊢ - [*50*, *50*] *50*)
where
Assm: *A* ∈ *set* Γ ⟹ Γ ⊢ *A*
| *BotE*: Γ ⊢ *Bot* ⟹ Γ ⊢ *A*
| *ImpI*: *A* # Γ ⊢ *B* ⟹ Γ ⊢ *Imp A B*
| *ImpE*: Γ ⊢ *Imp A B* ⟹ Γ ⊢ *A* ⟹ Γ ⊢ *B*
| *ConI*: Γ ⊢ *A* ⟹ Γ ⊢ *B* ⟹ Γ ⊢ *Con A B*
| *ConE1*: Γ ⊢ *Con A B* ⟹ Γ ⊢ *A*
| *ConE2*: Γ ⊢ *Con A B* ⟹ Γ ⊢ *B*
| *DisI1*: Γ ⊢ *A* ⟹ Γ ⊢ *Dis A B*
| *DisI2*: Γ ⊢ *B* ⟹ Γ ⊢ *Dis A B*
| *DisE*: Γ ⊢ *Dis A B* ⟹ *A* # Γ ⊢ *C* ⟹ *B* # Γ ⊢ *C* ⟹ Γ ⊢ *C*

lemma *weakening*:
assumes Γ ⊢ *A* **and** *set* Γ ⊆ *set* Δ
shows Δ ⊢ *A*
using *assms*
proof (*induction arbitrary*: Δ *rule*: *derives.induct*)
case (*Assm A* Γ)
then show *?case*
by (*meson derives.Assm subsetD*)
next
case (*BotE* Γ *A*)
then show *?case*
by (*meson derives.BotE*)
next
case (*ImpI A* Γ *B*)
show *?case*
proof (*rule derives.ImpI*)

```
    show A # Δ ⊢ B
      by (rule ImpI.IH) (use ImpI.prems in auto)
  qed
next
  case (ImpE Γ A B)
  then show ?case
    by (meson derives.ImpE)
next
  case (ConI Γ A B)
  then show ?case
    by (meson derives.ConI)
next
  case (ConE1 Γ A B)
  then show ?case
    by (meson derives.ConE1)
next
  case (ConE2 Γ A B)
  then show ?case
    by (meson derives.ConE2)
next
  case (DisI1 Γ A B)
  then show ?case
    by (meson derives.DisI1)
next
  case (DisI2 Γ B A)
  then show ?case
    by (meson derives.DisI2)
next
  case (DisE Γ A B C)
  show ?case
  proof (rule derives.DisE)
    show Δ ⊢ Dis A B
      using DisE.IH(1) DisE.prems by blast
    show A # Δ ⊢ C
      by (rule DisE.IH(2)) (use DisE.prems in auto)
    show B # Δ ⊢ C
      by (rule DisE.IH(3)) (use DisE.prems in auto)
  qed
qed

lemma weakening-cons:
  assumes Γ ⊢ A
  shows B # Γ ⊢ A
  using assms by (rule weakening) auto

lemma exchange:
  assumes A # B # Γ ⊢ C
  shows B # A # Γ ⊢ C
  using assms by (rule weakening) auto
```

lemma *cut-general*:
assumes $\Gamma \vdash A$ **and** $\Delta \vdash B$ **and** *set* $\Delta \subseteq$ *insert* A (*set* Γ)
shows $\Gamma \vdash B$
using *assms*(*2*,*3*,*1*)
proof (*induction arbitrary*: Γ A *rule*: *derives.induct*)
case (*Assm* B Δ)
from *Assm.hyps Assm.prems*(*1*) **have** $B = A \vee B \in$ *set* Γ
by *auto*
then show *?case*
proof
assume $B = A$
then show *?thesis*
using *Assm.prems*(*2*) **by** *simp*
next
assume $B \in$ *set* Γ
then show *?thesis*
by (*rule derives.Assm*)
qed
next
case (*BotE* Δ B)
then show *?case*
by (*meson derives.BotE*)
next
case (*ImpI* C Δ D)
show *?case*
proof (*rule derives.ImpI*)
have *deriv*: $C \# \Gamma \vdash A$
using *ImpI.prems*(*2*) **by** (*rule weakening-cons*)
have *sub*: *set* ($C \# \Delta$) $\subseteq$ *insert* A (*set* ($C \# \Gamma$))
using *ImpI.prems*(*1*) **by** *auto*
show $C \# \Gamma \vdash D$
by (*rule ImpI.IH*[*OF sub deriv*])
qed
next
case (*ImpE* Δ C D)
then show *?case*
by (*meson derives.ImpE*)
next
case (*ConI* Δ C D)
then show *?case*
by (*meson derives.ConI*)
next
case (*ConE1* Δ C D)
then show *?case*
by (*meson derives.ConE1*)
next
case (*ConE2* Δ C D)
then show *?case*

```
    by (meson derives.ConE2)
next
  case (DisI1 Δ C D)
  then show ?case
    by (meson derives.DisI1)
next
  case (DisI2 Δ D C)
  then show ?case
    by (meson derives.DisI2)
next
  case (DisE Δ C D E)
  show ?case
  proof (rule derives.DisE)
    show Γ ⊢ Dis C D
      using DisE.IH(1) DisE.prems by blast
    have deriv-C: C # Γ ⊢ A
      using DisE.prems(2) by (rule weakening-cons)
    have sub-C: set (C # Δ) ⊆ insert A (set (C # Γ))
      using DisE.prems(1) by auto
    show C # Γ ⊢ E
      by (rule DisE.IH(2)[OF sub-C deriv-C])
    have deriv-D: D # Γ ⊢ A
      using DisE.prems(2) by (rule weakening-cons)
    have sub-D: set (D # Δ) ⊆ insert A (set (D # Γ))
      using DisE.prems(1) by auto
    show D # Γ ⊢ E
      by (rule DisE.IH(3)[OF sub-D deriv-D])
  qed
qed

lemma cut:
  assumes Γ ⊢ A and A # Γ ⊢ B
  shows Γ ⊢ B
  using assms by (rule cut-general) auto

theorem deduction-theorem:
  A # Γ ⊢ B ⟷ Γ ⊢ Imp A B
proof
  show A # Γ ⊢ B ⟹ Γ ⊢ Imp A B
    by (rule derives.ImpI)
next
  assume Γ ⊢ Imp A B
  then have A # Γ ⊢ Imp A B
    by (rule weakening-cons)
  moreover have A # Γ ⊢ A
    by (rule derives.Assm) simp
  ultimately show A # Γ ⊢ B
    by (rule derives.ImpE)
qed
```

lemma *substitution-atom*:
assumes $\Gamma \vdash A$
shows *map* (*fm-subst* σ) $\Gamma \vdash$ *fm-subst* σ A
using *assms*
by (*induction rule*: *derives.induct*) (*auto intro*: *derives.intros*)

end

theory *Labelled-Natural-Deduction*
imports *Natural-Deduction*
begin

This theory introduces the parametric labelled natural-deduction kernel. A locale fixes inert label constructors for the logical rules, but imposes no equations on them; concrete label disciplines are supplied later by interpretation.

locale *label-structure* =
fixes *app* :: $'l \Rightarrow 'l \Rightarrow 'l$
and *lam* :: $'l \Rightarrow 'l \Rightarrow 'l$
and *pair* :: $'l \Rightarrow 'l \Rightarrow 'l$
and *fst-l* :: $'l \Rightarrow 'l$
and *snd-l* :: $'l \Rightarrow 'l$
and *abort* :: $'l \Rightarrow 'l$
and *inl* :: $'l \Rightarrow 'l$
and *inr* :: $'l \Rightarrow 'l$
and *cases* :: $'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l$
begin

inductive *lderives* :: $('l, 'a)$ *lfm list* $\Rightarrow$ $('l, 'a)$ *lfm* $\Rightarrow$ *bool*
(- $\vdash L$ - [*50*, *50*] *50*)
where
LAssm: $x \in$ *set* $\Gamma \Longrightarrow \Gamma \vdash L\ x$
| *LBotE*: $\Gamma \vdash L\ (l, Bot) \Longrightarrow \Gamma \vdash L\ (abort\ l, A)$
| *LImpI*: $(l, A)\ \#\ \Gamma \vdash L\ (m, B) \Longrightarrow \Gamma \vdash L\ (lam\ l\ m, Imp\ A\ B)$
| *LImpE*: $\Gamma \vdash L\ (l, Imp\ A\ B) \Longrightarrow \Gamma \vdash L\ (m, A) \Longrightarrow \Gamma \vdash L\ (app\ l\ m, B)$
| *LConI*: $\Gamma \vdash L\ (l, A) \Longrightarrow \Gamma \vdash L\ (m, B) \Longrightarrow \Gamma \vdash L\ (pair\ l\ m, Con\ A\ B)$
| *LConE1*: $\Gamma \vdash L\ (l, Con\ A\ B) \Longrightarrow \Gamma \vdash L\ (fst\text{-}l\ l, A)$
| *LConE2*: $\Gamma \vdash L\ (l, Con\ A\ B) \Longrightarrow \Gamma \vdash L\ (snd\text{-}l\ l, B)$
| *LDisI1*: $\Gamma \vdash L\ (l, A) \Longrightarrow \Gamma \vdash L\ (inl\ l, Dis\ A\ B)$
| *LDisI2*: $\Gamma \vdash L\ (l, B) \Longrightarrow \Gamma \vdash L\ (inr\ l, Dis\ A\ B)$
| *LDisE*:
$\Gamma \vdash L\ (l, Dis\ A\ B) \Longrightarrow$
$(m, A)\ \#\ \Gamma \vdash L\ (n, C) \Longrightarrow$
$(p, B)\ \#\ \Gamma \vdash L\ (q, C) \Longrightarrow$
$\Gamma \vdash L\ (cases\ l\ m\ n\ p\ q, C)$

lemma *labelled-weakening*:

```
  assumes Γ ⊢L x and set Γ ⊆ set Δ
  shows Δ ⊢L x
  using assms
proof (induction arbitrary: Δ rule: lderives.induct)
  case (LAssm x Γ)
  then have x ∈ set Δ
    by auto
  then show ?case
    by (rule lderives.intros(1))
next
  case (LBotE Γ l A)
  then show ?case
    by (meson lderives.intros)
next
  case (LImpI l A Γ m B)
  show ?case
  proof (rule lderives.intros(3))
    show (l, A) # Δ ⊢L (m, B)
      by (rule LImpI.IH) (use LImpI.prems in auto)
  qed
next
  case (LImpE Γ l A B m)
  then show ?case
    by (meson lderives.intros)
next
  case (LConI Γ l A m B)
  then show ?case
    by (meson lderives.intros)
next
  case (LConE1 Γ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LConE2 Γ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisI1 Γ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisI2 Γ l B A)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisE Γ l A B m n C p q)
  show ?case
  proof (rule lderives.intros(10))
    show Δ ⊢L (l, Dis A B)
```

```
    using LDisE.IH(1) LDisE.prems by blast
  show (m, A) # Δ ⊢L (n, C)
    by (rule LDisE.IH(2)) (use LDisE.prems in auto)
  show (p, B) # Δ ⊢L (q, C)
    by (rule LDisE.IH(3)) (use LDisE.prems in auto)
 qed
qed

lemma labelled-weakening-cons:
 assumes Γ ⊢L x
 shows y # Γ ⊢L x
 using assms by (rule labelled-weakening) auto

lemma labelled-exchange:
 assumes x # y # Γ ⊢L z
 shows y # x # Γ ⊢L z
 using assms by (rule labelled-weakening) auto

lemma labelled-cut-general:
 assumes Γ ⊢L x
   and Δ ⊢L y
   and set Δ ⊆ insert x (set Σ)
   and set Γ ⊆ set Σ
 shows Σ ⊢L y
 using assms(2,3,1,4)
proof (induction arbitrary: Γ x Σ rule: lderives.induct)
 case (LAssm y Δ)
 from LAssm.hyps LAssm.prems(1) have y = x ∨ y ∈ set Σ
   by auto
 then show ?case
 proof
   assume y = x
   then show ?thesis
     using LAssm.prems(2,3) labelled-weakening by blast
 next
   assume y ∈ set Σ
   then show ?thesis
     by (rule lderives.intros(1))
 qed
next
 case (LBotE Δ l A)
 then show ?case
   by (meson lderives.intros)
next
 case (LImpI l A Δ m B)
 show ?case
 proof (rule lderives.intros(3))
   have sub: set ((l, A) # Δ) ⊆ insert x (set ((l, A) # Σ))
     using LImpI.prems(1) by auto
```

```
    have sub-context: set Γ ⊆ set ((l, A) # Σ)
      using LImpI.prems(3) by auto
    show (l, A) # Σ ⊢L (m, B)
      by (rule LImpI.IH[OF sub LImpI.prems(2) sub-context])
  qed
next
  case (LImpE Δ l A B m)
  then show ?case
    by (meson lderives.intros)
next
  case (LConI Δ l A m B)
  then show ?case
    by (meson lderives.intros)
next
  case (LConE1 Δ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LConE2 Δ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisI1 Δ l A B)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisI2 Δ l B A)
  then show ?case
    by (meson lderives.intros)
next
  case (LDisE Δ l A B m n C p q)
  show ?case
  proof (rule lderives.intros(10))
    show Σ ⊢L (l, Dis A B)
      using LDisE.IH(1) LDisE.prems by blast
    have sub-m: set ((m, A) # Δ) ⊆ insert x (set ((m, A) # Σ))
      using LDisE.prems(1) by auto
    have sub-context-m: set Γ ⊆ set ((m, A) # Σ)
      using LDisE.prems(3) by auto
    show (m, A) # Σ ⊢L (n, C)
      by (rule LDisE.IH(2)[OF sub-m LDisE.prems(2) sub-context-m])
    have sub-p: set ((p, B) # Δ) ⊆ insert x (set ((p, B) # Σ))
      using LDisE.prems(1) by auto
    have sub-context-p: set Γ ⊆ set ((p, B) # Σ)
      using LDisE.prems(3) by auto
    show (p, B) # Σ ⊢L (q, C)
      by (rule LDisE.IH(3)[OF sub-p LDisE.prems(2) sub-context-p])
  qed
qed
```

lemma *labelled-cut*:
assumes $\Gamma \vdash L\ x$ **and** $x\ \#\ \Delta \vdash L\ y$
shows $\Gamma\ @\ \Delta \vdash L\ y$
proof (*rule labelled-cut-general*)
show $\Gamma \vdash L\ x$
by (*rule assms*(*1*))
show $x\ \#\ \Delta \vdash L\ y$
by (*rule assms*(*2*))
show *set* $(x\ \#\ \Delta) \subseteq$ *insert* x (*set* $(\Gamma\ @\ \Delta)$)
by *auto*
show *set* $\Gamma \subseteq$ *set* $(\Gamma\ @\ \Delta)$
by *auto*
qed

lemma *assumption-substitution-lift*:
assumes $x \in$ *set* Γ
shows (*fst* x, *fm-subst* σ (*snd* x)) $\in$ *set* (*map* (*map-prod id* (*fm-subst* σ)) Γ)
using *assms* **by** (*induction* Γ) *auto*

lemma *labelled-substitution-atom-aux*:
assumes $\Gamma \vdash L\ x$
shows *map* (*map-prod id* (*fm-subst* σ)) $\Gamma \vdash L$ (*fst* x, *fm-subst* σ (*snd* x))
using *assms*
proof (*induction rule*: *lderives.induct*)
case (*LAssm* $x\ \Gamma$)
then have (*fst* x, *fm-subst* σ (*snd* x)) $\in$
set (*map* (*map-prod id* (*fm-subst* σ)) Γ)
by (*rule assumption-substitution-lift*)
then show *?case*
by (*rule lderives.LAssm*)
next
case (*LBotE* $\Gamma\ l\ A$)
then show *?case*
by (*auto intro*: *lderives.LBotE*)
next
case (*LImpI* $l\ A\ \Gamma\ m\ B$)
then have (l, *fm-subst* $\sigma\ A$) $\#$ *map* (*map-prod id* (*fm-subst* σ)) Γ
$\vdash L$ (m, *fm-subst* $\sigma\ B$)
by (*simp add*: *id-def*)
then have *map* (*map-prod id* (*fm-subst* σ)) Γ
$\vdash L$ (*lam* $l\ m$, *Imp* (*fm-subst* $\sigma\ A$) (*fm-subst* $\sigma\ B$))
by (*rule lderives.LImpI*)
then show *?case*
by (*simp add*: *id-def*)
next
case (*LImpE* $\Gamma\ l\ A\ B\ m$)
then show *?case*
by (*auto intro*: *lderives.LImpE*)

next
case (*LConI* Γ *l A m B*)
then show *?case*
by (*auto intro*: *lderives.LConI*)
next
case (*LConE1* Γ *l A B*)
then show *?case*
by (*auto intro*: *lderives.LConE1*)
next
case (*LConE2* Γ *l A B*)
then show *?case*
by (*auto intro*: *lderives.LConE2*)
next
case (*LDisI1* Γ *l A B*)
then show *?case*
by (*auto intro*: *lderives.LDisI1*)
next
case (*LDisI2* Γ *l B A*)
then show *?case*
by (*auto intro*: *lderives.LDisI2*)
next
case (*LDisE* Γ *l A B m n C p q*)
then show *?case*
by (*auto intro*: *lderives.LDisE*)
qed

lemma *labelled-substitution-atom*:
assumes Γ ⊢*L* (*l*, *A*)
shows *map* (*map-prod id* (*fm-subst σ*)) Γ ⊢*L* (*l*, *fm-subst σ A*)
using *labelled-substitution-atom-aux*[*OF assms*] **by** *simp*

end

end

theory *Erasure*
imports *Labelled-Natural-Deduction*
begin

This theory proves that labels are conservative annotations for the parametric kernel: every labelled derivation erases to an ordinary natural deduction derivation of the underlying formula from the erased context.

context *label-structure*
begin

theorem *labelled-sound-under-erasure*:
assumes Γ ⊢*L x*
shows *erase-ctx* Γ ⊢ *erase-lfm x*

```
  using assms
proof (induction rule: lderives.induct)
  case (LAssm x Γ)
  have erase-lfm x ∈ set (erase-ctx Γ)
    using LAssm.hyps by (induction Γ) (auto simp: erase-ctx-def erase-lfm-def)
  then show ?case
    by (rule derives.Assm)
next
  case (LBotE Γ l A)
  then show ?case
    by (simp add: derives.BotE)
next
  case (LImpI l A Γ m B)
  then show ?case
    by (simp add: derives.ImpI)
next
  case (LImpE Γ l A B m)
  then show ?case
    by (simp add: derives.ImpE)
next
  case (LConI Γ l A m B)
  then show ?case
    by (simp add: derives.ConI)
next
  case (LConE1 Γ l A B)
  then show ?case
    by (simp add: derives.ConE1)
next
  case (LConE2 Γ l A B)
  then show ?case
    by (simp add: derives.ConE2)
next
  case (LDisI1 Γ l A B)
  then show ?case
    by (simp add: derives.DisI1)
next
  case (LDisI2 Γ l B A)
  then show ?case
    by (simp add: derives.DisI2)
next
  case (LDisE Γ l A B m n C p q)
  then show ?case
    by (simp add: derives.DisE)
qed

lemmas erasure-sound = labelled-sound-under-erasure

end
```

end

theory *Label-Algebra*
imports *Erasure*
begin

The label-algebra layer packages the two generic adequacy requirements used by the concrete interpretations. Label erasure targets *unit*, so every erasure map is extensionally the unique one. The lift operation may inspect the labelled context, which is needed for calculi whose assumption labels carry semantic data.

lemma *erase-ctx-mem-label*:
assumes $A \in$ *set* (*erase-ctx* Γ)
obtains l **where** $(l, A) \in$ *set* Γ
proof −
from *assms* **obtain** x **where** $x \in$ *set* Γ **and** *snd* $x = A$
unfolding *erase-ctx-def* **by** *auto*
obtain l B **where** $x = (l, B)$
by (*cases* x)
with ‹$x \in$ *set* Γ› ‹*snd* $x = A$› **have** $(l, A) \in$ *set* Γ
by *simp*
then show *thesis*
by (*rule that*)
qed

context *label-structure*
begin

lemma *lifting-general-aux*:
assumes $\Delta \vdash A$ **and** *erase-ctx* $\Gamma = \Delta$
shows $\exists\, l.\ \Gamma \vdash L\ (l, A)$
using *assms*
proof (*induction arbitrary*: Γ *rule*: *derives.induct*)
case (*Assm* A Δ)
then have $A \in$ *set* (*erase-ctx* Γ)
by *simp*
then obtain l **where** $(l, A) \in$ *set* Γ
by (*rule erase-ctx-mem-label*)
then have $\Gamma \vdash L\ (l, A)$
by (*rule LAssm*)
then show *?case*
by *blast*
next
case (*BotE* Δ A)
then obtain l **where** $\Gamma \vdash L\ (l, Bot)$
by *blast*
then have $\Gamma \vdash L\ (abort\ l, A)$
by (*rule LBotE*)

then show *?case*
by *blast*
next
case (*ImpI A Δ B*)
have *erase-ctx* ((*undefined*, *A*) # Γ) = *A* # Δ
using *ImpI.prems* **by** *simp*
then obtain *m* **where** (*undefined*, *A*) # Γ ⊢*L* (*m*, *B*)
using *ImpI.IH* **by** *blast*
then have Γ ⊢*L* (*lam undefined m*, *Imp A B*)
by (*rule LImpI*)
then show *?case*
by *blast*
next
case (*ImpE Δ A B*)
then obtain *l* **where** *imp*: Γ ⊢*L* (*l*, *Imp A B*)
by *blast*
from *ImpE* **obtain** *m* **where** *arg*: Γ ⊢*L* (*m*, *A*)
by *blast*
from *imp arg* **have** Γ ⊢*L* (*app l m*, *B*)
by (*rule LImpE*)
then show *?case*
by *blast*
next
case (*ConI Δ A B*)
then obtain *l* **where** *left*: Γ ⊢*L* (*l*, *A*)
by *blast*
from *ConI* **obtain** *m* **where** *right*: Γ ⊢*L* (*m*, *B*)
by *blast*
from *left right* **have** Γ ⊢*L* (*pair l m*, *Con A B*)
by (*rule LConI*)
then show *?case*
by *blast*
next
case (*ConE1 Δ A B*)
then obtain *l* **where** Γ ⊢*L* (*l*, *Con A B*)
by *blast*
then have Γ ⊢*L* (*fst-l l*, *A*)
by (*rule LConE1*)
then show *?case*
by *blast*
next
case (*ConE2 Δ A B*)
then obtain *l* **where** Γ ⊢*L* (*l*, *Con A B*)
by *blast*
then have Γ ⊢*L* (*snd-l l*, *B*)
by (*rule LConE2*)
then show *?case*
by *blast*
next

```
  case (DisI1 Δ A B)
  then obtain l where Γ ⊢L (l, A)
    by blast
  then have Γ ⊢L (inl l, Dis A B)
    by (rule LDisI1)
  then show ?case
    by blast
next
  case (DisI2 Δ B A)
  then obtain l where Γ ⊢L (l, B)
    by blast
  then have Γ ⊢L (inr l, Dis A B)
    by (rule LDisI2)
  then show ?case
    by blast
next
  case (DisE Δ A B C)
  then obtain l where disj: Γ ⊢L (l, Dis A B)
    by blast
  have left-ctx: erase-ctx ((undefined, A) # Γ) = A # Δ
    using DisE.prems by simp
  then obtain n where left: (undefined, A) # Γ ⊢L (n, C)
    using DisE.IH(2) by blast
  have right-ctx: erase-ctx ((undefined, B) # Γ) = B # Δ
    using DisE.prems by simp
  then obtain q where right: (undefined, B) # Γ ⊢L (q, C)
    using DisE.IH(3) by blast
  from disj left right have Γ ⊢L (cases l undefined n undefined q, C)
    by (rule LDisE)
  then show ?case
    by blast
qed

theorem lifting-general:
  assumes erase-ctx Γ ⊢ A
  shows ∃ l. Γ ⊢L (l, A)
  using lifting-general-aux[OF assms refl] .

end

locale label-algebra =
  fixes app :: 'l ⇒ 'l ⇒ 'l
    and lam :: 'l ⇒ 'l ⇒ 'l
    and pair :: 'l ⇒ 'l ⇒ 'l
    and fst-l :: 'l ⇒ 'l
    and snd-l :: 'l ⇒ 'l
    and abort :: 'l ⇒ 'l
    and inl :: 'l ⇒ 'l
    and inr :: 'l ⇒ 'l
```

and *cases* :: $'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l \Rightarrow 'l$
and *lderives* :: $('l, 'f)$ *lfm list* $\Rightarrow$ $('l, 'f)$ *lfm* $\Rightarrow$ *bool*
(- $\vdash L$ - [50, 50] 50)
and *erase-label* :: $'l \Rightarrow$ *unit*
and *lift-label* :: $('l, 'f)$ *lfm list* $\Rightarrow$ $'f$ *fm* $\Rightarrow$ $'l$
assumes *erase-label-unique* [*simp*]: *erase-label* $l = ()$
and *erasure-sound-law*:
$\Gamma \vdash L\ x \Longrightarrow$ *erase-ctx* $\Gamma \vdash$ *erase-lfm* x
and *lift-coherence*:
erase-ctx $\Gamma \vdash A \Longrightarrow \Gamma \vdash L$ (*lift-label* $\Gamma\ A, A$)
begin

definition *erase-judgement* ::
$(('l, 'f)$ *lfm list* $\times$ $('l, 'f)$ *lfm*$) \Rightarrow ('f$ *fm list* $\times$ $'f$ *fm*$)$
where *erase-judgement* $J =$ (*erase-ctx* (*fst* J), *erase-lfm* (*snd* J))

lemma *erase-judgement-simps* [*simp*]:
erase-judgement $(\Gamma, (l, A)) =$ (*erase-ctx* Γ, A)
by (*simp add*: *erase-judgement-def*)

lemma *erasure-uniformity*:
erase-label (*app* $l\ m$) = ()
erase-label (*lam* $l\ m$) = ()
erase-label (*pair* $l\ m$) = ()
erase-label (*fst-l* l) = ()
erase-label (*snd-l* l) = ()
erase-label (*abort* l) = ()
erase-label (*inl* l) = ()
erase-label (*inr* l) = ()
erase-label (*cases* $l\ m\ n\ p\ q$) = ()
by *simp-all*

lemma *erasure-uniformity-judgement*:
erase-judgement $(\Gamma, ($*app* $l\ m, A)) =$ (*erase-ctx* Γ, A)
erase-judgement $(\Gamma, ($*lam* $l\ m$, *Imp* $A\ B)) =$ (*erase-ctx* Γ, *Imp* $A\ B$)
erase-judgement $(\Gamma, ($*pair* $l\ m$, *Con* $A\ B)) =$ (*erase-ctx* Γ, *Con* $A\ B$)
erase-judgement $(\Gamma, ($*fst-l* $l, A)) =$ (*erase-ctx* Γ, A)
erase-judgement $(\Gamma, ($*snd-l* $l, B)) =$ (*erase-ctx* Γ, B)
erase-judgement $(\Gamma, ($*abort* $l, A)) =$ (*erase-ctx* Γ, A)
erase-judgement $(\Gamma, ($*inl* l, *Dis* $A\ B)) =$ (*erase-ctx* Γ, *Dis* $A\ B$)
erase-judgement $(\Gamma, ($*inr* l, *Dis* $A\ B)) =$ (*erase-ctx* Γ, *Dis* $A\ B$)
erase-judgement $(\Gamma, ($*cases* $l\ m\ n\ p\ q, C)) =$ (*erase-ctx* Γ, C)
by *simp-all*

theorem *framework-adequacy*:
fixes $\Gamma L ::$ $('l, 'f)$ *lfm list*
and $A ::$ $'f$ *fm*
shows $(\exists l.\ \Gamma L \vdash L\ (l, A)) \longleftrightarrow$
(*erase-ctx* $\Gamma L \vdash A \wedge \Gamma L \vdash L$ (*lift-label* $\Gamma L\ A, A$))

proof
assume $\exists\, l.\ \Gamma L \vdash L\ (l,\ A)$
then obtain l **where** *deriv*: $\Gamma L \vdash L\ (l,\ A)$
by *blast*
have *erase-ctx* $\Gamma L \vdash$ *erase-lfm* $(l,\ A)$
using *deriv* **by** (*rule erasure-sound-law*)
then have *ordinary*: *erase-ctx* $\Gamma L \vdash A$
by *simp*
moreover have $\Gamma L \vdash L$ (*lift-label* $\Gamma L\ A,\ A$)
using *ordinary* **by** (*rule lift-coherence*)
ultimately show *erase-ctx* $\Gamma L \vdash A \wedge \Gamma L \vdash L$ (*lift-label* $\Gamma L\ A,\ A$)
by *blast*
next
assume *erase-ctx* $\Gamma L \vdash A \wedge \Gamma L \vdash L$ (*lift-label* $\Gamma L\ A,\ A$)
then show $\exists\, l.\ \Gamma L \vdash L\ (l,\ A)$
by *blast*
qed

corollary *framework-adequacy-iff*:
fixes ΓL :: $('l,\ 'f)$ *lfm list*
and A :: $'f$ *fm*
shows $(\exists\, l.\ \Gamma L \vdash L\ (l,\ A)) \longleftrightarrow$ *erase-ctx* $\Gamma L \vdash A$
proof
assume $\exists\, l.\ \Gamma L \vdash L\ (l,\ A)$
then show *erase-ctx* $\Gamma L \vdash A$
using *framework-adequacy* **by** *blast*
next
assume *ordinary*: *erase-ctx* $\Gamma L \vdash A$
then have $\Gamma L \vdash L$ (*lift-label* $\Gamma L\ A,\ A$)
by (*rule lift-coherence*)
with *ordinary* **show** $\exists\, l.\ \Gamma L \vdash L\ (l,\ A)$
using *framework-adequacy* **by** *blast*
qed

end

end

theory *Lifting*
imports *Label-Algebra*
begin

This theory supplies a concrete datatype of proof labels and interprets the abstract label-structure locale with its constructors. With these labels, ordinary derivations can always be decorated to obtain a labelled derivation over any labelled context whose erasure is the ordinary context.

datatype *label* =
Hyp nat

```
  | App label label
  | Lam label label
  | Pair label label
  | FstL label
  | SndL label
  | Inl label
  | Inr label
  | Cases label label label label label
  | Abort label

interpretation default-labels: label-structure
  where app = App
    and lam = Lam
    and pair = Pair
    and fst-l = FstL
    and snd-l = SndL
    and abort = Abort
    and inl = Inl
    and inr = Inr
    and cases = Cases
  .

definition default-erase-label :: label ⇒ unit where
  default-erase-label l = ()

definition default-lift-label :: (label, 'a) lfm list ⇒ 'a fm ⇒ label where
  default-lift-label Γ A =
    (SOME l. default-labels.lderives Γ (l, A))

interpretation default-labels: label-algebra
  where app = App
    and lam = Lam
    and pair = Pair
    and fst-l = FstL
    and snd-l = SndL
    and abort = Abort
    and inl = Inl
    and inr = Inr
    and cases = Cases
    and lderives = default-labels.lderives
    and erase-label = default-erase-label
    and lift-label = default-lift-label
proof
  fix l
  show default-erase-label l = ()
    by (simp add: default-erase-label-def)
next
  fix Γ :: (label, 'a) lfm list and x :: (label, 'a) lfm
  assume default-labels.lderives Γ x
```

```
  then show erase-ctx Γ ⊢ erase-lfm x
    by (rule default-labels.erasure-sound)
next
  fix Γ :: (label, 'a) lfm list and A :: 'a fm
  assume ordinary: erase-ctx Γ ⊢ A
  have ∃ l. default-labels.lderives Γ (l, A)
    by (rule default-labels.lifting-general[OF ordinary])
  then show default-labels.lderives Γ (default-lift-label Γ A, A)
    unfolding default-lift-label-def by (rule someI-ex)
qed

lemma erase-ctx-mem-ex:
  assumes A ∈ set (erase-ctx Γ)
  shows ∃ l. (l, A) ∈ set Γ
  using assms
  by (induction Γ) (auto simp: erase-ctx-def)

lemma erase-ctx-mem-witness:
  assumes A ∈ set (erase-ctx Γ)
  obtains l where (l, A) ∈ set Γ
  using erase-ctx-mem-ex[OF assms] by blast

lemma ordinary-derivations-have-labelling-aux:
  assumes Δ ⊢ A and erase-ctx Γ = Δ
  shows ∃ l. default-labels.lderives Γ (l, A)
  using assms
proof (induction arbitrary: Γ rule: derives.induct)
  case (Assm A Δ)
  then have A ∈ set (erase-ctx Γ)
    by simp
  then obtain l where (l, A) ∈ set Γ
    by (rule erase-ctx-mem-witness)
  then have default-labels.lderives Γ (l, A)
    by (rule default-labels.LAssm)
  then show ?case
    by blast
next
  case (BotE Δ A)
  then obtain l where default-labels.lderives Γ (l, Bot)
    by blast
  then have default-labels.lderives Γ (Abort l, A)
    by (rule default-labels.LBotE)
  then show ?case
    by blast
next
  case (ImpI A Δ B)
  have erase-ctx ((Hyp 0, A) # Γ) = A # Δ
    using ImpI.prems by simp
  then obtain m where default-labels.lderives ((Hyp 0, A) # Γ) (m, B)
```

```
    using ImpI.IH by blast
  then have default-labels.lderives Γ (Lam (Hyp 0) m, Imp A B)
    by (rule default-labels.LImpI)
  then show ?case
    by blast
next
  case (ImpE Δ A B)
  then obtain l where imp: default-labels.lderives Γ (l, Imp A B)
    by blast
  from ImpE obtain m where arg: default-labels.lderives Γ (m, A)
    by blast
  from imp arg have default-labels.lderives Γ (App l m, B)
    by (rule default-labels.LImpE)
  then show ?case
    by blast
next
  case (ConI Δ A B)
  then obtain l where left: default-labels.lderives Γ (l, A)
    by blast
  from ConI obtain m where right: default-labels.lderives Γ (m, B)
    by blast
  from left right have default-labels.lderives Γ (Pair l m, Con A B)
    by (rule default-labels.LConI)
  then show ?case
    by blast
next
  case (ConE1 Δ A B)
  then obtain l where default-labels.lderives Γ (l, Con A B)
    by blast
  then have default-labels.lderives Γ (FstL l, A)
    by (rule default-labels.LConE1)
  then show ?case
    by blast
next
  case (ConE2 Δ A B)
  then obtain l where default-labels.lderives Γ (l, Con A B)
    by blast
  then have default-labels.lderives Γ (SndL l, B)
    by (rule default-labels.LConE2)
  then show ?case
    by blast
next
  case (DisI1 Δ A B)
  then obtain l where default-labels.lderives Γ (l, A)
    by blast
  then have default-labels.lderives Γ (Inl l, Dis A B)
    by (rule default-labels.LDisI1)
  then show ?case
    by blast
```

next
case (*DisI2* Δ *B A*)
then obtain *l* **where** *default-labels.lderives* Γ (*l*, *B*)
by *blast*
then have *default-labels.lderives* Γ (*Inr l*, *Dis A B*)
by (*rule default-labels.LDisI2*)
then show *?case*
by *blast*
next
case (*DisE* Δ *A B C*)
then obtain *l* **where** *disj*: *default-labels.lderives* Γ (*l*, *Dis A B*)
by *blast*
have *left-ctx*: *erase-ctx* ((*Hyp 0*, *A*) # Γ) = *A* # Δ
using *DisE.prems* **by** *simp*
then obtain *n* **where** *left*: *default-labels.lderives* ((*Hyp 0*, *A*) # Γ) (*n*, *C*)
using *DisE.IH*(*2*) **by** *blast*
have *right-ctx*: *erase-ctx* ((*Hyp 1*, *B*) # Γ) = *B* # Δ
using *DisE.prems* **by** *simp*
then obtain *q* **where** *right*: *default-labels.lderives* ((*Hyp 1*, *B*) # Γ) (*q*, *C*)
using *DisE.IH*(*3*) **by** *blast*
from *disj left right* **have** *default-labels.lderives* Γ (*Cases l* (*Hyp 0*) *n* (*Hyp 1*) *q*, *C*)
by (*rule default-labels.LDisE*)
then show *?case*
by *blast*
qed

theorem *ordinary-derivations-have-labelling*:
assumes *erase-ctx* Γ ⊢ *A*
shows ∃ *l*. *default-labels.lderives* Γ (*l*, *A*)
proof −
have *default-labels.lderives* Γ (*default-lift-label* Γ *A*, *A*)
using *assms* **by** (*rule default-labels.lift-coherence*)
with *assms* **show** *?thesis*
using *default-labels.framework-adequacy* **by** *blast*
qed

theorem *ordinary-derivations-admit-labelling*:
assumes *erase-ctx* Γ ⊢ *A*
shows ∃ *l*. *default-labels.lderives* Γ (*l*, *A*)
using *assms ordinary-derivations-have-labelling* **by** *blast*

end

theory *Unit-Labels*
imports *Label-Algebra*
begin

This theory instantiates the labelled kernel with a singleton label type.

Since all label constructors return the unique label, labelled derivability over the lifted context is equivalent to ordinary natural deduction.

datatype *unit-label* = *Star*

interpretation *unit-labels*: *label-structure*
  where *app* = λ- -. *Star*
    and *lam* = λ- -. *Star*
    and *pair* = λ- -. *Star*
    and *fst-l* = λ-. *Star*
    and *snd-l* = λ-. *Star*
    and *abort* = λ-. *Star*
    and *inl* = λ-. *Star*
    and *inr* = λ-. *Star*
    and *cases* = λ- - - - -. *Star*
  .

definition *unit-erase-label* :: *unit-label* ⇒ *unit* **where**
  unit-erase-label l = ()

definition *unit-lift-label* :: (*unit-label*, *'a*) *lfm list* ⇒ *'a fm* ⇒ *unit-label* **where**
  unit-lift-label Γ *A* = *Star*

interpretation *unit-labels*: *label-algebra*
  where *app* = λ- -. *Star*
    and *lam* = λ- -. *Star*
    and *pair* = λ- -. *Star*
    and *fst-l* = λ-. *Star*
    and *snd-l* = λ-. *Star*
    and *abort* = λ-. *Star*
    and *inl* = λ-. *Star*
    and *inr* = λ-. *Star*
    and *cases* = λ- - - - -. *Star*
    and *lderives* = *unit-labels.lderives*
    and *erase-label* = *unit-erase-label*
    and *lift-label* = *unit-lift-label*
proof
  fix *l*
  show *unit-erase-label l* = ()
    by (*simp add*: *unit-erase-label-def*)
next
  fix Γ :: (*unit-label*, *'a*) *lfm list* **and** *x* :: (*unit-label*, *'a*) *lfm*
  assume *unit-labels.lderives* Γ *x*
  then show *erase-ctx* Γ ⊢ *erase-lfm x*
    by (*rule unit-labels.erasure-sound*)
next
  fix Γ :: (*unit-label*, *'a*) *lfm list* **and** *A* :: *'a fm*
  assume *ordinary*: *erase-ctx* Γ ⊢ *A*
  then obtain *l* **where** *deriv*: *unit-labels.lderives* Γ (*l*, *A*)
    using *unit-labels.lifting-general* **by** *blast*

```
  then show unit-labels.lderives Γ (unit-lift-label Γ A, A)
    by (cases l) (simp add: unit-lift-label-def)
qed

definition unit-lift :: 'a fm list ⇒ (unit-label × 'a fm) list where
  unit-lift Γ = map (λA. (Star, A)) Γ

lemma erase-ctx-unit-lift [simp]:
  erase-ctx (unit-lift Γ) = Γ
  by (simp add: unit-lift-def erase-ctx-def comp-def)

lemma unit-labels-from-ordinary:
  assumes Γ ⊢ A
  shows unit-labels.lderives (unit-lift Γ) (Star, A)
  using assms
proof (induction rule: derives.induct)
  case (Assm A Γ)
  then have (Star, A) ∈ set (unit-lift Γ)
    by (auto simp: unit-lift-def)
  then show ?case
    by (rule unit-labels.LAssm)
next
  case (BotE Γ A)
  have unit-labels.lderives (unit-lift Γ) ((λ-. Star) Star, A)
    using BotE.IH by (rule unit-labels.LBotE)
  then show ?case
    by simp
next
  case (ImpI A Γ B)
  have unit-labels.lderives ((Star, A) # unit-lift Γ) (Star, B)
    using ImpI.IH by (simp add: unit-lift-def)
  then show ?case
    by (rule unit-labels.LImpI)
next
  case (ImpE Γ A B)
  have unit-labels.lderives (unit-lift Γ) ((λ- -. Star) Star Star, B)
    using ImpE.IH by (rule unit-labels.LImpE)
  then show ?case
    by simp
next
  case (ConI Γ A B)
  have unit-labels.lderives (unit-lift Γ) ((λ- -. Star) Star Star, Con A B)
    using ConI.IH by (rule unit-labels.LConI)
  then show ?case
    by simp
next
  case (ConE1 Γ A B)
  have unit-labels.lderives (unit-lift Γ) ((λ-. Star) Star, A)
    using ConE1.IH by (rule unit-labels.LConE1)
```

```
  then show ?case
    by simp
next
  case (ConE2 Γ A B)
  have unit-labels.lderives (unit-lift Γ) ((λ-. Star) Star, B)
    using ConE2.IH by (rule unit-labels.LConE2)
  then show ?case
    by simp
next
  case (DisI1 Γ A B)
  have unit-labels.lderives (unit-lift Γ) ((λ-. Star) Star, Dis A B)
    using DisI1.IH by (rule unit-labels.LDisI1)
  then show ?case
    by simp
next
  case (DisI2 Γ B A)
  have unit-labels.lderives (unit-lift Γ) ((λ-. Star) Star, Dis A B)
    using DisI2.IH by (rule unit-labels.LDisI2)
  then show ?case
    by simp
next
  case (DisE Γ A B C)
  have disj: unit-labels.lderives (unit-lift Γ) (Star, Dis A B)
    by (rule DisE.IH(1))
  have left: unit-labels.lderives ((Star, A) # unit-lift Γ) (Star, C)
    using DisE.IH(2) by (simp add: unit-lift-def)
  have right: unit-labels.lderives ((Star, B) # unit-lift Γ) (Star, C)
    using DisE.IH(3) by (simp add: unit-lift-def)
  have unit-labels.lderives (unit-lift Γ) ((λ- - - - -. Star) Star Star Star Star Star,
C)
    using disj left right by (rule unit-labels.LDisE)
  then show ?case
    by simp
qed

theorem unit-labels-conservative:
  Γ ⊢ A ⟷ unit-labels.lderives (unit-lift Γ) (Star, A)
proof
  assume ordinary: Γ ⊢ A
  then have ordinary-erased: erase-ctx (unit-lift Γ) ⊢ A
    by simp
  have unit-labels.lderives
      (unit-lift Γ) (unit-lift-label (unit-lift Γ) A, A)
    using ordinary-erased by (rule unit-labels.lift-coherence)
  with ordinary-erased have ∃ l. unit-labels.lderives (unit-lift Γ) (l, A)
    using unit-labels.framework-adequacy by blast
  then obtain l where unit-labels.lderives (unit-lift Γ) (l, A)
    by blast
  then show unit-labels.lderives (unit-lift Γ) (Star, A)
```

by (*cases l*) *simp*
next
assume *unit-labels.lderives* (*unit-lift* Γ) (*Star*, *A*)
then have ∃ *l*. *unit-labels.lderives* (*unit-lift* Γ) (*l*, *A*)
by *blast*
then have *erase-ctx* (*unit-lift* Γ) ⊢ *A*
using *unit-labels.framework-adequacy* **by** *blast*
then show Γ ⊢ *A*
by *simp*
qed

end

theory *Provenance-Labels*
imports *Label-Algebra*
begin

This theory instantiates labels by sets of natural-number assumption indices. Application and pairing combine provenance by union, implication introduction removes the discharged assumption label, and disjunction elimination records the analysed disjunction together with the branch dependencies that survive discharge.

type-synonym *prov* = *nat set*

interpretation *provenance*: *label-structure*
where *app* = λ(*S*::*prov*) *T*. *S* ∪ *T*
and *lam* = λ(*S*::*prov*) *T*. *T* − *S*
and *pair* = λ(*S*::*prov*) *T*. *S* ∪ *T*
and *fst-l* = λ(*S*::*prov*). *S*
and *snd-l* = λ(*S*::*prov*). *S*
and *abort* = λ(*S*::*prov*). *S*
and *inl* = λ(*S*::*prov*). *S*
and *inr* = λ(*S*::*prov*). *S*
and *cases* = λ(*S*::*prov*) (*M*::*prov*) (*N*::*prov*) (*P*::*prov*) (*Q*::*prov*).
S ∪ (*N* − *M*) ∪ (*Q* − *P*)
.

definition *provenance-erase-label* :: *prov* ⇒ *unit* **where**
provenance-erase-label S = ()

definition *provenance-lift-label* :: (*prov*, *'a*) *lfm list* ⇒ *'a fm* ⇒ *prov* **where**
provenance-lift-label Γ *A* =
(*SOME S*. *provenance.lderives* Γ (*S*, *A*))

interpretation *provenance*: *label-algebra*
where *app* = λ(*S*::*prov*) *T*. *S* ∪ *T*
and *lam* = λ(*S*::*prov*) *T*. *T* − *S*
and *pair* = λ(*S*::*prov*) *T*. *S* ∪ *T*

```
    and fst-l = λ(S::prov). S
    and snd-l = λ(S::prov). S
    and abort = λ(S::prov). S
    and inl = λ(S::prov). S
    and inr = λ(S::prov). S
    and cases = λ(S::prov) (M::prov) (N::prov) (P::prov) (Q::prov).
      S ∪ (N − M) ∪ (Q − P)
    and lderives = provenance.lderives
    and erase-label = provenance-erase-label
    and lift-label = provenance-lift-label
proof
  fix S :: prov
  show provenance-erase-label S = ()
    by (simp add: provenance-erase-label-def)
next
  fix Γ :: (prov, 'a) lfm list and x :: (prov, 'a) lfm
  assume provenance.lderives Γ x
  then show erase-ctx Γ ⊢ erase-lfm x
    by (rule provenance.erasure-sound)
next
  fix Γ :: (prov, 'a) lfm list and A :: 'a fm
  assume ordinary: erase-ctx Γ ⊢ A
  have ∃ S. provenance.lderives Γ (S, A)
    by (rule provenance.lifting-general[OF ordinary])
  then show provenance.lderives Γ (provenance-lift-label Γ A, A)
    unfolding provenance-lift-label-def by (rule someI-ex)
qed

fun indexed :: nat ⇒ 'a fm list ⇒ (prov × 'a fm) list where
  indexed n [] = []
| indexed n (A # Γ) = ({n}, A) # indexed (Suc n) Γ

definition ctx-prov :: (prov, 'a) lfm list ⇒ prov where
  ctx-prov Γ = ⋃(fst ‘ set Γ)

lemma ctx-prov-Cons [simp]:
  ctx-prov ((M, A) # Γ) = M ∪ ctx-prov Γ
  unfolding ctx-prov-def
  by auto

lemma erase-ctx-indexed [simp]:
  erase-ctx (indexed n Γ) = Γ
  by (induction Γ arbitrary: n) auto

lemma finite-ctx-prov-indexed [simp]:
  finite (ctx-prov (indexed n Γ))
  by (induction Γ arbitrary: n) (auto simp: ctx-prov-def)

definition filter-indexed-by :: nat set ⇒ 'a fm list ⇒ 'a fm list where
```

filter-indexed-by $S\ \Gamma =$
map snd (*filter* ($\lambda(i,\ A).\ i \in S$) (*zip* [*0*..<*length* Γ] Γ))

lemma *filter-indexed-by-mono*:
assumes $S \subseteq T$
shows *set* (*filter-indexed-by* $S\ \Gamma$) $\subseteq$ *set* (*filter-indexed-by* $T\ \Gamma$)
using *assms*
unfolding *filter-indexed-by-def*
by *auto*

lemma *filter-indexed-by-subset-ctx*:
shows *set* (*filter-indexed-by* $S\ \Gamma$) $\subseteq$ *set* Γ
unfolding *filter-indexed-by-def*
by (*auto simp*: *in-set-zip*)

lemma *filter-indexed-by-empty* [*simp*]:
filter-indexed-by $\{\}\ \Gamma = []$
unfolding *filter-indexed-by-def*
by *auto*

lemma *filter-indexed-by-union*:
set (*filter-indexed-by* $(S \cup T)\ \Gamma$) =
set (*filter-indexed-by* $S\ \Gamma$) $\cup$ *set* (*filter-indexed-by* $T\ \Gamma$)
unfolding *filter-indexed-by-def*
by *auto*

definition *filter-labeled-by* :: *prov* $\Rightarrow$ (*prov* $\times$ *'a fm*) *list* $\Rightarrow$ *'a fm list* **where**
filter-labeled-by $S\ \Gamma =$
map snd (*filter* ($\lambda(M,\ A).\ M \cap S \neq \{\}$) Γ)

definition *nonempty-prov-ctx* :: (*prov* $\times$ *'a fm*) *list* $\Rightarrow$ *bool* **where**
nonempty-prov-ctx $\Gamma \longleftrightarrow (\forall\, x \in$ *set* $\Gamma.$ *fst* $x \neq \{\})$

lemma *nonempty-prov-ctx-Cons* [*simp*]:
nonempty-prov-ctx $((S,\ A)\ \#\ \Gamma) \longleftrightarrow S \neq \{\} \wedge$ *nonempty-prov-ctx* Γ
unfolding *nonempty-prov-ctx-def*
by *auto*

lemma *nonempty-prov-ctx-indexed* [*simp*]:
nonempty-prov-ctx (*indexed* $n\ \Gamma$)
by (*induction* Γ *arbitrary*: n) (*auto simp*: *nonempty-prov-ctx-def*)

lemma *filter-labeled-by-mono*:
assumes $S \subseteq T$
shows *set* (*filter-labeled-by* $S\ \Gamma$) $\subseteq$ *set* (*filter-labeled-by* $T\ \Gamma$)
using *assms*
unfolding *filter-labeled-by-def*
by *auto*

lemma *filter-labeled-by-union-left*:
  set (*filter-labeled-by* S Γ) $\subseteq$ *set* (*filter-labeled-by* ($S \cup T$) Γ)
  by (*rule filter-labeled-by-mono*) *auto*

lemma *filter-labeled-by-union-right*:
  set (*filter-labeled-by* T Γ) $\subseteq$ *set* (*filter-labeled-by* ($S \cup T$) Γ)
  by (*rule filter-labeled-by-mono*) *auto*

lemma *ctx-prov-contains-label*:
  assumes (M, A) $\in$ *set* Γ
  shows $M \subseteq$ *ctx-prov* Γ
  using *assms*
  unfolding *ctx-prov-def*
  by *auto*

lemma *filter-labeled-by-minus-fresh*:
  assumes $M \cap$ *ctx-prov* $\Gamma = \{\}$
  shows *set* (*filter-labeled-by* N Γ) $\subseteq$ *set* (*filter-labeled-by* ($N - M$) Γ)
proof
  fix A
  assume $A \in$ *set* (*filter-labeled-by* N Γ)
  then obtain y **where** *y-in*: $y \in$ *set* Γ
    and *y-used*: (*case* y *of* (L, B) $\Rightarrow$ $L \cap N \neq \{\}$)
    and *y-snd*: *snd* $y = A$
    unfolding *filter-labeled-by-def*
    by *auto*
  obtain L B **where** y: $y = (L, B)$
    by (*cases* y)
  have *in-ctx*: (L, A) $\in$ *set* Γ
    using y *y-in y-snd* **by** *auto*
  have *used*: $L \cap N \neq \{\}$
    using y *y-used* **by** *simp*
  have $L \cap M = \{\}$
    using *assms ctx-prov-contains-label*[*OF in-ctx*] **by** *auto*
  with *used* **have** $L \cap (N - M) \neq \{\}$
    by *auto*
  with *in-ctx* **have** (L, A) $\in$
      set (*filter* (λ(K, B). $K \cap (N - M) \neq \{\}$) Γ)
    by *auto*
  then show $A \in$ *set* (*filter-labeled-by* ($N - M$) Γ)
    unfolding *filter-labeled-by-def* **by** *force*
qed

lemma *filter-labeled-by-cons-minus-subset*:
  assumes $M \cap$ *ctx-prov* $\Gamma = \{\}$
  shows *set* (*filter-labeled-by* N ((M, A) # Γ)) $\subseteq$
    set (A # *filter-labeled-by* ($N - M$) Γ)
proof −
  have *set* (*filter-labeled-by* N ((M, A) # Γ)) $\subseteq$

```
    insert A (set (filter-labeled-by N Γ))
   unfolding filter-labeled-by-def
   by auto
 also have ... ⊆ set (A # filter-labeled-by (N − M) Γ)
   using filter-labeled-by-minus-fresh[OF assms] by auto
 finally show ?thesis .
qed

lemma filter-labeled-by-cons-minus-mono-subset:
 assumes M ∩ ctx-prov Γ = {} and N − M ⊆ U
 shows set (filter-labeled-by N ((M, A) # Γ)) ⊆
   set (A # filter-labeled-by U Γ)
proof −
 have set (filter-labeled-by N ((M, A) # Γ)) ⊆
     set (A # filter-labeled-by (N − M) Γ)
   using assms(1) by (rule filter-labeled-by-cons-minus-subset)
 moreover have set (filter-labeled-by (N − M) Γ) ⊆
     set (filter-labeled-by U Γ)
   using assms(2) by (rule filter-labeled-by-mono)
 ultimately show ?thesis
   by auto
qed

lemma filter-labeled-by-indexed-upt:
 filter-labeled-by S (indexed n Γ) =
   map snd (filter (λ(i, A). i ∈ S) (zip [n..<n + length Γ] Γ))
proof (induction Γ arbitrary: n)
 case Nil
 then show ?case
   by (simp add: filter-labeled-by-def)
next
 case (Cons A Γ)
 have upt: [n..<n + length (A # Γ)] =
     n # [Suc n..<Suc n + length Γ]
 proof −
   have [n..<n + length (A # Γ)] = n # [Suc n..<n + length (A # Γ)]
     by (rule upt-conv-Cons) simp
   also have ... = n # [Suc n..<Suc n + length Γ]
     by simp
   finally show ?thesis .
 qed
 show ?case
   using Cons.IH[of Suc n] upt
   by (auto simp: filter-labeled-by-def)
qed

lemma filter-labeled-by-indexed-0:
 filter-labeled-by S (indexed 0 Γ) = filter-indexed-by S Γ
 by (simp add: filter-labeled-by-indexed-upt filter-indexed-by-def)
```

inductive *prov-safe* :: (*prov* × *'a fm*) *list* ⇒ *prov* × *'a fm* ⇒ *bool*
(- ⊢*P* - [*50*, *50*] *50*)
where
PAssm:
$x \in set\ \Gamma \Longrightarrow \Gamma \vdash P\ x$
| *PBotE*:
$\Gamma \vdash P\ (S, Bot) \Longrightarrow \Gamma \vdash P\ (S, A)$
| *PImpI*:
$(M, A)\ \#\ \Gamma \vdash P\ (N, B) \Longrightarrow$
$M \cap ctx\text{-}prov\ \Gamma = \{\} \Longrightarrow$
$M \neq \{\} \Longrightarrow$
$\Gamma \vdash P\ (N - M, Imp\ A\ B)$
| *PImpE*:
$\Gamma \vdash P\ (S, Imp\ A\ B) \Longrightarrow$
$\Gamma \vdash P\ (T, A) \Longrightarrow$
$\Gamma \vdash P\ (S \cup T, B)$
| *PConI*:
$\Gamma \vdash P\ (S, A) \Longrightarrow$
$\Gamma \vdash P\ (T, B) \Longrightarrow$
$\Gamma \vdash P\ (S \cup T, Con\ A\ B)$
| *PConE1*:
$\Gamma \vdash P\ (S, Con\ A\ B) \Longrightarrow \Gamma \vdash P\ (S, A)$
| *PConE2*:
$\Gamma \vdash P\ (S, Con\ A\ B) \Longrightarrow \Gamma \vdash P\ (S, B)$
| *PDisI1*:
$\Gamma \vdash P\ (S, A) \Longrightarrow \Gamma \vdash P\ (S, Dis\ A\ B)$
| *PDisI2*:
$\Gamma \vdash P\ (S, B) \Longrightarrow \Gamma \vdash P\ (S, Dis\ A\ B)$
| *PDisE*:
$\Gamma \vdash P\ (S, Dis\ A\ B) \Longrightarrow$
$(M, A)\ \#\ \Gamma \vdash P\ (N, C) \Longrightarrow$
$(P, B)\ \#\ \Gamma \vdash P\ (Q, C) \Longrightarrow$
$M \cap ctx\text{-}prov\ \Gamma = \{\} \Longrightarrow M \neq \{\} \Longrightarrow$
$P \cap ctx\text{-}prov\ \Gamma = \{\} \Longrightarrow P \neq \{\} \Longrightarrow$
$M \cap P = \{\} \Longrightarrow$
$\Gamma \vdash P\ (S \cup (N - M) \cup (Q - P), C)$

lemma *provenance-labels-in-context*:
assumes *provenance.lderives* Γ *x*
shows *fst x* ⊆ *ctx-prov* Γ
using *assms*
by (*induction rule*: *provenance.lderives.induct*) (*auto simp*: *ctx-prov-def*)

lemma *ctx-prov-indexed-bound*:
assumes *i* ∈ *ctx-prov* (*indexed n* Γ)
shows *i* < *n* + *length* Γ
using *assms*
proof (*induction* Γ *arbitrary*: *n i*)

```
  case Nil
  then show ?case
    by (simp add: ctx-prov-def)
next
  case (Cons A Γ)
  then consider i = n | i ∈ ctx-prov (indexed (Suc n) Γ)
    by (auto simp: ctx-prov-def)
  then show ?case
  proof cases
    case 1
    then show ?thesis
      by simp
  next
    case 2
    then have i < Suc n + length Γ
      by (rule Cons.IH)
    then show ?thesis
      by simp
  qed
qed

theorem provenance-overapproximates-dependencies:
  assumes provenance.lderives (indexed 0 Γ0) (S, A)
  shows ∀ i. i ∈ S ⟶ i < length Γ0
proof
  fix i
  show i ∈ S ⟶ i < length Γ0
  proof
    assume i ∈ S
    moreover have S ⊆ ctx-prov (indexed 0 Γ0)
      using provenance-labels-in-context[OF assms] by simp
    ultimately have i ∈ ctx-prov (indexed 0 Γ0)
      by auto
    then show i < length Γ0
      using ctx-prov-indexed-bound[of i 0 Γ0] by simp
  qed
qed

lemma prov-safe-implies-lderives:
  assumes Γ ⊢P x
  shows provenance.lderives Γ x
  using assms
  by (induction rule: prov-safe.induct)
    (auto intro: provenance.LAssm provenance.LBotE provenance.LImpI
     provenance.LImpE provenance.LConI provenance.LConE1 provenance.LConE2
      provenance.LDisI1 provenance.LDisI2 provenance.LDisE)

lemma provenance-drop-unused-general:
  assumes Γ ⊢P x and nonempty-prov-ctx Γ
```

```
  shows filter-labeled-by (fst x) Γ ⊢ snd x
  using assms
proof (induction rule: prov-safe.induct)
  case (PAssm x Γ)
  obtain M A where x: x = (M, A)
    by (cases x)
  then have in-ctx: (M, A) ∈ set Γ
    using PAssm by simp
  moreover have M ≠ {}
    using PAssm x
    unfolding nonempty-prov-ctx-def filter-labeled-by-def
    by auto
  then have M ∩ M ≠ {}
    by auto
  ultimately have (M, A) ∈
      set (filter (λ(L, B). L ∩ M ≠ {}) Γ)
    by auto
  then have A ∈ set (filter-labeled-by M Γ)
    unfolding filter-labeled-by-def by force
  then show ?case
    using x by (auto intro: derives.Assm)
next
  case (PBotE Γ S A)
  then show ?case
    by (auto intro: derives.BotE)
next
  case (PImpI M A Γ N B)
  have ne: nonempty-prov-ctx ((M, A) # Γ)
    using PImpI by simp
  have body: filter-labeled-by N ((M, A) # Γ) ⊢ B
    using PImpI.IH[OF ne] by simp
  have sub: set (filter-labeled-by N ((M, A) # Γ)) ⊆
      set (A # filter-labeled-by (N − M) Γ)
    using PImpI by (intro filter-labeled-by-cons-minus-subset) auto
  have filter-labeled-by (N − M) Γ ⊢ Imp A B
  proof (rule derives.ImpI)
    show A # filter-labeled-by (N − M) Γ ⊢ B
      by (rule weakening[OF body sub])
  qed
  then show ?case
    by simp
next
  case (PImpE Γ S A B T)
  have imp0: filter-labeled-by S Γ ⊢ Imp A B
    using PImpE.IH(1)[OF PImpE.prems] by simp
  have arg0: filter-labeled-by T Γ ⊢ A
    using PImpE.IH(2)[OF PImpE.prems] by simp
  have imp: filter-labeled-by (S ∪ T) Γ ⊢ Imp A B
    by (rule weakening[OF imp0 filter-labeled-by-union-left])
```

```
  have arg: filter-labeled-by (S ∪ T) Γ ⊢ A
    by (rule weakening[OF arg0 filter-labeled-by-union-right])
  have filter-labeled-by (S ∪ T) Γ ⊢ B
    by (rule derives.ImpE[OF imp arg])
  then show ?case
    by simp
next
  case (PConI Γ S A T B)
  have left0: filter-labeled-by S Γ ⊢ A
    using PConI.IH(1)[OF PConI.prems] by simp
  have right0: filter-labeled-by T Γ ⊢ B
    using PConI.IH(2)[OF PConI.prems] by simp
  have left: filter-labeled-by (S ∪ T) Γ ⊢ A
    by (rule weakening[OF left0 filter-labeled-by-union-left])
  have right: filter-labeled-by (S ∪ T) Γ ⊢ B
    by (rule weakening[OF right0 filter-labeled-by-union-right])
  have filter-labeled-by (S ∪ T) Γ ⊢ Con A B
    by (rule derives.ConI[OF left right])
  then show ?case
    by simp
next
  case (PConE1 Γ S A B)
  then show ?case
    by (auto intro: derives.ConE1)
next
  case (PConE2 Γ S A B)
  then show ?case
    by (auto intro: derives.ConE2)
next
  case (PDisI1 Γ S A B)
  then show ?case
    by (auto intro: derives.DisI1)
next
  case (PDisI2 Γ S B A)
  then show ?case
    by (auto intro: derives.DisI2)
next
  case (PDisE Γ S A B M N C P Q)
  let ?U = S ∪ (N − M) ∪ (Q − P)
  have ne-left: nonempty-prov-ctx ((M, A) # Γ)
    using PDisE by simp
  have ne-right: nonempty-prov-ctx ((P, B) # Γ)
    using PDisE by simp
  have disj0: filter-labeled-by S Γ ⊢ Dis A B
    using PDisE.IH(1)[OF PDisE.prems] by simp
  have left0: filter-labeled-by N ((M, A) # Γ) ⊢ C
    using PDisE.IH(2)[OF ne-left] by simp
  have right0: filter-labeled-by Q ((P, B) # Γ) ⊢ C
    using PDisE.IH(3)[OF ne-right] by simp
```

```
  have disj-sub: set (filter-labeled-by S Γ) ⊆
      set (filter-labeled-by ?U Γ)
    by (rule filter-labeled-by-mono) auto
  have disj: filter-labeled-by ?U Γ ⊢ Dis A B
    by (rule weakening[OF disj0 disj-sub])
  have left-sub: set (filter-labeled-by N ((M, A) # Γ)) ⊆
      set (A # filter-labeled-by ?U Γ)
    using PDisE
    by (intro filter-labeled-by-cons-minus-mono-subset) auto
  have left: A # filter-labeled-by ?U Γ ⊢ C
    by (rule weakening[OF left0 left-sub])
  have right-sub: set (filter-labeled-by Q ((P, B) # Γ)) ⊆
      set (B # filter-labeled-by ?U Γ)
    using PDisE
    by (intro filter-labeled-by-cons-minus-mono-subset) auto
  have right: B # filter-labeled-by ?U Γ ⊢ C
    by (rule weakening[OF right0 right-sub])
  have filter-labeled-by ?U Γ ⊢ C
    by (rule derives.DisE[OF disj left right])
  then show ?case
    by simp
qed

theorem provenance-drop-unused:
  assumes indexed 0 Γ0 ⊢P (S, A)
  shows filter-indexed-by S Γ0 ⊢ A
proof −
  have filter-labeled-by S (indexed 0 Γ0) ⊢ A
    using provenance-drop-unused-general[OF assms] by simp
  then show ?thesis
    by (simp add: filter-labeled-by-indexed-0)
qed

lemma fresh-nat:
  assumes finite F
  obtains i :: nat where i ∉ F
proof −
  have ∃ i :: nat. i ∉ F
    using assms ex-new-if-finite[OF infinite-UNIV-nat] by blast
  then show ?thesis
    using that by blast
qed

lemma fresh-nat2:
  assumes finite F
  obtains i j :: nat where i ∉ F and j ∉ F and i ≠ j
proof −
  obtain i where i: i ∉ F
    using assms by (rule fresh-nat)
```

```
  from assms have finite (insert i F)
    by simp
  then obtain j where j: j ∉ insert i F
    by (rule fresh-nat)
  from j have j ∉ F and i ≠ j
    by auto
  with i show ?thesis
    by (rule that)
qed

lemma ordinary-admits-safe-provenance-labeled:
  assumes Δ ⊢ A
    and erase-ctx Γ = Δ
    and nonempty-prov-ctx Γ
    and finite (ctx-prov Γ)
  shows ∃ S. Γ ⊢P (S, A) ∧ S ⊆ ctx-prov Γ
  using assms
proof (induction arbitrary: Γ rule: derives.induct)
  case (Assm A Δ)
  then have A ∈ set (erase-ctx Γ)
    by simp
  then obtain S where in-ctx: (S, A) ∈ set Γ
    by (rule erase-ctx-mem-label)
  have Γ ⊢P (S, A)
    using in-ctx by (rule prov-safe.PAssm)
  moreover have S ⊆ ctx-prov Γ
    using in-ctx by (rule ctx-prov-contains-label)
  ultimately show ?case
    by blast
next
  case (BotE Δ A)
  then obtain S where deriv: Γ ⊢P (S, Bot)
    and sub: S ⊆ ctx-prov Γ
    by blast
  have Γ ⊢P (S, A)
    by (rule prov-safe.PBotE[OF deriv])
  with sub show ?case
    by blast
next
  case (ImpI A Δ B)
  obtain i where fresh: i ∉ ctx-prov Γ
    using ImpI.prems(3) by (rule fresh-nat)
  let ?M = {i}
  have erase: erase-ctx ((?M, A) # Γ) = A # Δ
    using ImpI.prems(1) by simp
  have nonempty: nonempty-prov-ctx ((?M, A) # Γ)
    using ImpI.prems(2) by simp
  have finite: finite (ctx-prov ((?M, A) # Γ))
    using ImpI.prems(3) by simp
```

```
  obtain N where body: (?M, A) # Γ ⊢P (N, B)
    and sub-body: N ⊆ ctx-prov ((?M, A) # Γ)
    using ImpI.IH[OF erase nonempty finite] by blast
  have deriv: Γ ⊢P (N − ?M, Imp A B)
    by (rule prov-safe.PImpI[OF body]) (use fresh in auto)
  have N − ?M ⊆ ctx-prov Γ
    using sub-body fresh by auto
  with deriv show ?case
    by blast
next
  case (ImpE Δ A B)
  obtain S where imp: Γ ⊢P (S, Imp A B)
    and sub-imp: S ⊆ ctx-prov Γ
    using ImpE.IH(1)[OF ImpE.prems] by blast
  obtain T where arg: Γ ⊢P (T, A)
    and sub-arg: T ⊆ ctx-prov Γ
    using ImpE.IH(2)[OF ImpE.prems] by blast
  have Γ ⊢P (S ∪ T, B)
    by (rule prov-safe.PImpE[OF imp arg])
  moreover have S ∪ T ⊆ ctx-prov Γ
    using sub-imp sub-arg by auto
  ultimately show ?case
    by blast
next
  case (ConI Δ A B)
  obtain S where left: Γ ⊢P (S, A)
    and sub-left: S ⊆ ctx-prov Γ
    using ConI.IH(1)[OF ConI.prems] by blast
  obtain T where right: Γ ⊢P (T, B)
    and sub-right: T ⊆ ctx-prov Γ
    using ConI.IH(2)[OF ConI.prems] by blast
  have Γ ⊢P (S ∪ T, Con A B)
    by (rule prov-safe.PConI[OF left right])
  moreover have S ∪ T ⊆ ctx-prov Γ
    using sub-left sub-right by auto
  ultimately show ?case
    by blast
next
  case (ConE1 Δ A B)
  then obtain S where deriv: Γ ⊢P (S, Con A B)
    and sub: S ⊆ ctx-prov Γ
    by blast
  have Γ ⊢P (S, A)
    by (rule prov-safe.PConE1[OF deriv])
  with sub show ?case
    by blast
next
  case (ConE2 Δ A B)
  then obtain S where deriv: Γ ⊢P (S, Con A B)
```

```
    and sub: S ⊆ ctx-prov Γ
    by blast
  have Γ ⊢P (S, B)
    by (rule prov-safe.PConE2[OF deriv])
  with sub show ?case
    by blast
next
  case (DisI1 Δ A B)
  then obtain S where deriv: Γ ⊢P (S, A)
    and sub: S ⊆ ctx-prov Γ
    by blast
  have Γ ⊢P (S, Dis A B)
    by (rule prov-safe.PDisI1[OF deriv])
  with sub show ?case
    by blast
next
  case (DisI2 Δ B A)
  then obtain S where deriv: Γ ⊢P (S, B)
    and sub: S ⊆ ctx-prov Γ
    by blast
  have Γ ⊢P (S, Dis A B)
    by (rule prov-safe.PDisI2[OF deriv])
  with sub show ?case
    by blast
next
  case (DisE Δ A B C)
  obtain i j where fresh-i: i ∉ ctx-prov Γ
    and fresh-j: j ∉ ctx-prov Γ
    and neq: i ≠ j
    using DisE.prems(3) by (rule fresh-nat2)
  let ?M = {i}
  let ?P = {j}
  obtain S where disj: Γ ⊢P (S, Dis A B)
    and sub-disj: S ⊆ ctx-prov Γ
    using DisE.IH(1)[OF DisE.prems] by blast
  have erase-left: erase-ctx ((?M, A) # Γ) = A # Δ
    using DisE.prems(1) by simp
  have nonempty-left: nonempty-prov-ctx ((?M, A) # Γ)
    using DisE.prems(2) by simp
  have finite-left: finite (ctx-prov ((?M, A) # Γ))
    using DisE.prems(3) by simp
  obtain N where left: (?M, A) # Γ ⊢P (N, C)
    and sub-left: N ⊆ ctx-prov ((?M, A) # Γ)
    using DisE.IH(2)[OF erase-left nonempty-left finite-left] by blast
  have erase-right: erase-ctx ((?P, B) # Γ) = B # Δ
    using DisE.prems(1) by simp
  have nonempty-right: nonempty-prov-ctx ((?P, B) # Γ)
    using DisE.prems(2) by simp
  have finite-right: finite (ctx-prov ((?P, B) # Γ))
```

```
    using DisE.prems(3) by simp
  obtain Q where right: (?P, B) # Γ ⊢P (Q, C)
    and sub-right: Q ⊆ ctx-prov ((?P, B) # Γ)
    using DisE.IH(3)[OF erase-right nonempty-right finite-right] by blast
  have deriv: Γ ⊢P (S ∪ (N − ?M) ∪ (Q − ?P), C)
    by (rule prov-safe.PDisE[OF disj left right]) (use fresh-i fresh-j neq in auto)
  have S ∪ (N − ?M) ∪ (Q − ?P) ⊆ ctx-prov Γ
    using sub-disj sub-left sub-right fresh-i fresh-j by auto
  with deriv show ?case
    by blast
qed

theorem ordinary-admits-safe-provenance:
  assumes Γ ⊢ A
  shows ∃S. (indexed 0 Γ ⊢P (S, A))
          ∧ S ⊆ {0..<length Γ}
          ∧ set (filter-indexed-by S Γ) ⊆ set Γ
proof −
  obtain S where deriv: indexed 0 Γ ⊢P (S, A)
    and sub-ctx: S ⊆ ctx-prov (indexed 0 Γ)
    using ordinary-admits-safe-provenance-labeled[OF assms, of indexed 0 Γ]
    by simp blast
  have sub-indices: S ⊆ {0..<length Γ}
  proof
    fix i
    assume i ∈ S
    then have i ∈ ctx-prov (indexed 0 Γ)
      using sub-ctx by auto
    then have i < length Γ
      using ctx-prov-indexed-bound[of i 0 Γ] by simp
    then show i ∈ {0..<length Γ}
      by simp
  qed
  have set (filter-indexed-by S Γ) ⊆ set Γ
    by (rule filter-indexed-by-subset-ctx)
  with deriv sub-indices show ?thesis
    by blast
qed

corollary ordinary-admits-tight-provenance:
  assumes Γ ⊢ A
  shows ∃S. (indexed 0 Γ ⊢P (S, A))
          ∧ S ⊆ {0..<length Γ}
          ∧ filter-indexed-by S Γ ⊢ A
proof −
  obtain S where deriv: indexed 0 Γ ⊢P (S, A)
    and sub: S ⊆ {0..<length Γ}
    using ordinary-admits-safe-provenance[OF assms] by blast
  have filter-indexed-by S Γ ⊢ A
```

by (*rule provenance-drop-unused*[*OF deriv*])
with *deriv sub* **show** *?thesis*
by *blast*
qed

theorem *provenance-completeness*:
shows $(\Gamma \vdash A) \longleftrightarrow$
$(\exists S.\ (\mathit{indexed}\ 0\ \Gamma \vdash P\ (S,\ A)) \wedge \mathit{filter\text{-}indexed\text{-}by}\ S\ \Gamma \vdash A)$
proof
assume $\Gamma \vdash A$
then show $\exists S.\ (\mathit{indexed}\ 0\ \Gamma \vdash P\ (S,\ A)) \wedge \mathit{filter\text{-}indexed\text{-}by}\ S\ \Gamma \vdash A$
using *ordinary-admits-tight-provenance* **by** *blast*
next
assume $\exists S.\ (\mathit{indexed}\ 0\ \Gamma \vdash P\ (S,\ A)) \wedge \mathit{filter\text{-}indexed\text{-}by}\ S\ \Gamma \vdash A$
then obtain S **where** *deriv*: $\mathit{indexed}\ 0\ \Gamma \vdash P\ (S,\ A)$
by *blast*
have $\mathit{filter\text{-}indexed\text{-}by}\ S\ \Gamma \vdash A$
by (*rule provenance-drop-unused*[*OF deriv*])
moreover have $\mathit{set}\ (\mathit{filter\text{-}indexed\text{-}by}\ S\ \Gamma) \subseteq \mathit{set}\ \Gamma$
by (*rule filter-indexed-by-subset-ctx*)
ultimately show $\Gamma \vdash A$
by (*rule weakening*)
qed

end

theory *Modal-Labels-Example*
imports *Label-Algebra*
begin

This theory instantiates the framework locale as *modal-labels* and then extends the resulting propositional labelled calculus with modal *BoxI* and *BoxE* rules. Modal labels carry possible-world annotations; the propositional constructors preserve the current world, while box elimination projects to an accessible world.

datatype *modal-label* =
MWorld nat
| *MApp modal-label modal-label*
| *MLam modal-label modal-label*
| *MPair modal-label modal-label*
| *MFst modal-label*
| *MSnd modal-label*
| *MInl modal-label*
| *MInr modal-label*
| *MCases modal-label modal-label modal-label modal-label modal-label*
| *MAbort modal-label*
| *MBoxI modal-label*
| *MBoxE modal-label*

fun *world-of* :: *modal-label* ⇒ *nat* **where**
  world-of (*MWorld w*) = *w*
| *world-of* (*MApp l m*) = *world-of l*
| *world-of* (*MLam l m*) = *world-of m*
| *world-of* (*MPair l m*) = *world-of l*
| *world-of* (*MFst l*) = *world-of l*
| *world-of* (*MSnd l*) = *world-of l*
| *world-of* (*MInl l*) = *world-of l*
| *world-of* (*MInr l*) = *world-of l*
| *world-of* (*MCases l m n p q*) = *world-of l*
| *world-of* (*MAbort l*) = *world-of l*
| *world-of* (*MBoxI l*) = *world-of l*
| *world-of* (*MBoxE l*) = *world-of l*

interpretation *modal-labels*: *label-structure*
  where *app* = *MApp*
    and *lam* = *MLam*
    and *pair* = *MPair*
    and *fst-l* = *MFst*
    and *snd-l* = *MSnd*
    and *abort* = *MAbort*
    and *inl* = *MInl*
    and *inr* = *MInr*
    and *cases* = *MCases*
  .

definition *modal-erase-label* :: *modal-label* ⇒ *unit* **where**
  modal-erase-label l = ()

definition *modal-lift-label* :: (*modal-label*, *'a*) *lfm list* ⇒ *'a fm* ⇒ *modal-label*
where
  modal-lift-label Γ *A* =
    (*SOME l*. *modal-labels.lderives* Γ (*l*, *A*))

interpretation *modal-labels*: *label-algebra*
  where *app* = *MApp*
    and *lam* = *MLam*
    and *pair* = *MPair*
    and *fst-l* = *MFst*
    and *snd-l* = *MSnd*
    and *abort* = *MAbort*
    and *inl* = *MInl*
    and *inr* = *MInr*
    and *cases* = *MCases*
    and *lderives* = *modal-labels.lderives*
    and *erase-label* = *modal-erase-label*
    and *lift-label* = *modal-lift-label*
proof

```
  fix l
  show modal-erase-label l = ()
    by (simp add: modal-erase-label-def)
next
  fix Γ :: (modal-label, 'a) lfm list and x :: (modal-label, 'a) lfm
  assume modal-labels.lderives Γ x
  then show erase-ctx Γ ⊢ erase-lfm x
    by (rule modal-labels.erasure-sound)
next
  fix Γ :: (modal-label, 'a) lfm list and A :: 'a fm
  assume ordinary: erase-ctx Γ ⊢ A
  have ∃ l. modal-labels.lderives Γ (l, A)
    by (rule modal-labels.lifting-general[OF ordinary])
  then show modal-labels.lderives Γ (modal-lift-label Γ A, A)
    unfolding modal-lift-label-def by (rule someI-ex)
qed

datatype 'a mfm =
    MAtom 'a
  | MBot
  | MImp 'a mfm 'a mfm
  | MCon 'a mfm 'a mfm
  | MDis 'a mfm 'a mfm
  | Box 'a mfm

primrec modalise-fm :: 'a fm ⇒ 'a mfm where
  modalise-fm (Atom p) = MAtom p
| modalise-fm Bot = MBot
| modalise-fm (Imp A B) = MImp (modalise-fm A) (modalise-fm B)
| modalise-fm (Con A B) = MCon (modalise-fm A) (modalise-fm B)
| modalise-fm (Dis A B) = MDis (modalise-fm A) (modalise-fm B)

definition modalise-lfm :: nat ⇒ (modal-label, 'a) lfm ⇒ modal-label × 'a mfm
where
  modalise-lfm w x = (MWorld w, modalise-fm (snd x))

primrec modal-kripke-sat ::
    (nat ⇒ 'a ⇒ bool) ⇒ (nat ⇒ nat ⇒ bool) ⇒ nat ⇒ 'a mfm ⇒ bool
where
  modal-kripke-sat V R w (MAtom p) = V w p
| modal-kripke-sat V R w MBot = False
| modal-kripke-sat V R w (MImp A B) =
    (modal-kripke-sat V R w A ⟶ modal-kripke-sat V R w B)
| modal-kripke-sat V R w (MCon A B) =
    (modal-kripke-sat V R w A ∧ modal-kripke-sat V R w B)
| modal-kripke-sat V R w (MDis A B) =
    (modal-kripke-sat V R w A ∨ modal-kripke-sat V R w B)
| modal-kripke-sat V R w (Box A) =
    (∀ v. R w v ⟶ modal-kripke-sat V R v A)
```

inductive *mlderives* ::
  (*modal-label* × *'a mfm*) *list* ⇒
   (*nat* ⇒ *nat* ⇒ *bool*) ⇒ (*modal-label* × *'a mfm*) ⇒ *bool*
where
  MAssm: *x* ∈ *set* Γ ⟹ *mlderives* Γ *R x*
| *MBotE*: *mlderives* Γ *R* (*l*, *MBot*) ⟹ *mlderives* Γ *R* (*MAbort l*, *A*)
| *MImpI*:
  world-of l = *world-of m* ⟹
   mlderives ((*l*, *A*) # Γ) *R* (*m*, *B*) ⟹
   mlderives Γ *R* (*MLam l m*, *MImp A B*)
| *MImpE*:
  world-of l = *world-of m* ⟹
   mlderives Γ *R* (*l*, *MImp A B*) ⟹
   mlderives Γ *R* (*m*, *A*) ⟹
   mlderives Γ *R* (*MApp l m*, *B*)
| *MConI*:
  world-of l = *world-of m* ⟹
   mlderives Γ *R* (*l*, *A*) ⟹
   mlderives Γ *R* (*m*, *B*) ⟹
   mlderives Γ *R* (*MPair l m*, *MCon A B*)
| *MConE1*: *mlderives* Γ *R* (*l*, *MCon A B*) ⟹ *mlderives* Γ *R* (*MFst l*, *A*)
| *MConE2*: *mlderives* Γ *R* (*l*, *MCon A B*) ⟹ *mlderives* Γ *R* (*MSnd l*, *B*)
| *MDisI1*: *mlderives* Γ *R* (*l*, *A*) ⟹ *mlderives* Γ *R* (*MInl l*, *MDis A B*)
| *MDisI2*: *mlderives* Γ *R* (*l*, *B*) ⟹ *mlderives* Γ *R* (*MInr l*, *MDis A B*)
| *MDisE*:
  world-of l = *world-of m* ⟹
   world-of l = *world-of n* ⟹
   world-of l = *world-of p* ⟹
   world-of l = *world-of q* ⟹
   mlderives Γ *R* (*l*, *MDis A B*) ⟹
   mlderives ((*m*, *A*) # Γ) *R* (*n*, *C*) ⟹
   mlderives ((*p*, *B*) # Γ) *R* (*q*, *C*) ⟹
   mlderives Γ *R* (*MCases l m n p q*, *C*)
| *MBoxI*:
  (⋀*v*. *R* (*world-of l*) *v* ⟹
    ∃ *m*. *world-of m* = *v* ∧ *mlderives* Γ *R* (*m*, *A*)) ⟹
   mlderives Γ *R* (*MBoxI l*, *Box A*)
| *MBoxE*:
  mlderives Γ *R* (*l*, *Box A*) ⟹
   R (*world-of l*) *v* ⟹
   mlderives Γ *R* (*MBoxE* (*MWorld v*), *A*)

theorem *propositional-modal-sound*:
  assumes *modal-labels.lderives* Γ *x*
  shows ∃ *l*. *world-of l* = *w* ∧
   mlderives (*map* (*modalise-lfm w*) Γ) *R* (*l*, *modalise-fm* (*snd x*))
  using *assms*
proof (*induction arbitrary*: *w R rule*: *modal-labels.lderives.induct*)

```
  case (LAssm x Γ)
  then show ?case
    by (intro exI[of - MWorld w])
      (auto simp: modalise-lfm-def intro: mlderives.MAssm)
next
  case (LBotE Γ l A)
  then obtain k where k: world-of k = w
      mlderives (map (modalise-lfm w) Γ) R (k, MBot)
    by auto
  then show ?case
    by (intro exI[of - MAbort k]) (auto intro: mlderives.MBotE)
next
  case (LImpI l A Γ m B)
  then obtain k where k: world-of k = w
      mlderives (map (modalise-lfm w) ((l, A) # Γ)) R (k, modalise-fm B)
    by auto
  then have body: mlderives ((MWorld w, modalise-fm A) #
      map (modalise-lfm w) Γ) R (k, modalise-fm B)
    by (simp add: modalise-lfm-def)
  then have mlderives (map (modalise-lfm w) Γ) R
      (MLam (MWorld w) k, MImp (modalise-fm A) (modalise-fm B))
    using k(1) by (auto intro: mlderives.MImpI)
  then show ?case
    by (intro exI[of - MLam (MWorld w) k]) (simp add: k(1))
next
  case (LImpE Γ l A B m)
  from LImpE.IH(1)[of w R] obtain k where k: world-of k = w
    mlderives (map (modalise-lfm w) Γ) R (k, MImp (modalise-fm A) (modalise-fm
B))
    by auto
  from LImpE.IH(2)[of w R] obtain h where h: world-of h = w
      mlderives (map (modalise-lfm w) Γ) R (h, modalise-fm A)
    by auto
  then have mlderives (map (modalise-lfm w) Γ) R (MApp k h, modalise-fm B)
    using k by (auto intro: mlderives.MImpE)
  then show ?case
    by (intro exI[of - MApp k h]) (simp add: k h)
next
  case (LConI Γ l A m B)
  from LConI.IH(1)[of w R] obtain k where k: world-of k = w
      mlderives (map (modalise-lfm w) Γ) R (k, modalise-fm A)
    by auto
  from LConI.IH(2)[of w R] obtain h where h: world-of h = w
      mlderives (map (modalise-lfm w) Γ) R (h, modalise-fm B)
    by auto
  then have mlderives (map (modalise-lfm w) Γ) R
      (MPair k h, MCon (modalise-fm A) (modalise-fm B))
    using k by (auto intro: mlderives.MConI)
  then show ?case
```

```
    by (intro exI[of - MPair k h]) (simp add: k h)
next
  case (LConE1 Γ l A B)
  then obtain k where k: world-of k = w
     mlderives (map (modalise-lfm w) Γ) R (k, MCon (modalise-fm A) (modalise-fm
B))
    by auto
  then show ?case
    by (intro exI[of - MFst k]) (auto intro: mlderives.MConE1)
next
  case (LConE2 Γ l A B)
  then obtain k where k: world-of k = w
     mlderives (map (modalise-lfm w) Γ) R (k, MCon (modalise-fm A) (modalise-fm
B))
    by auto
  then show ?case
    by (intro exI[of - MSnd k]) (auto intro: mlderives.MConE2)
next
  case (LDisI1 Γ l A B)
  then obtain k where k: world-of k = w
      mlderives (map (modalise-lfm w) Γ) R (k, modalise-fm A)
    by auto
  then show ?case
    by (intro exI[of - MInl k]) (auto intro: mlderives.MDisI1)
next
  case (LDisI2 Γ l B A)
  then obtain k where k: world-of k = w
      mlderives (map (modalise-lfm w) Γ) R (k, modalise-fm B)
    by auto
  then show ?case
    by (intro exI[of - MInr k]) (auto intro: mlderives.MDisI2)
next
  case (LDisE Γ l A B m n C p q)
  from LDisE.IH(1)[of w R] obtain d where d:
      world-of d = w
     mlderives (map (modalise-lfm w) Γ) R (d, MDis (modalise-fm A) (modalise-fm
B))
    by auto
  from LDisE.IH(2)[of w R] obtain left where left:
      world-of left = w
      mlderives (map (modalise-lfm w) ((m, A) # Γ)) R (left, modalise-fm C)
    by auto
  from LDisE.IH(3)[of w R] obtain right where right:
      world-of right = w
      mlderives (map (modalise-lfm w) ((p, B) # Γ)) R (right, modalise-fm C)
    by auto
  from left have left-deriv: mlderives ((MWorld w, modalise-fm A) #
      map (modalise-lfm w) Γ) R (left, modalise-fm C)
    by (simp add: modalise-lfm-def)
```

```
  from right have right-deriv: mlderives ((MWorld w, modalise-fm B) #
      map (modalise-lfm w) Γ) R (right, modalise-fm C)
    by (simp add: modalise-lfm-def)
  have mlderives (map (modalise-lfm w) Γ) R
      (MCases d (MWorld w) left (MWorld w) right, modalise-fm C)
    using d left right left-deriv right-deriv by (auto intro: mlderives.MDisE)
  then show ?case
    by (intro exI[of - MCases d (MWorld w) left (MWorld w) right])
      (simp add: d left right)
qed

theorem modal-labels-sound:
  assumes mlderives Γ R x
    and ∀(l, A) ∈ set Γ. modal-kripke-sat V R (world-of l) A
  shows modal-kripke-sat V R (world-of (fst x)) (snd x)
  using assms
proof (induction arbitrary: V rule: mlderives.induct)
  case (MAssm x Γ R)
  then show ?case
    by (cases x) auto
next
  case (MBotE Γ R l A)
  then show ?case
    by auto
next
  case (MImpI l m A Γ R B)
  then show ?case
    by auto
next
  case (MImpE l m Γ R A B)
  then show ?case
    by auto
next
  case (MConI l m Γ R A B)
  then show ?case
    by auto
next
  case (MConE1 Γ R l A B)
  then show ?case
    by auto
next
  case (MConE2 Γ R l A B)
  then show ?case
    by auto
next
  case (MDisI1 Γ R l A B)
  then show ?case
    by auto
next
```

```
  case (MDisI2 Γ R l B A)
  then show ?case
    by auto
next
  case (MDisE l m n p q Γ R A B C)
  then show ?case
    by auto
next
  case (MBoxI l R Γ A)
  then show ?case
    by auto
next
  case (MBoxE Γ R l A v)
  then show ?case
    by auto
qed

end


theory Modal-K-Schema
  imports Modal-Labels-Example
begin
```

The modal instance derives the normal modal K axiom schema directly in the labelled calculus. The proof is uniform in the formulae, the accessibility relation, and the chosen world; necessitation is available only for formulae that are derivable from the empty context at every world and under every accessibility relation.

```
theorem modal-K-schema:
  fixes A B :: 'a mfm
    and R :: nat ⇒ nat ⇒ bool
    and w :: nat
  shows
    ∃ l. world-of l = w ∧
         mlderives [] R (l, MImp (Box (MImp A B)) (MImp (Box A) (Box B)))
proof −
  let ?imp-assm = MWorld w
  let ?box-assm = MWorld w
  let ?box-label = MBoxI (MWorld w)
  let ?inner-label = MLam ?box-assm ?box-label
  let ?outer-label = MLam ?imp-assm ?inner-label
  let ?K = MImp (Box (MImp A B)) (MImp (Box A) (Box B))
  let ?Γ = [(?box-assm, Box A), (?imp-assm, Box (MImp A B))]

  have box-deriv: mlderives ?Γ R (?box-label, Box B)
  proof (rule mlderives.MBoxI)
    fix v
    assume accessible: R (world-of (MWorld w)) v
```

```
  let ?elim-label = MBoxE (MWorld v)
  have imp-assm-deriv: mlderives ?Γ R (?imp-assm, Box (MImp A B))
    by (rule mlderives.MAssm) simp
  have imp-at-v: mlderives ?Γ R (?elim-label, MImp A B)
  proof (rule mlderives.MBoxE)
    show mlderives ?Γ R (?imp-assm, Box (MImp A B))
      by (rule imp-assm-deriv)
    show R (world-of ?imp-assm) v
      using accessible by simp
  qed
  have box-assm-deriv: mlderives ?Γ R (?box-assm, Box A)
    by (rule mlderives.MAssm) simp
  have A-at-v: mlderives ?Γ R (?elim-label, A)
  proof (rule mlderives.MBoxE)
    show mlderives ?Γ R (?box-assm, Box A)
      by (rule box-assm-deriv)
    show R (world-of ?box-assm) v
      using accessible by simp
  qed
  have B-at-v: mlderives ?Γ R (MApp ?elim-label ?elim-label, B)
  proof (rule mlderives.MImpE)
    show world-of ?elim-label = world-of ?elim-label
      by simp
    show mlderives ?Γ R (?elim-label, MImp A B)
      by (rule imp-at-v)
    show mlderives ?Γ R (?elim-label, A)
      by (rule A-at-v)
  qed
  show ∃ m. world-of m = v ∧ mlderives ?Γ R (m, B)
    using B-at-v by (intro exI[of - MApp ?elim-label ?elim-label]) simp
qed

have inner-deriv:
  mlderives [(?imp-assm, Box (MImp A B))] R
    (?inner-label, MImp (Box A) (Box B))
proof (rule mlderives.MImpI)
  show world-of ?box-assm = world-of ?box-label
    by simp
  show mlderives ((?box-assm, Box A) # [(?imp-assm, Box (MImp A B))]) R
    (?box-label, Box B)
    using box-deriv by simp
qed

have outer-deriv: mlderives [] R (?outer-label, ?K)
proof (rule mlderives.MImpI)
  show world-of ?imp-assm = world-of ?inner-label
    by simp
  show mlderives ((?imp-assm, Box (MImp A B)) # []) R
    (?inner-label, MImp (Box A) (Box B))
```

```
    using inner-deriv by simp
  qed

  show ?thesis
    using outer-deriv by (intro exI[of - ?outer-label]) simp
qed

theorem modal-necessitation:
  fixes A :: 'a mfm
    and R :: nat ⇒ nat ⇒ bool
    and w :: nat
  assumes provable-everywhere:
    ⋀v R'. ∃ l. world-of l = v ∧ mlderives [] R' (l, A)
  shows
    ∃ l. world-of l = w ∧ mlderives [] R (l, Box A)
proof −
  have box-deriv: mlderives [] R (MBoxI (MWorld w), Box A)
  proof (rule mlderives.MBoxI)
    fix v
    assume R (world-of (MWorld w)) v
    from provable-everywhere[of v R]
    show ∃ m. world-of m = v ∧ mlderives [] R (m, A) .
  qed
  show ?thesis
    using box-deriv by (intro exI[of - MBoxI (MWorld w)]) simp
qed

corollary modal-K-kripke-valid:
  fixes A B :: 'a mfm
  shows ∀ V R w. modal-kripke-sat V R w
                 (MImp (Box (MImp A B)) (MImp (Box A) (Box B)))
proof (intro allI)
  fix V R w
  let ?K = MImp (Box (MImp A B)) (MImp (Box A) (Box B))
  obtain l where l-world: world-of l = w
    and deriv: mlderives [] R (l, ?K)
    using modal-K-schema[where A=A and B=B and R=R and w=w] by blast
  have modal-kripke-sat V R (world-of (fst (l, ?K))) (snd (l, ?K))
  proof (rule modal-labels-sound)
    show mlderives [] R (l, ?K)
      by (rule deriv)
    show ∀ (l, A) ∈ set []. modal-kripke-sat V R (world-of l) A
      by simp
  qed
  with l-world show modal-kripke-sat V R w ?K
    by simp
qed

corollary modal-K-example:
```

shows $\exists l.$ *world-of* $l = 0 \wedge$ *mlderives* [] R
(l, *MImp* (*Box* (*MImp* (*MAtom* (*0*::*nat*)) (*MAtom 1*)))
(*MImp* (*Box* (*MAtom 0*)) (*Box* (*MAtom 1*))))
using *modal-K-schema*[
where *A*=*MAtom* (*0*::*nat*) **and** *B*=*MAtom* (*1*::*nat*) **and** *R*=*R* **and** *w*=*0*]
by *simp*

end

theory *Examples*
imports *Provenance-Labels Modal-Labels-Example*
begin

This theory collects small derivations that exercise the labelled framework: provenance-labelled implicational combinators, labelled modus ponens, and a possible-world derivation of the modal K axiom.

lemma *K-axiom-labelled*:
provenance.lderives [] ({}, *Imp A* (*Imp B A*))
proof −
have *a-from-ab*: *provenance.lderives* [({*1*}, *B*), ({*0*}, *A*)] ({*0*}, *A*)
by (*rule provenance.LAssm*) *simp*
have *b-imp-a*: *provenance.lderives* [({*0*}, *A*)] ({*0*}, *Imp B A*)
proof −
have *provenance.lderives* [({*0*}, *A*)] ((λ*S T*. *T* − *S*) {*1*} {*0*}, *Imp B A*)
using *a-from-ab* **by** (*rule provenance.LImpI*)
then show *?thesis*
by *auto*
qed
have *provenance.lderives* [] ((λ*S T*. *T* − *S*) {*0*} {*0*}, *Imp A* (*Imp B A*))
using *b-imp-a* **by** (*rule provenance.LImpI*)
then show *?thesis*
by *auto*
qed

lemma *S-axiom-labelled*:
provenance.lderives []
({}, *Imp* (*Imp A* (*Imp B C*)) (*Imp* (*Imp A B*) (*Imp A C*)))
proof −
let *?F* = *Imp A* (*Imp B C*)
let *?G* = *Imp A B*
let *?ctx* = [({*2*}, *A*), ({*1*}, *?G*), ({*0*}, *?F*)]
have *f*: *provenance.lderives ?ctx* ({*0*}, *?F*)
by (*rule provenance.LAssm*) *simp*
have *g*: *provenance.lderives ?ctx* ({*1*}, *?G*)
by (*rule provenance.LAssm*) *simp*
have *a*: *provenance.lderives ?ctx* ({*2*}, *A*)
by (*rule provenance.LAssm*) *simp*
have *f-applied*: *provenance.lderives ?ctx* ({*0*, *2*}, *Imp B C*)

proof −
have *raw*: *provenance.lderives ?ctx* ((λ*S T*. *S* ∪ *T*) {*0*} {*2*}, *Imp B C*)
using *f a* **by** (*rule provenance.LImpE*)
have ((λ*S T*. *S* ∪ *T*) ({*0*}::*prov*) ({*2*}::*prov*)) = {*0*, *2*}
by *auto*
with *raw* **show** *?thesis*
by *simp*
qed
have *b*: *provenance.lderives ?ctx* ({*1*, *2*}, *B*)
proof −
have *raw*: *provenance.lderives ?ctx* ((λ*S T*. *S* ∪ *T*) {*1*} {*2*}, *B*)
using *g a* **by** (*rule provenance.LImpE*)
have ((λ*S T*. *S* ∪ *T*) ({*1*}::*prov*) ({*2*}::*prov*)) = {*1*, *2*}
by *auto*
with *raw* **show** *?thesis*
by *simp*
qed
have *c*: *provenance.lderives ?ctx* ({*0*, *1*, *2*}, *C*)
proof −
have *raw*: *provenance.lderives ?ctx* ((λ*S T*. *S* ∪ *T*) {*0*, *2*} {*1*, *2*}, *C*)
using *f-applied b* **by** (*rule provenance.LImpE*)
have ((λ*S T*. *S* ∪ *T*) ({*0*, *2*}::*prov*) ({*1*, *2*}::*prov*)) = {*0*, *1*, *2*}
by *auto*
with *raw* **show** *?thesis*
by *simp*
qed
have *imp-a-c*: *provenance.lderives* [({*1*}, *?G*), ({*0*}, *?F*)] ({*0*, *1*}, *Imp A C*)
proof −
have *raw*: *provenance.lderives* [({*1*}, *?G*), ({*0*}, *?F*)]
((λ*S T*. *T* − *S*) {*2*} {*0*, *1*, *2*}, *Imp A C*)
using *c* **by** (*rule provenance.LImpI*)
have ((λ*S T*. *T* − *S*) ({*2*}::*prov*) ({*0*, *1*, *2*}::*prov*)) = {*0*, *1*}
by *auto*
with *raw* **show** *?thesis*
by *simp*
qed
have *imp-g-a-c*: *provenance.lderives* [({*0*}, *?F*)] ({*0*}, *Imp ?G* (*Imp A C*))
proof −
have *raw*: *provenance.lderives* [({*0*}, *?F*)]
((λ*S T*. *T* − *S*) {*1*} {*0*, *1*}, *Imp ?G* (*Imp A C*))
using *imp-a-c* **by** (*rule provenance.LImpI*)
have ((λ*S T*. *T* − *S*) ({*1*}::*prov*) ({*0*, *1*}::*prov*)) = {*0*}
by *auto*
with *raw* **show** *?thesis*
by *simp*
qed
have *raw*: *provenance.lderives* []
((λ*S T*. *T* − *S*) {*0*} {*0*},
Imp ?F (*Imp ?G* (*Imp A C*)))

```
    using imp-g-a-c by (rule provenance.LImpI)
  have ((λS T. T − S) ({0}::prov) ({0}::prov)) = {}
    by auto
  with raw show ?thesis
    by simp
qed

lemma modus-ponens-labelled:
  provenance.lderives [({0}, Imp A B), ({1}, A)] ({0, 1}, B)
proof −
  have imp: provenance.lderives [({0}, Imp A B), ({1}, A)] ({0}, Imp A B)
    by (rule provenance.LAssm) simp
  have arg: provenance.lderives [({0}, Imp A B), ({1}, A)] ({1}, A)
    by (rule provenance.LAssm) simp
  have raw: provenance.lderives [({0}, Imp A B), ({1}, A)]
      ((λS T. S ∪ T) {0} {1}, B)
    using imp arg by (rule provenance.LImpE)
  have ((λS T. S ∪ T) ({0}::prov) ({1}::prov)) = {0, 1}
    by auto
  with raw show ?thesis
    by simp
qed

lemma modal-K-axiom:
  mlderives [] R
    (MLam (MWorld w) (MLam (MWorld w) (MBoxI (MWorld w))),
      MImp (Box (MImp A B)) (MImp (Box A) (Box B)))
proof (rule mlderives.MImpI)
  show world-of (MWorld w) = world-of (MLam (MWorld w) (MBoxI (MWorld
w)))
    by simp
  show mlderives [(MWorld w, Box (MImp A B))] R
      (MLam (MWorld w) (MBoxI (MWorld w)), MImp (Box A) (Box B))
  proof (rule mlderives.MImpI)
    show world-of (MWorld w) = world-of (MBoxI (MWorld w))
      by simp
    show mlderives [(MWorld w, Box A), (MWorld w, Box (MImp A B))] R
        (MBoxI (MWorld w), Box B)
    proof (rule mlderives.MBoxI)
      fix v
      assume access: R (world-of (MWorld w)) v
      have imp: mlderives [(MWorld w, Box A), (MWorld w, Box (MImp A B))] R
          (MBoxE (MWorld v), MImp A B)
        by (meson MAssm MBoxE access list.set-intros(1,2))
      have arg: mlderives [(MWorld w, Box A), (MWorld w, Box (MImp A B))] R
          (MBoxE (MWorld v), A)
        by (meson MAssm MBoxE access list.set-intros(1))
      have result: mlderives [(MWorld w, Box A), (MWorld w, Box (MImp A B))]
R
```

```
      (MApp (MBoxE (MWorld v)) (MBoxE (MWorld v)), B)
    by (rule mlderives.MImpE[OF - imp arg]) simp
  show ∃ m. world-of m = v ∧
      mlderives [(MWorld w, Box A), (MWorld w, Box (MImp A B))] R (m, B)
    using result
    by (intro exI[of - MApp (MBoxE (MWorld v)) (MBoxE (MWorld v))]) simp
 qed
 qed
qed

end
```

References

[1] D. Basin, S. Matthews, and L. Viganò. Labelled propositional modal logics: theory and practice. *Journal of Logic and Computation*, 7(6):685–717, 1997.

[2] D. M. Gabbay. *Labelled Deductive Systems, Volume 1.* Oxford University Press, 1996.

[3] A. K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic.* PhD thesis, University of Edinburgh, 1994.

[4] L. Viganò. *Labelled Non-Classical Logics.* Kluwer Academic Publishers, 2000.