Linear Temporal Logic

Salomon Sickert

March 17, 2025

Abstract

This theory provides a formalisation of linear temporal logic (LTL) and unifies previous formalisations within the AFP. This entry establishes syntax and semantics for this logic and decouples it from existing entries, yielding a common environment for theories reasoning about LTL. Furthermore a parser written in SML and an executable simplifier are provided.

Contents

1 Linear Temporal Logic

theory *LTL*
imports
Main HOL−Library.Omega-Words-Fun

begin

This theory provides a formalisation of linear temporal logic. It provides three variants:

1. LTL with syntactic sugar. This variant is the semantic reference and the included parser generates ASTs of this datatype.
2. LTL in negation normal form without syntactic sugar. This variant is used by the included rewriting engine and is used for the translation to automata (implemented in other entries).
3. LTL in restricted negation normal form without the rather uncommon operators "weak until" and "strong release". It is used by the formalization of Gerth's algorithm.
4. PLTL. A variant with a reduced set of operators.

This theory subsumes (and partly reuses) the existing formalisation found in LTL_to_GBA and Stuttering_Equivalence and unifies them.

1.1 LTL with Syntactic Sugar

In this section, we provide a formulation of LTL with explicit syntactic sugar deeply embedded. This formalization serves as a reference semantics.

1.1.1 Syntax

datatype (*atoms-ltlc*: *'a*) *ltlc* =
  True-ltlc (‹$true_c$›)
| *False-ltlc* (‹$false_c$›)
| *Prop-ltlc 'a* (‹$prop_c$'(-')›)
| *Not-ltlc 'a ltlc* (‹not_c -› [*85*] *85*)
| *And-ltlc 'a ltlc 'a ltlc* (‹- and_c -› [*82*,*82*] *81*)
| *Or-ltlc 'a ltlc 'a ltlc* (‹- or_c -› [*81*,*81*] *80*)
| *Implies-ltlc 'a ltlc 'a ltlc* (‹- $implies_c$ -› [*81*,*81*] *80*)
| *Next-ltlc 'a ltlc* (‹X_c -› [*88*] *87*)
| *Final-ltlc 'a ltlc* (‹F_c -› [*88*] *87*)
| *Global-ltlc 'a ltlc* (‹G_c -› [*88*] *87*)
| *Until-ltlc 'a ltlc 'a ltlc* (‹- U_c -› [*84*,*84*] *83*)
| *Release-ltlc 'a ltlc 'a ltlc* (‹- R_c -› [*84*,*84*] *83*)
| *WeakUntil-ltlc 'a ltlc 'a ltlc* (‹- W_c -› [*84*,*84*] *83*)
| *StrongRelease-ltlc 'a ltlc 'a ltlc* (‹- M_c -› [*84*,*84*] *83*)

definition *Iff-ltlc* (‹- iff_c -› [*81*,*81*] *80*)
where
  φ iff_c $\psi \equiv (\varphi$ $implies_c$ $\psi)$ and_c $(\psi$ $implies_c$ $\varphi)$

1.1.2 Semantics

primrec *semantics-ltlc* :: [*'a set word*, *'a ltlc*] ⇒ *bool* (‹- ⊨$_c$ -› [*80*,*80*] *80*)
where
  $\xi \models_c true_c = True$
| $\xi \models_c false_c = False$
| $\xi \models_c prop_c(q) = (q{\in}\xi\ 0)$
| $\xi \models_c not_c\ \varphi = (\neg\ \xi \models_c \varphi)$
| $\xi \models_c \varphi\ and_c\ \psi = (\xi \models_c \varphi \wedge \xi \models_c \psi)$
| $\xi \models_c \varphi\ or_c\ \psi = (\xi \models_c \varphi \vee \xi \models_c \psi)$
| $\xi \models_c \varphi\ implies_c\ \psi = (\xi \models_c \varphi \longrightarrow \xi \models_c \psi)$
| $\xi \models_c X_c\ \varphi = (suffix\ 1\ \xi \models_c \varphi)$
| $\xi \models_c F_c\ \varphi = (\exists\, i.\ suffix\ i\ \xi \models_c \varphi)$
| $\xi \models_c G_c\ \varphi = (\forall\, i.\ suffix\ i\ \xi \models_c \varphi)$
| $\xi \models_c \varphi\ U_c\ \psi = (\exists\, i.\ suffix\ i\ \xi \models_c \psi \wedge (\forall\, j{<}i.\ suffix\ j\ \xi \models_c \varphi))$
| $\xi \models_c \varphi\ R_c\ \psi = (\forall\, i.\ suffix\ i\ \xi \models_c \psi \vee (\exists\, j{<}i.\ suffix\ j\ \xi \models_c \varphi))$
| $\xi \models_c \varphi\ W_c\ \psi = (\forall\, i.\ suffix\ i\ \xi \models_c \varphi \vee (\exists\, j{\leq}i.\ suffix\ j\ \xi \models_c \psi))$
| $\xi \models_c \varphi\ M_c\ \psi = (\exists\, i.\ suffix\ i\ \xi \models_c \varphi \wedge (\forall\, j{\leq}i.\ suffix\ j\ \xi \models_c \psi))$

lemma *semantics-ltlc-sugar* [*simp*]:
  $\xi \models_c \varphi\ iff_c\ \psi = (\xi \models_c \varphi \longleftrightarrow \xi \models_c \psi)$
  $\xi \models_c F_c\ \varphi = \xi \models_c (true_c\ U_c\ \varphi)$
  $\xi \models_c G_c\ \varphi = \xi \models_c (false_c\ R_c\ \varphi)$
  by (*auto simp add*: *Iff-ltlc-def*)

definition *language-ltlc* $\varphi \equiv \{\xi.\ \xi \models_c \varphi\}$

lemma *language-ltlc-negate*[*simp*]:
  language-ltlc $(not_c\ \varphi)$ = − *language-ltlc* φ
  unfolding *language-ltlc-def* **by** *auto*

lemma *ltl-true-or-con*[*simp*]:
  $\xi \models_c prop_c(p)\ or_c\ (not_c\ prop_c(p))$
  by *auto*

lemma *ltl-false-true-con*[*simp*]:
  $\xi \models_c not_c\ true_c \longleftrightarrow \xi \models_c false_c$
  by *auto*

lemma *ltl-Next-Neg-con*[*simp*]:
  $\xi \models_c X_c\ (not_c\ \varphi) \longleftrightarrow \xi \models_c not_c\ X_c\ \varphi$
  by *auto*

— The connection between dual operators

lemma *ltl-Until-Release-con*:
$\xi \models_c \varphi\ R_c\ \psi \longleftrightarrow (\neg\ \xi \models_c (not_c\ \varphi)\ U_c\ (not_c\ \psi))$
$\xi \models_c \varphi\ U_c\ \psi \longleftrightarrow (\neg\ \xi \models_c (not_c\ \varphi)\ R_c\ (not_c\ \psi))$
by *auto*

lemma *ltl-WeakUntil-StrongRelease-con*:
$\xi \models_c \varphi\ W_c\ \psi \longleftrightarrow (\neg\ \xi \models_c (not_c\ \varphi)\ M_c\ (not_c\ \psi))$
$\xi \models_c \varphi\ M_c\ \psi \longleftrightarrow (\neg\ \xi \models_c (not_c\ \varphi)\ W_c\ (not_c\ \psi))$
by *auto*

— The connection between weak and strong operators

lemma *ltl-Release-StrongRelease-con*:
$\xi \models_c \varphi\ R_c\ \psi \longleftrightarrow \xi \models_c (\varphi\ M_c\ \psi)\ or_c\ (G_c\ \psi)$
$\xi \models_c \varphi\ M_c\ \psi \longleftrightarrow \xi \models_c (\varphi\ R_c\ \psi)\ and_c\ (F_c\ \varphi)$
proof *safe*
assume *asm*: $\xi \models_c \varphi\ R_c\ \psi$

show $\xi \models_c (\varphi\ M_c\ \psi)\ or_c\ (G_c\ \psi)$
proof (*cases* $\xi \models_c G_c\ \psi$)
case *False*

then obtain i **where** $\neg$ *suffix* $i\ \xi \models_c \psi$ **and** $\forall j{<}i.$ *suffix* $j\ \xi \models_c \psi$
using *exists-least-iff*[*of* $\lambda i. \neg$ *suffix* $i\ \xi \models_c \psi$] **by** *force*

then show *?thesis*
using *asm* **by** *force*
qed *simp*
next
assume *asm*: $\xi \models_c (\varphi\ R_c\ \psi)\ and_c\ (F_c\ \varphi)$

then show $\xi \models_c \varphi\ M_c\ \psi$
proof (*cases* $\xi \models_c F_c\ \varphi$)
case *True*

then obtain i **where** *suffix* $i\ \xi \models_c \varphi$ **and** $\forall j{<}i. \neg$ *suffix* $j\ \xi \models_c \varphi$
using *exists-least-iff*[*of* $\lambda i.$ *suffix* $i\ \xi \models_c \varphi$] **by** *force*

then show *?thesis*
using *asm* **by** *force*
qed *simp*
qed (*unfold semantics-ltlc.simps*; *insert not-less*, *blast*)+

lemma *ltl-Until-WeakUntil-con*:
$\xi \models_c \varphi\ U_c\ \psi \longleftrightarrow \xi \models_c (\varphi\ W_c\ \psi)\ and_c\ (F_c\ \psi)$
$\xi \models_c \varphi\ W_c\ \psi \longleftrightarrow \xi \models_c (\varphi\ U_c\ \psi)\ or_c\ (G_c\ \varphi)$
proof *safe*
assume *asm*: $\xi \models_c (\varphi\ W_c\ \psi)\ and_c\ (F_c\ \psi)$

then show $\xi \models_c \varphi\ U_c\ \psi$
proof (*cases* $\xi \models_c F_c\ \psi$)
case *True*

then obtain *i* **where** *suffix i* $\xi \models_c \psi$ **and** $\forall j<i.\ \neg$ *suffix j* $\xi \models_c \psi$
using *exists-least-iff*[*of* $\lambda i.$ *suffix i* $\xi \models_c \psi$] **by** *force*

then show *?thesis*
using *asm* **by** *force*
qed *simp*
next
assume *asm*: $\xi \models_c \varphi\ W_c\ \psi$

then show $\xi \models_c (\varphi\ U_c\ \psi)\ or_c\ (G_c\ \varphi)$
proof (*cases* $\xi \models_c G_c\ \varphi$)
case *False*

then obtain *i* **where** $\neg$ *suffix i* $\xi \models_c \varphi$ **and** $\forall j<i.$ *suffix j* $\xi \models_c \varphi$
using *exists-least-iff*[*of* $\lambda i.\ \neg$ *suffix i* $\xi \models_c \varphi$] **by** *force*

then show *?thesis*
using *asm* **by** *force*
qed *simp*
qed (*unfold semantics-ltlc.simps*; *insert not-less*, *blast*)+

lemma *ltl-StrongRelease-Until-con*:
$\xi \models_c \varphi\ M_c\ \psi \longleftrightarrow \xi \models_c \psi\ U_c\ (\varphi\ and_c\ \psi)$
using *order.order-iff-strict* **by** *auto*

lemma *ltl-WeakUntil-Release-con*:
$\xi \models_c \varphi\ R_c\ \psi \longleftrightarrow \xi \models_c \psi\ W_c\ (\varphi\ and_c\ \psi)$
by (*meson ltl-Release-StrongRelease-con*(*1*) *ltl-StrongRelease-Until-con ltl-Until-WeakUntil-con*(*2*) *semantics-ltlc.simps*(*6*))

definition *pw-eq-on S w w'* $\equiv \forall i.\ w\ i \cap S = w'\ i \cap S$

lemma *pw-eq-on-refl*[*simp*]: *pw-eq-on S w w*

and *pw-eq-on-sym*: *pw-eq-on S w w′* $\Longrightarrow$ *pw-eq-on S w′ w*
and *pw-eq-on-trans*[*trans*]: ⟦*pw-eq-on S w w′*; *pw-eq-on S w′ w″*⟧ $\Longrightarrow$ *pw-eq-on S w w″*
unfolding *pw-eq-on-def* **by** *auto*

lemma *pw-eq-on-suffix*:
pw-eq-on S w w′ $\Longrightarrow$ *pw-eq-on S* (*suffix k w*) (*suffix k w′*)
by (*simp add*: *pw-eq-on-def*)

lemma *pw-eq-on-subset*:
$S \subseteq S'$ $\Longrightarrow$ *pw-eq-on S′ w w′* $\Longrightarrow$ *pw-eq-on S w w′*
by (*auto simp add*: *pw-eq-on-def*)

lemma *ltlc-eq-on-aux*:
pw-eq-on (*atoms-ltlc* φ) *w w′* $\Longrightarrow$ *w* $\models_c \varphi$ $\Longrightarrow$ *w′* $\models_c \varphi$
proof (*induction* φ *arbitrary*: *w w′*)
case *Until-ltlc*
thus *?case*
by *simp* (*meson Un-upper1 Un-upper2 pw-eq-on-subset pw-eq-on-suffix*)
next
case *Release-ltlc*
thus *?case*
by *simp* (*metis Un-upper1 pw-eq-on-subset pw-eq-on-suffix sup-commute*)
next
case *WeakUntil-ltlc*
thus *?case*
by *simp* (*meson pw-eq-on-subset pw-eq-on-suffix sup.cobounded1 sup-ge2*)
next
case *StrongRelease-ltlc*
thus *?case*
by *simp* (*metis Un-upper1 pw-eq-on-subset pw-eq-on-suffix pw-eq-on-sym sup-ge2*)
next
case (*And-ltlc* φ ψ)
thus *?case*
by *simp* (*meson Un-upper1 inf-sup-ord*(*4*) *pw-eq-on-subset*)
next
case (*Or-ltlc* φ ψ)
thus *?case*
by *simp* (*meson Un-upper2 pw-eq-on-subset sup-ge1*)
next
case (*Implies-ltlc* φ ψ)
thus *?case*
by *simp* (*meson Un-upper1 Un-upper2 pw-eq-on-subset*[*of atoms-ltlc -*

atoms-ltlc $\varphi \cup$ *atoms-ltlc* ψ] *pw-eq-on-sym*)
qed (*auto simp add*: *pw-eq-on-def*; *metis suffix-nth*)+

lemma *ltlc-eq-on*:
pw-eq-on (*atoms-ltlc* φ) $w\ w' \Longrightarrow w \models_c \varphi \longleftrightarrow w' \models_c \varphi$
using *ltlc-eq-on-aux pw-eq-on-sym* **by** *blast*

lemma *suffix-comp*: $(\lambda i.\ f\ (\mathit{suffix}\ k\ w\ i)) = \mathit{suffix}\ k\ (f\ o\ w)$
by *auto*

lemma *suffix-range*: $\bigcup(\mathit{range}\ \xi) \subseteq \mathit{APs} \Longrightarrow \bigcup(\mathit{range}\ (\mathit{suffix}\ k\ \xi)) \subseteq \mathit{APs}$
by *auto*

lemma *map-semantics-ltlc-aux*:
assumes *inj-on f APs*
assumes $\bigcup(\mathit{range}\ w) \subseteq \mathit{APs}$
assumes *atoms-ltlc* $\varphi \subseteq \mathit{APs}$
shows $w \models_c \varphi \longleftrightarrow (\lambda i.\ f\ `\ w\ i) \models_c$ *map-ltlc f* φ
using *assms*(*2*,*3*)
proof (*induction* φ *arbitrary*: w)
case (*Prop-ltlc x*)
thus *?case* **using** *assms*(*1*)
by (*simp add*: *SUP-le-iff inj-on-image-mem-iff*)
next
case (*Next-ltlc* φ)
show *?case*
using *Next-ltlc*(*1*)[*of suffix 1 w*, *unfolded suffix-comp comp-def*] *Next-ltlc*(*2*,*3*)
apply *simp*
by (*metis Next-ltlc.prems*(*1*) *One-nat-def* ‹⟦$\bigcup$(*range* (*suffix 1 w*)) ⊆ *APs*; *atoms-ltlc* $\varphi \subseteq$ *APs*⟧ ⟹ *suffix 1 w* $\models_c \varphi$ = *suffix 1* $(\lambda x.\ f\ `\ w\ x) \models_c$ *map-ltlc f* φ› *suffix-range*)
next
case (*Final-ltlc* φ)
thus *?case*
using *Final-ltlc*(*1*)[*of suffix - -*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
next
case (*Global-ltlc*)
thus *?case*
using *Global-ltlc*(*1*)[*of suffix - w*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
next
case (*Until-ltlc*)
thus *?case*

using *Until-ltlc*(*1*,*2*)[*of suffix - w*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
next
case (*Release-ltlc*)
thus *?case*
using *Release-ltlc*(*1*,*2*)[*of suffix - w*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
next
case (*WeakUntil-ltlc*)
thus *?case*
using *WeakUntil-ltlc*(*1*,*2*)[*of suffix - w*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
next
case (*StrongRelease-ltlc*)
thus *?case*
using *StrongRelease-ltlc*(*1*,*2*)[*of suffix - w*, *unfolded suffix-comp comp-def*, *OF suffix-range*] **by** *fastforce*
qed *simp+*

definition *map-props f APs* ≡ {*i*. ∃ *p*∈*APs*. *f p* = *Some i*}

lemma *map-semantics-ltlc*:
assumes *INJ*: *inj-on f* (*dom f*) **and** *DOM*: *atoms-ltlc* φ ⊆ *dom f*
shows ξ ⊨*c* φ ⟷ (*map-props f o* ξ) ⊨*c map-ltlc* (*the o f*) φ
proof −
let *?*ξ*r* = λ*i*. ξ *i* ∩ *atoms-ltlc* φ
let *?*ξ*r*′ = λ*i*. ξ *i* ∩ *dom f*

have *1*: ⋃(*range ?*ξ*r*) ⊆ *atoms-ltlc* φ **by** *auto*

have *INJ-the-dom*: *inj-on* (*the o f*) (*dom f*)
using *assms*
by (*auto simp*: *inj-on-def domIff*)
note *2* = *subset-inj-on*[*OF this DOM*]

have *3*: (λ*i*. (*the o f*) ‘ *?*ξ*r*′ *i*) = *map-props f o* ξ **using** *DOM INJ*
apply (*auto intro*!: *ext simp*: *map-props-def domIff image-iff*)
by (*metis Int-iff domI option.sel*)

have ξ ⊨*c* φ ⟷ *?*ξ*r* ⊨*c* φ
apply (*rule ltlc-eq-on*)
apply (*auto simp*: *pw-eq-on-def*)
done
also from *map-semantics-ltlc-aux*[*OF 2 1 subset-refl*]

have ... $\longleftrightarrow$ (λi. (*the o f*) ‘ *?ξr i*) $\models_c$ *map-ltlc* (*the o f*) φ **.**
also have ... $\longleftrightarrow$ (λi. (*the o f*) ‘ *?ξr′ i*) $\models_c$ *map-ltlc* (*the o f*) φ
apply (*rule ltlc-eq-on*) **using** *DOM INJ*
apply (*auto simp*: *pw-eq-on-def ltlc.set-map domIff image-iff*)
by (*metis Int-iff contra-subsetD domD domI inj-on-eq-iff option.sel*)
also note *3*
finally show *?thesis* **.**
qed

lemma *map-semantics-ltlc-inv*:
assumes *INJ*: *inj-on f* (*dom f*) **and** *DOM*: *atoms-ltlc* $\varphi \subseteq$ *dom f*
shows $\xi \models_c$ *map-ltlc* (*the o f*) $\varphi \longleftrightarrow$ (λi. (*the o f*) $-$‘ ξ *i*) $\models_c \varphi$
using *map-semantics-ltlc*[*OF assms*]
apply *simp*
apply (*intro ltlc-eq-on*)
apply (*auto simp add*: *pw-eq-on-def ltlc.set-map map-props-def*)
by (*metis DOM comp-apply contra-subsetD domD option.sel vimage-eq*)

1.2 LTL in Negation Normal Form

We define a type of LTL formula in negation normal form (NNF).

1.2.1 Syntax

datatype (*atoms-ltln*: *′a*) *ltln* =
True-ltln (‹*true*$_n$›)
| *False-ltln* (‹*false*$_n$›)
| *Prop-ltln ′a* (‹*prop*$_n$ *′*(-*′*)›)
| *Nprop-ltln ′a* (‹*nprop*$_n$ *′*(-*′*)›)
| *And-ltln ′a ltln ′a ltln* (‹- *and*$_n$ -› [*82*,*82*] *81*)
| *Or-ltln ′a ltln ′a ltln* (‹- *or*$_n$ -› [*84*,*84*] *83*)
| *Next-ltln ′a ltln* (‹*X*$_n$ -› [*88*] *87*)
| *Until-ltln ′a ltln ′a ltln* (‹- *U*$_n$ -› [*84*,*84*] *83*)
| *Release-ltln ′a ltln ′a ltln* (‹- *R*$_n$ -› [*84*,*84*] *83*)
| *WeakUntil-ltln ′a ltln ′a ltln* (‹- *W*$_n$ -› [*84*,*84*] *83*)
| *StrongRelease-ltln ′a ltln ′a ltln* (‹- *M*$_n$ -› [*84*,*84*] *83*)

abbreviation *finally*$_n$ :: *′a ltln* $\Rightarrow$ *′a ltln* (‹*F*$_n$ -› [*88*] *87*)
where
F$_n$ $\varphi \equiv$ *true*$_n$ *U*$_n$ φ

notation (*input*) *finally*$_n$ (‹$\Diamond_n$ -› [*88*] *87*)

abbreviation *globally*$_n$ :: *′a ltln* $\Rightarrow$ *′a ltln* (‹*G*$_n$ -› [*88*] *87*)

where
$G_n\ \varphi \equiv false_n\ R_n\ \varphi$

notation (*input*) $globally_n$ ($\langle\Box_n$ -$\rangle$ [88] 87)

1.2.2 Semantics

primrec *semantics-ltln* :: ['*a set word*, '*a ltln*] ⇒ *bool* ($\langle$- $\models_n$ -$\rangle$ [80,80] 80)
where
$\xi \models_n true_n = True$
| $\xi \models_n false_n = False$
| $\xi \models_n prop_n(q) = (q \in \xi\ 0)$
| $\xi \models_n nprop_n(q) = (q \notin \xi\ 0)$
| $\xi \models_n \varphi\ and_n\ \psi = (\xi \models_n \varphi \wedge \xi \models_n \psi)$
| $\xi \models_n \varphi\ or_n\ \psi = (\xi \models_n \varphi \vee \xi \models_n \psi)$
| $\xi \models_n X_n\ \varphi = (suffix\ 1\ \xi \models_n \varphi)$
| $\xi \models_n \varphi\ U_n\ \psi = (\exists\, i.\ suffix\ i\ \xi \models_n \psi \wedge (\forall\, j{<}i.\ suffix\ j\ \xi \models_n \varphi))$
| $\xi \models_n \varphi\ R_n\ \psi = (\forall\, i.\ suffix\ i\ \xi \models_n \psi \vee (\exists\, j{<}i.\ suffix\ j\ \xi \models_n \varphi))$
| $\xi \models_n \varphi\ W_n\ \psi = (\forall\, i.\ suffix\ i\ \xi \models_n \varphi \vee (\exists\, j{\leq}i.\ suffix\ j\ \xi \models_n \psi))$
| $\xi \models_n \varphi\ M_n\ \psi = (\exists\, i.\ suffix\ i\ \xi \models_n \varphi \wedge (\forall\, j{\leq}i.\ suffix\ j\ \xi \models_n \psi))$

definition *language-ltln* $\varphi \equiv \{\xi.\ \xi \models_n \varphi\}$

lemma *semantics-ltln-ite-simps*[*simp*]:
$w \models_n (if\ P\ then\ true_n\ else\ false_n) = P$
$w \models_n (if\ P\ then\ false_n\ else\ true_n) = (\neg P)$
by *simp-all*

1.2.3 Conversion

fun *ltlc-to-ltln'* :: *bool* ⇒ '*a ltlc* ⇒ '*a ltln*
where
ltlc-to-ltln' False $true_c = true_n$
| *ltlc-to-ltln' False* $false_c = false_n$
| *ltlc-to-ltln' False* $prop_c(q) = prop_n(q)$
| *ltlc-to-ltln' False* ($\varphi\ and_c\ \psi$) = (*ltlc-to-ltln' False* φ) and_n (*ltlc-to-ltln' False* ψ)
| *ltlc-to-ltln' False* ($\varphi\ or_c\ \psi$) = (*ltlc-to-ltln' False* φ) or_n (*ltlc-to-ltln' False* ψ)
| *ltlc-to-ltln' False* ($\varphi\ implies_c\ \psi$) = (*ltlc-to-ltln' True* φ) or_n (*ltlc-to-ltln' False* ψ)
| *ltlc-to-ltln' False* ($F_c\ \varphi$) = $true_n\ U_n$ (*ltlc-to-ltln' False* φ)
| *ltlc-to-ltln' False* ($G_c\ \varphi$) = $false_n\ R_n$ (*ltlc-to-ltln' False* φ)
| *ltlc-to-ltln' False* ($\varphi\ U_c\ \psi$) = (*ltlc-to-ltln' False* φ) U_n (*ltlc-to-ltln' False*

ψ)
| *ltlc-to-ltln'* *False* (φ R_c ψ) = (*ltlc-to-ltln'* *False* φ) R_n (*ltlc-to-ltln'* *False* ψ)
| *ltlc-to-ltln'* *False* (φ W_c ψ) = (*ltlc-to-ltln'* *False* φ) W_n (*ltlc-to-ltln'* *False* ψ)
| *ltlc-to-ltln'* *False* (φ M_c ψ) = (*ltlc-to-ltln'* *False* φ) M_n (*ltlc-to-ltln'* *False* ψ)
| *ltlc-to-ltln'* *True* $true_c$ = $false_n$
| *ltlc-to-ltln'* *True* $false_c$ = $true_n$
| *ltlc-to-ltln'* *True* $prop_c(q)$ = $nprop_n(q)$
| *ltlc-to-ltln'* *True* (φ and_c ψ) = (*ltlc-to-ltln'* *True* φ) or_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (φ or_c ψ) = (*ltlc-to-ltln'* *True* φ) and_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (φ $implies_c$ ψ) = (*ltlc-to-ltln'* *False* φ) and_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (F_c φ) = $false_n$ R_n (*ltlc-to-ltln'* *True* φ)
| *ltlc-to-ltln'* *True* (G_c φ) = $true_n$ U_n (*ltlc-to-ltln'* *True* φ)
| *ltlc-to-ltln'* *True* (φ U_c ψ) = (*ltlc-to-ltln'* *True* φ) R_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (φ R_c ψ) = (*ltlc-to-ltln'* *True* φ) U_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (φ W_c ψ) = (*ltlc-to-ltln'* *True* φ) M_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *True* (φ M_c ψ) = (*ltlc-to-ltln'* *True* φ) W_n (*ltlc-to-ltln'* *True* ψ)
| *ltlc-to-ltln'* *b* (not_c φ) = *ltlc-to-ltln'* ($\neg$ *b*) φ
| *ltlc-to-ltln'* *b* (X_c φ) = X_n (*ltlc-to-ltln'* *b* φ)

fun *ltlc-to-ltln* :: *'a ltlc* ⇒ *'a ltln*
where
ltlc-to-ltln φ = *ltlc-to-ltln'* *False* φ

fun *ltln-to-ltlc* :: *'a ltln* ⇒ *'a ltlc*
where
ltln-to-ltlc $true_n$ = $true_c$
| *ltln-to-ltlc* $false_n$ = $false_c$
| *ltln-to-ltlc* $prop_n(q)$ = $prop_c(q)$
| *ltln-to-ltlc* $nprop_n(q)$ = not_c ($prop_c(q)$)
| *ltln-to-ltlc* (φ and_n ψ) = (*ltln-to-ltlc* φ and_c *ltln-to-ltlc* ψ)
| *ltln-to-ltlc* (φ or_n ψ) = (*ltln-to-ltlc* φ or_c *ltln-to-ltlc* ψ)
| *ltln-to-ltlc* (X_n φ) = (X_c *ltln-to-ltlc* φ)
| *ltln-to-ltlc* (φ U_n ψ) = (*ltln-to-ltlc* φ U_c *ltln-to-ltlc* ψ)
| *ltln-to-ltlc* (φ R_n ψ) = (*ltln-to-ltlc* φ R_c *ltln-to-ltlc* ψ)
| *ltln-to-ltlc* (φ W_n ψ) = (*ltln-to-ltlc* φ W_c *ltln-to-ltlc* ψ)
| *ltln-to-ltlc* (φ M_n ψ) = (*ltln-to-ltlc* φ M_c *ltln-to-ltlc* ψ)

lemma *ltlc-to-ltln′-correct*:
$w \models_n$ (*ltlc-to-ltln′ True* φ) $\longleftrightarrow \neg\ w \models_c \varphi$
$w \models_n$ (*ltlc-to-ltln′ False* φ) $\longleftrightarrow\ w \models_c \varphi$
size (*ltlc-to-ltln′ True* φ) $\leq$ 2 $*$ *size* φ
size (*ltlc-to-ltln′ False* φ) $\leq$ 2 $*$ *size* φ
by (*induction* φ *arbitrary*: w) *simp+*

lemma *ltlc-to-ltln-semantics* [*simp*]:
$w \models_n$ *ltlc-to-ltln* $\varphi \longleftrightarrow\ w \models_c \varphi$
using *ltlc-to-ltln′-correct* **by** *auto*

lemma *ltlc-to-ltln-size*:
size (*ltlc-to-ltln* φ) $\leq$ 2 $*$ *size* φ
using *ltlc-to-ltln′-correct* **by** *simp*

lemma *ltln-to-ltlc-semantics* [*simp*]:
$w \models_c$ *ltln-to-ltlc* $\varphi \longleftrightarrow\ w \models_n \varphi$
by (*induction* φ *arbitrary*: w) *simp+*

lemma *ltlc-to-ltln-atoms*:
atoms-ltln (*ltlc-to-ltln* φ) = *atoms-ltlc* φ
proof −
have *atoms-ltln* (*ltlc-to-ltln′ True* φ) = *atoms-ltlc* φ
atoms-ltln (*ltlc-to-ltln′ False* φ) = *atoms-ltlc* φ
by (*induction* φ) *simp+*
thus *?thesis*
by *simp*
qed

1.2.4 Negation

fun not_n
where
$not_n\ true_n = false_n$
| $not_n\ false_n = true_n$
| $not_n\ prop_n(a) = nprop_n(a)$
| $not_n\ nprop_n(a) = prop_n(a)$
| $not_n\ (\varphi\ and_n\ \psi) = (not_n\ \varphi)\ or_n\ (not_n\ \psi)$
| $not_n\ (\varphi\ or_n\ \psi) = (not_n\ \varphi)\ and_n\ (not_n\ \psi)$
| $not_n\ (X_n\ \varphi) = X_n\ (not_n\ \varphi)$
| $not_n\ (\varphi\ U_n\ \psi) = (not_n\ \varphi)\ R_n\ (not_n\ \psi)$
| $not_n\ (\varphi\ R_n\ \psi) = (not_n\ \varphi)\ U_n\ (not_n\ \psi)$
| $not_n\ (\varphi\ W_n\ \psi) = (not_n\ \varphi)\ M_n\ (not_n\ \psi)$

| $not_n\ (\varphi\ M_n\ \psi) = (not_n\ \varphi)\ W_n\ (not_n\ \psi)$

lemma *not$_n$-semantics*[*simp*]:
$w \models_n not_n\ \varphi \longleftrightarrow \neg\ w \models_n \varphi$
by (*induction* φ *arbitrary*: w) *auto*

lemma *not$_n$-size*:
size $(not_n\ \varphi) =$ *size* φ
by (*induction* φ) *auto*

1.2.5 Subformulas

fun *subfrmlsn* :: *'a ltln* ⇒ *'a ltln set*
where
subfrmlsn $(\varphi\ and_n\ \psi) = \{\varphi\ and_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\varphi\ or_n\ \psi) = \{\varphi\ or_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(X_n\ \varphi) = \{X_n\ \varphi\} \cup$ *subfrmlsn* φ
| *subfrmlsn* $(\varphi\ U_n\ \psi) = \{\varphi\ U_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\varphi\ R_n\ \psi) = \{\varphi\ R_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\varphi\ W_n\ \psi) = \{\varphi\ W_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $(\varphi\ M_n\ \psi) = \{\varphi\ M_n\ \psi\} \cup$ *subfrmlsn* $\varphi \cup$ *subfrmlsn* ψ
| *subfrmlsn* $\varphi = \{\varphi\}$

lemma *subfrmlsn-id*[*simp*]:
$\varphi \in$ *subfrmlsn* φ
by (*induction* φ) *auto*

lemma *subfrmlsn-finite*:
finite (*subfrmlsn* φ)
by (*induction* φ) *auto*

lemma *subfrmlsn-card*:
card (*subfrmlsn* φ) $\leq$ *size* φ
by (*induction* φ) (*simp-all add*: *card-insert-if subfrmlsn-finite*, (*meson add-le-mono card-Un-le dual-order.trans le-SucI*)+)

lemma *subfrmlsn-subset*:
$\psi \in$ *subfrmlsn* $\varphi \Longrightarrow$ *subfrmlsn* $\psi \subseteq$ *subfrmlsn* φ
by (*induction* φ) *auto*

lemma *subfrmlsn-size*:
$\psi \in$ *subfrmlsn* $\varphi \Longrightarrow$ *size* $\psi <$ *size* $\varphi \vee \psi = \varphi$
by (*induction* φ) *auto*

abbreviation
size-set S ≡ *sum* (λ*x*. *2* ∗ *size x* + *1*) *S*

lemma *size-set-diff*:
finite S ⟹ *S'* ⊆ *S* ⟹ *size-set* (*S* − *S'*) = *size-set S* − *size-set S'*
using *sum-diff-nat finite-subset* **by** *metis*

1.2.6 Constant Folding

lemma *U-consts*[*intro*, *simp*]:
w ⊨$_n$ φ *U*$_n$ *true*$_n$
¬ (*w* ⊨$_n$ φ *U*$_n$ *false*$_n$)
(*w* ⊨$_n$ *false*$_n$ *U*$_n$ φ) = (*w* ⊨$_n$ φ)
by *force*+

lemma *R-consts*[*intro*, *simp*]:
w ⊨$_n$ φ *R*$_n$ *true*$_n$
¬ (*w* ⊨$_n$ φ *R*$_n$ *false*$_n$)
(*w* ⊨$_n$ *true*$_n$ *R*$_n$ φ) = (*w* ⊨$_n$ φ)
by *force*+

lemma *W-consts*[*intro*, *simp*]:
w ⊨$_n$ *true*$_n$ *W*$_n$ φ
w ⊨$_n$ φ *W*$_n$ *true*$_n$
(*w* ⊨$_n$ *false*$_n$ *W*$_n$ φ) = (*w* ⊨$_n$ φ)
(*w* ⊨$_n$ φ *W*$_n$ *false*$_n$) = (*w* ⊨$_n$ *G*$_n$ φ)
by *force*+

lemma *M-consts*[*intro*, *simp*]:
¬ (*w* ⊨$_n$ *false*$_n$ *M*$_n$ φ)
¬ (*w* ⊨$_n$ φ *M*$_n$ *false*$_n$)
(*w* ⊨$_n$ *true*$_n$ *M*$_n$ φ) = (*w* ⊨$_n$ φ)
(*w* ⊨$_n$ φ *M*$_n$ *true*$_n$) = (*w* ⊨$_n$ *F*$_n$ φ)
by *force*+

1.2.7 Distributivity

lemma *until-and-left-distrib*:
w ⊨$_n$ (φ$_1$ *and*$_n$ φ$_2$) *U*$_n$ ψ ⟷ *w* ⊨$_n$ (φ$_1$ *U*$_n$ ψ) *and*$_n$ (φ$_2$ *U*$_n$ ψ)
proof
assume *w* ⊨$_n$ φ$_1$ *U*$_n$ ψ *and*$_n$ φ$_2$ *U*$_n$ ψ

then obtain *i1 i2* **where** *suffix i1 w* ⊨$_n$ ψ ∧ (∀ *j*<*i1*. *suffix j w* ⊨$_n$ φ$_1$)
and *suffix i2 w* ⊨$_n$ ψ ∧ (∀ *j*<*i2*. *suffix j w* ⊨$_n$ φ$_2$)

by *auto*

then have *suffix* (*min i1 i2*) $w \models_n \psi \wedge (\forall j{<}\mathit{min}\ i1\ i2.\ \mathit{suffix}\ j\ w \models_n \varphi_1\ \mathit{and}_n\ \varphi_2)$
by (*simp add*: *min-def*)

then show $w \models_n (\varphi_1\ \mathit{and}_n\ \varphi_2)\ U_n\ \psi$
by *force*
qed *auto*

lemma *until-or-right-distrib*:
$w \models_n \varphi\ U_n\ (\psi_1\ \mathit{or}_n\ \psi_2) \longleftrightarrow w \models_n (\varphi\ U_n\ \psi_1)\ \mathit{or}_n\ (\varphi\ U_n\ \psi_2)$
by *auto*

lemma *release-and-right-distrib*:
$w \models_n \varphi\ R_n\ (\psi_1\ \mathit{and}_n\ \psi_2) \longleftrightarrow w \models_n (\varphi\ R_n\ \psi_1)\ \mathit{and}_n\ (\varphi\ R_n\ \psi_2)$
by *auto*

lemma *release-or-left-distrib*:
$w \models_n (\varphi_1\ \mathit{or}_n\ \varphi_2)\ R_n\ \psi \longleftrightarrow w \models_n (\varphi_1\ R_n\ \psi)\ \mathit{or}_n\ (\varphi_2\ R_n\ \psi)$
by (*metis* not_n*.simps*(6) not_n*.simps*(9) not_n*-semantics until-and-left-distrib*)

lemma *strong-release-and-right-distrib*:
$w \models_n \varphi\ M_n\ (\psi_1\ \mathit{and}_n\ \psi_2) \longleftrightarrow w \models_n (\varphi\ M_n\ \psi_1)\ \mathit{and}_n\ (\varphi\ M_n\ \psi_2)$
proof
assume $w \models_n (\varphi\ M_n\ \psi_1)\ \mathit{and}_n\ (\varphi\ M_n\ \psi_2)$

then obtain *i1 i2* **where** *suffix i1* $w \models_n \varphi \wedge (\forall j{\leq}i1.\ \mathit{suffix}\ j\ w \models_n \psi_1)$ **and** *suffix i2* $w \models_n \varphi \wedge (\forall j{\leq}i2.\ \mathit{suffix}\ j\ w \models_n \psi_2)$
by *auto*

then have *suffix* (*min i1 i2*) $w \models_n \varphi \wedge (\forall j{\leq}\mathit{min}\ i1\ i2.\ \mathit{suffix}\ j\ w \models_n \psi_1\ \mathit{and}_n\ \psi_2)$
by (*simp add*: *min-def*)

then show $w \models_n \varphi\ M_n\ (\psi_1\ \mathit{and}_n\ \psi_2)$
by *force*
qed *auto*

lemma *strong-release-or-left-distrib*:
$w \models_n (\varphi_1\ \mathit{or}_n\ \varphi_2)\ M_n\ \psi \longleftrightarrow w \models_n (\varphi_1\ M_n\ \psi)\ \mathit{or}_n\ (\varphi_2\ M_n\ \psi)$
by *auto*

lemma *weak-until-and-left-distrib*:

$w \models_n (\varphi_1\ and_n\ \varphi_2)\ W_n\ \psi \longleftrightarrow w \models_n (\varphi_1\ W_n\ \psi)\ and_n\ (\varphi_2\ W_n\ \psi)$
by *auto*

lemma *weak-until-or-right-distrib*:
$w \models_n \varphi\ W_n\ (\psi_1\ or_n\ \psi_2) \longleftrightarrow w \models_n (\varphi\ W_n\ \psi_1)\ or_n\ (\varphi\ W_n\ \psi_2)$
by (*metis not*$_n$*.simps*(*10*) *not*$_n$*.simps*(*6*) *not*$_n$*-semantics strong-release-and-right-distrib*)

lemma *next-until-distrib*:
$w \models_n X_n\ (\varphi\ U_n\ \psi) \longleftrightarrow w \models_n (X_n\ \varphi)\ U_n\ (X_n\ \psi)$
by *auto*

lemma *next-release-distrib*:
$w \models_n X_n\ (\varphi\ R_n\ \psi) \longleftrightarrow w \models_n (X_n\ \varphi)\ R_n\ (X_n\ \psi)$
by *auto*

lemma *next-weak-until-distrib*:
$w \models_n X_n\ (\varphi\ W_n\ \psi) \longleftrightarrow w \models_n (X_n\ \varphi)\ W_n\ (X_n\ \psi)$
by *auto*

lemma *next-strong-release-distrib*:
$w \models_n X_n\ (\varphi\ M_n\ \psi) \longleftrightarrow w \models_n (X_n\ \varphi)\ M_n\ (X_n\ \psi)$
by *auto*

1.2.8 Nested operators

lemma *finally-until*[*simp*]:
$w \models_n F_n\ (\varphi\ U_n\ \psi) \longleftrightarrow w \models_n F_n\ \psi$
by *auto force*

lemma *globally-release*[*simp*]:
$w \models_n G_n\ (\varphi\ R_n\ \psi) \longleftrightarrow w \models_n G_n\ \psi$
by *auto force*

lemma *globally-weak-until*[*simp*]:
$w \models_n G_n\ (\varphi\ W_n\ \psi) \longleftrightarrow w \models_n G_n\ (\varphi\ or_n\ \psi)$
by *auto force*

lemma *finally-strong-release*[*simp*]:
$w \models_n F_n\ (\varphi\ M_n\ \psi) \longleftrightarrow w \models_n F_n\ (\varphi\ and_n\ \psi)$
by *auto force*

1.2.9 Weak and strong operators

lemma *ltln-weak-strong*:
$w \models_n \varphi\ W_n\ \psi \longleftrightarrow w \models_n (G_n\ \varphi)\ or_n\ (\varphi\ U_n\ \psi)$
$w \models_n \varphi\ R_n\ \psi \longleftrightarrow w \models_n (G_n\ \psi)\ or_n\ (\varphi\ M_n\ \psi)$
proof *auto*
fix i
assume $\forall i.$ *suffix* $i\ w \models_n \varphi \vee (\exists j{\leq}i.$ *suffix* $j\ w \models_n \psi)$
and $\forall i.$ *suffix* $i\ w \models_n \psi \longrightarrow (\exists j{<}i. \neg$ *suffix* $j\ w \models_n \varphi)$

then show *suffix* $i\ w \models_n \varphi$
by (*induction* i *rule*: *less-induct*) *force*
next
fix $i\ k$
assume $\forall j{\leq}i. \neg$ *suffix* $j\ w \models_n \psi$
and *suffix* $k\ w \models_n \psi$
and $\forall j{<}k.$ *suffix* $j\ w \models_n \varphi$

then show *suffix* $i\ w \models_n \varphi$
by (*cases* $i < k$) *simp-all*
next
fix i
assume $\forall i.$ *suffix* $i\ w \models_n \psi \vee (\exists j{<}i.$ *suffix* $j\ w \models_n \varphi)$
and $\forall i.$ *suffix* $i\ w \models_n \varphi \longrightarrow (\exists j{\leq}i. \neg$ *suffix* $j\ w \models_n \psi)$

then show *suffix* $i\ w \models_n \psi$
by (*induction* i *rule*: *less-induct*) *force*
next
fix $i\ k$
assume $\forall j{<}i. \neg$ *suffix* $j\ w \models_n \varphi$
and *suffix* $k\ w \models_n \varphi$
and $\forall j{\leq}k.$ *suffix* $j\ w \models_n \psi$

then show *suffix* $i\ w \models_n \psi$
by (*cases* $i \leq k$) *simp-all*
qed

lemma *ltln-strong-weak*:
$w \models_n \varphi\ U_n\ \psi \longleftrightarrow w \models_n (F_n\ \psi)\ and_n\ (\varphi\ W_n\ \psi)$
$w \models_n \varphi\ M_n\ \psi \longleftrightarrow w \models_n (F_n\ \varphi)\ and_n\ (\varphi\ R_n\ \psi)$
by (*metis ltln-weak-strong* not_n*.simps*(1,5,8−11) not_n*-semantics*)+

lemma *ltln-strong-to-weak*:
$w \models_n \varphi\ U_n\ \psi \Longrightarrow w \models_n \varphi\ W_n\ \psi$

$w \models_n \varphi\ M_n\ \psi \Longrightarrow w \models_n \varphi\ R_n\ \psi$
using *ltln-weak-strong* **by** *simp-all blast+*

lemma *ltln-weak-to-strong*:
$[\![w \models_n \varphi\ W_n\ \psi;\ \neg\ w \models_n G_n\ \varphi]\!] \Longrightarrow w \models_n \varphi\ U_n\ \psi$
$[\![w \models_n \varphi\ W_n\ \psi;\ w \models_n F_n\ \psi]\!] \Longrightarrow w \models_n \varphi\ U_n\ \psi$
$[\![w \models_n \varphi\ R_n\ \psi;\ \neg\ w \models_n G_n\ \psi]\!] \Longrightarrow w \models_n \varphi\ M_n\ \psi$
$[\![w \models_n \varphi\ R_n\ \psi;\ w \models_n F_n\ \varphi]\!] \Longrightarrow w \models_n \varphi\ M_n\ \psi$
unfolding *ltln-weak-strong*[*of w* φ ψ] **by** *auto*

lemma *ltln-StrongRelease-to-Until*:
$w \models_n \varphi\ M_n\ \psi \longleftrightarrow w \models_n \psi\ U_n\ (\varphi\ and_n\ \psi)$
using *order.order-iff-strict* **by** *auto*

lemma *ltln-Release-to-WeakUntil*:
$w \models_n \varphi\ R_n\ \psi \longleftrightarrow w \models_n \psi\ W_n\ (\varphi\ and_n\ \psi)$
by (*meson ltln-StrongRelease-to-Until ltln-weak-strong semantics-ltln.simps*(*6*))

lemma *ltln-WeakUntil-to-Release*:
$w \models_n \varphi\ W_n\ \psi \longleftrightarrow w \models_n \psi\ R_n\ (\varphi\ or_n\ \psi)$
by (*metis ltln-StrongRelease-to-Until* not_n*.simps*(*6*,*9*,*10*) not_n*-semantics*)

lemma *ltln-Until-to-StrongRelease*:
$w \models_n \varphi\ U_n\ \psi \longleftrightarrow w \models_n \psi\ M_n\ (\varphi\ or_n\ \psi)$
by (*metis ltln-Release-to-WeakUntil* not_n*.simps*(*6*,*8*,*11*) not_n*-semantics*)

1.2.10 GF and FG semantics

lemma *GF-suffix*:
suffix i w $\models_n G_n\ (F_n\ \psi) \longleftrightarrow w \models_n G_n\ (F_n\ \psi)$
by *auto* (*metis ab-semigroup-add-class.add-ac*(*1*) *add.left-commute*)

lemma *FG-suffix*:
suffix i w $\models_n F_n\ (G_n\ \psi) \longleftrightarrow w \models_n F_n\ (G_n\ \psi)$
by (*auto simp*: *algebra-simps*) (*metis add.commute add.left-commute*)

lemma *GF-Inf-many*:
$w \models_n G_n\ (F_n\ \varphi) \longleftrightarrow (\exists_\infty\ i.$ *suffix i w* $\models_n \varphi)$
unfolding *INFM-nat-le*
by *simp* (*blast dest*: *le-Suc-ex intro*: *le-add1*)

lemma *FG-Alm-all*:
$w \models_n F_n\ (G_n\ \varphi) \longleftrightarrow (\forall_\infty\ i.$ *suffix i w* $\models_n \varphi)$

unfolding *MOST-nat-le*
by *simp* (*blast dest*: *le-Suc-ex intro*: *le-add1*)

lemma *MOST-nat-add*:
$(\forall_\infty i{::}nat.\ P\ i) \longleftrightarrow (\forall_\infty i.\ P\ (i + j))$
by (*simp add*: *cofinite-eq-sequentially*)

lemma *INFM-nat-add*:
$(\exists_\infty i{::}nat.\ P\ i) \longleftrightarrow (\exists_\infty i.\ P\ (i + j))$
using *MOST-nat-add not-MOST not-INFM* **by** *blast*

lemma *FG-suffix-G*:
$w \models_n F_n\ (G_n\ \varphi) \Longrightarrow \forall_\infty i.\ suffix\ i\ w \models_n G_n\ \varphi$
proof −
assume $w \models_n F_n\ (G_n\ \varphi)$
then have $w \models_n F_n\ (G_n\ (G_n\ \varphi))$
by (*meson globally-release semantics-ltln.simps*(8))
then show $\forall_\infty i.\ suffix\ i\ w \models_n G_n\ \varphi$
unfolding *FG-Alm-all* **.**
qed

lemma *Alm-all-GF-F*:
$\forall_\infty i.\ suffix\ i\ w \models_n G_n\ (F_n\ \psi) \longleftrightarrow suffix\ i\ w \models_n F_n\ \psi$
unfolding *MOST-nat*
proof *standard*+
fix $i :: nat$
assume $suffix\ i\ w \models_n G_n\ (F_n\ \psi)$
then show $suffix\ i\ w \models_n F_n\ \psi$
unfolding *GF-Inf-many INFM-nat* **by** *fastforce*
next
fix $i :: nat$
assume *suffix*: $suffix\ i\ w \models_n F_n\ \psi$
assume *max*: $i > Max\ \{i.\ suffix\ i\ w \models_n \psi\}$

with *suffix* **obtain** j **where** $j \geq i$ **and** *j-suffix*: $suffix\ j\ w \models_n \psi$
by *simp* (*blast intro*: *le-add1*)

with *max* **have** *j-max*: $j > Max\ \{i.\ suffix\ i\ w \models_n \psi\}$
by *fastforce*

show $suffix\ i\ w \models_n G_n\ (F_n\ \psi)$

```
  proof (cases w ⊨n Gn (Fn ψ))
    case False
    then have ¬ (∃∞i. suffix i w ⊨n ψ)
      unfolding GF-Inf-many by simp
    then have finite {i. suffix i w ⊨n ψ}
      by (simp add: INFM-iff-infinite)
    then have ∀ i>Max {i. suffix i w ⊨n ψ}. ¬ suffix i w ⊨n ψ
      using Max-ge not-le by auto
    then show ?thesis
      using j-suffix j-max by blast
  qed force
qed

lemma Alm-all-FG-G:
  ∀∞i. suffix i w ⊨n Fn (Gn ψ) ⟷ suffix i w ⊨n Gn ψ
  unfolding MOST-nat
proof standard+
  fix i :: nat
  assume suffix i w ⊨n Gn ψ
  then show suffix i w ⊨n Fn (Gn ψ)
    unfolding FG-Alm-all INFM-nat by fastforce
next
  fix i :: nat
  assume suffix: suffix i w ⊨n Fn (Gn ψ)
  assume max: i > Max {i. ¬ suffix i w ⊨n Gn ψ}

  with suffix have ∀∞j. suffix (i + j) w ⊨n Gn ψ
    using FG-suffix-G[of suffix i w] suffix-suffix
    by fastforce
  then have ¬ (∃∞j. ¬ suffix j w ⊨n Gn ψ)
    using MOST-nat-add[of λi. suffix i w ⊨n Gn ψ i]
    by (simp add: algebra-simps)
  then have finite {i. ¬ suffix i w ⊨n Gn ψ}
    by (simp add: INFM-iff-infinite)

  with max show suffix i w ⊨n Gn ψ
    using Max-ge leD by blast
qed
```

1.2.11 Expansion

```
lemma ltln-expand-Until:
  ξ ⊨n φ Un ψ ⟷ (ξ ⊨n ψ orn (φ andn (Xn (φ Un ψ))))
  (is ?lhs = ?rhs)
```

proof
  assume *?lhs*
  then obtain *i* **where** *suffix i ξ* $\models_n$ *ψ*
    and $\forall j{<}i$. *suffix j ξ* $\models_n$ *φ*
    by *auto*
  thus *?rhs*
    by (*cases i*) *auto*
next
  assume *?rhs*
  show *?lhs*
  proof (*cases ξ* $\models_n$ *ψ*)
    case *False*
    then have *ξ* $\models_n$ *φ* **and** *ξ* $\models_n$ X_n (*φ* U_n *ψ*)
      using ‹*?rhs*› **by** *auto*
    thus *?lhs*
      using *less-Suc-eq-0-disj suffix-singleton-suffix* **by** *force*
  qed *force*
qed

lemma *ltln-expand-Release*:
  ξ $\models_n$ *φ* R_n *ψ* $\longleftrightarrow$ (*ξ* $\models_n$ *ψ* and_n (*φ* or_n (X_n (*φ* R_n *ψ*))))
  (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  thus *?rhs*
    using *less-Suc-eq-0-disj* **by** *force*
next
  assume *?rhs*

  {
    fix *i*
    assume ¬ *suffix i ξ* $\models_n$ *ψ*
    then have $\exists j{<}i$. *suffix j ξ* $\models_n$ *φ*
      using ‹*?rhs*› **by** (*cases i*) *force*+
  }

  thus *?lhs*
    by *auto*
qed

lemma *ltln-expand-WeakUntil*:
  ξ $\models_n$ *φ* W_n *ψ* $\longleftrightarrow$ (*ξ* $\models_n$ *ψ* or_n (*φ* and_n (X_n (*φ* W_n *ψ*))))
  (**is** *?lhs* = *?rhs*)
proof

assume *?lhs*
thus *?rhs*
by (*metis ltln-expand-Release ltln-expand-Until ltln-weak-strong*(*1*) *semantics-ltln.simps*(*2*,*5*,*6*,*7*))
next
assume *?rhs*

{
fix *i*
assume ¬ *suffix i ξ* $\models_n$ *φ*
then have ∃*j*≤*i*. *suffix j ξ* $\models_n$ *ψ*
using ‹*?rhs*› **by** (*cases i*) *force*+
}

thus *?lhs*
by *auto*
qed

lemma *ltln-expand-StrongRelease*:
ξ $\models_n$ *φ* M_n *ψ* ⟷ (*ξ* $\models_n$ *ψ* and_n (*φ* or_n (X_n (*φ* M_n *ψ*))))
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *i* **where** *suffix i ξ* $\models_n$ *φ*
and ∀*j*≤*i*. *suffix j ξ* $\models_n$ *ψ*
by *auto*
thus *?rhs*
by (*cases i*) *auto*
next
assume *?rhs*
show *?lhs*
proof (*cases ξ* $\models_n$ *φ*)
case *True*
thus *?lhs*
using ‹*?rhs*› *ltln-expand-WeakUntil* **by** *fastforce*
next
case *False*
thus *?lhs*
by (*metis* ‹*?rhs*› *ltln-expand-WeakUntil not*$_n$*.simps*(*5*,*6*,*7*,*11*) *not*$_n$*-semantics*)
qed
qed

lemma *ltln-Release-alterdef*:
w $\models_n$ *φ* R_n *ψ* ⟷ *w* $\models_n$ (G_n *ψ*) or_n (*ψ* U_n (*φ* and_n *ψ*))

proof (*cases* $\exists\, i.\ \neg$*suffix* $i\ w \models_n \psi$)
case *True*
define i **where** $i \equiv$ *Least* $(\lambda i.\ \neg$*suffix* $i\ w \models_n \psi)$
have $\bigwedge j.\ j < i \Longrightarrow$ *suffix* $j\ w \models_n \psi$ **and** $\neg$ *suffix* $i\ w \models_n \psi$
using *True LeastI not-less-Least* **unfolding** *i-def* **by** *fast+*
hence $*$: $\forall\, i.$ *suffix* $i\ w \models_n \psi \vee (\exists\, j{<}i.$ *suffix* $j\ w \models_n \varphi) \Longrightarrow (\exists\, i.$ (*suffix* $i\ w \models_n \psi \wedge$ *suffix* $i\ w \models_n \varphi) \wedge (\forall\, j{<}i.$ *suffix* $j\ w \models_n \psi))$
by *fastforce*
hence $\exists\, i.$ (*suffix* $i\ w \models_n \psi \wedge$ *suffix* $i\ w \models_n \varphi) \wedge (\forall\, j{<}i.$ *suffix* $j\ w \models_n \psi)$ $\Longrightarrow (\forall\, i.$ *suffix* $i\ w \models_n \psi \vee (\exists\, j{<}i.$ *suffix* $j\ w \models_n \varphi))$
using *linorder-cases* **by** *blast*
thus *?thesis*
using *True* $*$ **by** *auto*
qed *auto*

1.3 LTL in restricted Negation Normal Form

Some algorithms do not handle the operators W and M, hence we also provide a datatype without these two operators.

1.3.1 Syntax

datatype (*atoms-ltlr*: *'a*) *ltlr* =

True-ltlr	(‹$true_r$›)
| *False-ltlr*	(‹$false_r$›)
| *Prop-ltlr 'a*	(‹$prop_r$ *'(-')*›)
| *Nprop-ltlr 'a*	(‹$nprop_r$ *'(-')*›)
| *And-ltlr 'a ltlr 'a ltlr*	(‹- and_r -› [*82*,*82*] *81*)
| *Or-ltlr 'a ltlr 'a ltlr*	(‹- or_r -› [*84*,*84*] *83*)
| *Next-ltlr 'a ltlr*	(‹X_r -› [*88*] *87*)
| *Until-ltlr 'a ltlr 'a ltlr*	(‹- U_r -› [*84*,*84*] *83*)
| *Release-ltlr 'a ltlr 'a ltlr*	(‹- R_r -› [*84*,*84*] *83*)

1.3.2 Semantics

primrec *semantics-ltlr* :: [*'a set word*, *'a ltlr*] $\Rightarrow$ *bool* (‹- $\models_r$ -› [*80*,*80*] *80*)
where
$\xi \models_r true_r = True$
$| \ \xi \models_r false_r = False$
$| \ \xi \models_r prop_r(q) = (q \in \xi\ 0)$
$| \ \xi \models_r nprop_r(q) = (q \notin \xi\ 0)$
$| \ \xi \models_r \varphi\ and_r\ \psi = (\xi \models_r \varphi \wedge \xi \models_r \psi)$
$| \ \xi \models_r \varphi\ or_r\ \psi = (\xi \models_r \varphi \vee \xi \models_r \psi)$
$| \ \xi \models_r X_r\ \varphi = (suffix\ 1\ \xi \models_r \varphi)$

$| \ \xi \models_r \varphi \ U_r \ \psi = (\exists\, i.\ \mathit{suffix}\ i\ \xi \models_r \psi \wedge (\forall\, j{<}i.\ \mathit{suffix}\ j\ \xi \models_r \varphi))$
$| \ \xi \models_r \varphi \ R_r \ \psi = (\forall\, i.\ \mathit{suffix}\ i\ \xi \models_r \psi \vee (\exists\, j{<}i.\ \mathit{suffix}\ j\ \xi \models_r \varphi))$

1.3.3 Conversion

fun *ltln-to-ltlr* :: *'a ltln* ⇒ *'a ltlr*
where
ltln-to-ltlr $true_n = true_r$
| *ltln-to-ltlr* $false_n = false_r$
| *ltln-to-ltlr* $prop_n(a) = prop_r(a)$
| *ltln-to-ltlr* $nprop_n(a) = nprop_r(a)$
| *ltln-to-ltlr* $(\varphi\ and_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \varphi)\ and_r\ (ltln\text{-}to\text{-}ltlr\ \psi)$
| *ltln-to-ltlr* $(\varphi\ or_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \varphi)\ or_r\ (ltln\text{-}to\text{-}ltlr\ \psi)$
| *ltln-to-ltlr* $(X_n\ \varphi) = X_r\ (ltln\text{-}to\text{-}ltlr\ \varphi)$
| *ltln-to-ltlr* $(\varphi\ U_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \varphi)\ U_r\ (ltln\text{-}to\text{-}ltlr\ \psi)$
| *ltln-to-ltlr* $(\varphi\ R_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \varphi)\ R_r\ (ltln\text{-}to\text{-}ltlr\ \psi)$
| *ltln-to-ltlr* $(\varphi\ W_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \psi)\ R_r\ ((ltln\text{-}to\text{-}ltlr\ \varphi)\ or_r\ (ltln\text{-}to\text{-}ltlr\ \psi))$
| *ltln-to-ltlr* $(\varphi\ M_n\ \psi) = (ltln\text{-}to\text{-}ltlr\ \psi)\ U_r\ ((ltln\text{-}to\text{-}ltlr\ \varphi)\ and_r\ (ltln\text{-}to\text{-}ltlr\ \psi))$

fun *ltlr-to-ltln* :: *'a ltlr* ⇒ *'a ltln*
where
ltlr-to-ltln $true_r = true_n$
| *ltlr-to-ltln* $false_r = false_n$
| *ltlr-to-ltln* $prop_r(a) = prop_n(a)$
| *ltlr-to-ltln* $nprop_r(a) = nprop_n(a)$
| *ltlr-to-ltln* $(\varphi\ and_r\ \psi) = (ltlr\text{-}to\text{-}ltln\ \varphi)\ and_n\ (ltlr\text{-}to\text{-}ltln\ \psi)$
| *ltlr-to-ltln* $(\varphi\ or_r\ \psi) = (ltlr\text{-}to\text{-}ltln\ \varphi)\ or_n\ (ltlr\text{-}to\text{-}ltln\ \psi)$
| *ltlr-to-ltln* $(X_r\ \varphi) = X_n\ (ltlr\text{-}to\text{-}ltln\ \varphi)$
| *ltlr-to-ltln* $(\varphi\ U_r\ \psi) = (ltlr\text{-}to\text{-}ltln\ \varphi)\ U_n\ (ltlr\text{-}to\text{-}ltln\ \psi)$
| *ltlr-to-ltln* $(\varphi\ R_r\ \psi) = (ltlr\text{-}to\text{-}ltln\ \varphi)\ R_n\ (ltlr\text{-}to\text{-}ltln\ \psi)$

lemma *ltln-to-ltlr-semantics*:
$w \models_r ltln\text{-}to\text{-}ltlr\ \varphi \longleftrightarrow w \models_n \varphi$
by (*induction* φ *arbitrary*: *w*) (*unfold ltln-WeakUntil-to-Release ltln-StrongRelease-to-Until*, *simp-all*)

lemma *ltlr-to-ltln-semantics*:
$w \models_n ltlr\text{-}to\text{-}ltln\ \varphi \longleftrightarrow w \models_r \varphi$
by (*induction* φ *arbitrary*: *w*) *simp-all*

1.3.4 Negation

fun not_r
where
  $not_r\ true_r = false_r$
| $not_r\ false_r = true_r$
| $not_r\ prop_r(a) = nprop_r(a)$
| $not_r\ nprop_r(a) = prop_r(a)$
| $not_r\ (\varphi\ and_r\ \psi) = (not_r\ \varphi)\ or_r\ (not_r\ \psi)$
| $not_r\ (\varphi\ or_r\ \psi) = (not_r\ \varphi)\ and_r\ (not_r\ \psi)$
| $not_r\ (X_r\ \varphi) = X_r\ (not_r\ \varphi)$
| $not_r\ (\varphi\ U_r\ \psi) = (not_r\ \varphi)\ R_r\ (not_r\ \psi)$
| $not_r\ (\varphi\ R_r\ \psi) = (not_r\ \varphi)\ U_r\ (not_r\ \psi)$

lemma not_r*-semantics* [*simp*]:
  $w \models_r not_r\ \varphi \longleftrightarrow \neg\ w \models_r \varphi$
  by (*induction* φ *arbitrary*: w) *auto*

1.3.5 Subformulas

fun *subfrmlsr* :: *'a ltlr* ⇒ *'a ltlr set*
where
  subfrmlsr $(\varphi\ and_r\ \psi) = \{\varphi\ and_r\ \psi\} \cup$ *subfrmlsr* $\varphi \cup$ *subfrmlsr* ψ
| *subfrmlsr* $(\varphi\ or_r\ \psi) = \{\varphi\ or_r\ \psi\} \cup$ *subfrmlsr* $\varphi \cup$ *subfrmlsr* ψ
| *subfrmlsr* $(\varphi\ U_r\ \psi) = \{\varphi\ U_r\ \psi\} \cup$ *subfrmlsr* $\varphi \cup$ *subfrmlsr* ψ
| *subfrmlsr* $(\varphi\ R_r\ \psi) = \{\varphi\ R_r\ \psi\} \cup$ *subfrmlsr* $\varphi \cup$ *subfrmlsr* ψ
| *subfrmlsr* $(X_r\ \varphi) = \{X_r\ \varphi\} \cup$ *subfrmlsr* φ
| *subfrmlsr* $x = \{x\}$

lemma *subfrmlsr-id*[*simp*]:
  $\varphi \in$ *subfrmlsr* φ
  by (*induction* φ) *auto*

lemma *subfrmlsr-finite*:
  finite (*subfrmlsr* φ)
  by (*induction* φ) *auto*

lemma *subfrmlsr-subset*:
  $\psi \in$ *subfrmlsr* $\varphi \Longrightarrow$ *subfrmlsr* $\psi \subseteq$ *subfrmlsr* φ
  by (*induction* φ) *auto*

lemma *subfrmlsr-size*:
  $\psi \in$ *subfrmlsr* $\varphi \Longrightarrow$ *size* $\psi <$ *size* $\varphi \vee \psi = \varphi$
  by (*induction* φ) *auto*

1.3.6 Expansion lemmas

lemma *ltlr-expand-Until*:
$\xi \models_r \varphi\ U_r\ \psi \longleftrightarrow (\xi \models_r \psi\ or_r\ (\varphi\ and_r\ (X_r\ (\varphi\ U_r\ \psi))))$
by (*metis ltln-expand-Until ltlr-to-ltln.simps*(5−8) *ltlr-to-ltln-semantics*)

lemma *ltlr-expand-Release*:
$\xi \models_r \varphi\ R_r\ \psi \longleftrightarrow (\xi \models_r \psi\ and_r\ (\varphi\ or_r\ (X_r\ (\varphi\ R_r\ \psi))))$
by (*metis ltln-expand-Release ltlr-to-ltln.simps*(5−7,9) *ltlr-to-ltln-semantics*)

1.4 Propositional LTL

We define the syntax and semantics of propositional linear-time temporal logic PLTL. PLTL formulas are built from atomic formulas, propositional connectives, and the temporal operators "next" and "until". The following data type definition is parameterized by the type of states over which formulas are evaluated.

1.4.1 Syntax

datatype *'a pltl* =
  False-ltlp ($\langle false_p \rangle$)
| *Atom-ltlp 'a* ⇒ *bool* ($\langle atom_p\,'(-')\rangle$)
| *Implies-ltlp 'a pltl 'a pltl* ($\langle$- $implies_p$ -› [81,81] 80)
| *Next-ltlp 'a pltl* ($\langle X_p$ -› [88] 87)
| *Until-ltlp 'a pltl 'a pltl* ($\langle$- U_p -› [84,84] 83)

— Further connectives of PLTL can be defined in terms of the existing syntax.

definition *Not-ltlp* ($\langle not_p$ -› [85] 85)
where
$not_p\ \varphi \equiv \varphi\ implies_p\ false_p$

definition *True-ltlp* ($\langle true_p \rangle$)
where
$true_p \equiv not_p\ false_p$

definition *Or-ltlp* ($\langle$- or_p -› [81,81] 80)
where
$\varphi\ or_p\ \psi \equiv (not_p\ \varphi)\ implies_p\ \psi$

definition *And-ltlp* ($\langle$- and_p -› [82,82] 81)
where

$\varphi\ and_p\ \psi \equiv not_p\ ((not_p\ \varphi)\ or_p\ (not_p\ \psi))$

definition *Eventually-ltlp* (‹F_p -› [*88*] *87*)
where
$F_p\ \varphi \equiv true_p\ U_p\ \varphi$

definition *Always-ltlp* (‹G_p -› [*88*] *87*)
where
$G_p\ \varphi \equiv not_p\ (F_p\ (not_p\ \varphi))$

definition *Release-ltlp* (‹- R_p -› [*84*,*84*] *83*)
where
$\varphi\ R_p\ \psi \equiv not_p\ ((not_p\ \varphi)\ U_p\ (not_p\ \psi))$

definition *WeakUntil-ltlp* (‹- W_p -› [*84*,*84*] *83*)
where
$\varphi\ W_p\ \psi \equiv \psi\ R_p\ (\varphi\ or_p\ \psi)$

definition *StrongRelease-ltlp* (‹- M_p -› [*84*,*84*] *83*)
where
$\varphi\ M_p\ \psi \equiv \psi\ U_p\ (\varphi\ and_p\ \psi)$

1.4.2 Semantics

fun *semantics-pltl* :: ['*a word*, '*a pltl*] ⇒ *bool* (‹- $\models_p$ -› [*80*,*80*] *80*)
where
$w \models_p false_p = False$
| $w \models_p atom_p(p) = (p\ (w\ 0))$
| $w \models_p \varphi\ implies_p\ \psi = (w \models_p \varphi \longrightarrow w \models_p \psi)$
| $w \models_p X_p\ \varphi = (suffix\ 1\ w \models_p \varphi)$
| $w \models_p \varphi\ U_p\ \psi = (\exists\, i.\ suffix\ i\ w \models_p \psi \wedge (\forall\, j{<}i.\ suffix\ j\ w \models_p \varphi))$

lemma *semantics-pltl-sugar* [*simp*]:
$w \models_p not_p\ \varphi = (\neg w \models_p \varphi)$
$w \models_p true_p = True$
$w \models_p \varphi\ or_p\ \psi = (w \models_p \varphi \vee w \models_p \psi)$
$w \models_p \varphi\ and_p\ \psi = (w \models_p \varphi \wedge w \models_p \psi)$
$w \models_p F_p\ \varphi = (\exists\, i.\ suffix\ i\ w \models_p \varphi)$
$w \models_p G_p\ \varphi = (\forall\, i.\ suffix\ i\ w \models_p \varphi)$
$w \models_p \varphi\ R_p\ \psi = (\forall\, i.\ suffix\ i\ w \models_p \psi \vee (\exists\, j{<}i.\ suffix\ j\ w \models_p \varphi))$
$w \models_p \varphi\ W_p\ \psi = (\forall\, i.\ suffix\ i\ w \models_p \varphi \vee (\exists\, j{\leq}i.\ suffix\ j\ w \models_p \psi))$
$w \models_p \varphi\ M_p\ \psi = (\exists\, i.\ suffix\ i\ w \models_p \varphi \wedge (\forall\, j{\leq}i.\ suffix\ j\ w \models_p \psi))$
by (*auto simp*: *Not-ltlp-def True-ltlp-def Or-ltlp-def And-ltlp-def Eventually-ltlp-def Always-ltlp-def Release-ltlp-def WeakUntil-ltlp-def StrongRe-*

lease-ltlp-def) (*insert le-neq-implies-less*, *blast*)+

definition *language-ltlp* $\varphi \equiv \{\xi.\ \xi \models_p \varphi\}$

1.4.3 Conversion

fun *ltlc-to-pltl* :: $'a$ *ltlc* $\Rightarrow$ $'a$ *set pltl*
where
  ltlc-to-pltl $true_c$ = $true_p$
| *ltlc-to-pltl* $false_c$ = $false_p$
| *ltlc-to-pltl* $(prop_c(q))$ = $atom_p((\in)\ q)$
| *ltlc-to-pltl* $(not_c\ \varphi)$ = not_p (*ltlc-to-pltl* φ)
| *ltlc-to-pltl* $(\varphi\ and_c\ \psi)$ = (*ltlc-to-pltl* φ) and_p (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(\varphi\ or_c\ \psi)$ = (*ltlc-to-pltl* φ) or_p (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(\varphi\ implies_c\ \psi)$ = (*ltlc-to-pltl* φ) $implies_p$ (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(X_c\ \varphi)$ = X_p (*ltlc-to-pltl* φ)
| *ltlc-to-pltl* $(F_c\ \varphi)$ = F_p (*ltlc-to-pltl* φ)
| *ltlc-to-pltl* $(G_c\ \varphi)$ = G_p (*ltlc-to-pltl* φ)
| *ltlc-to-pltl* $(\varphi\ U_c\ \psi)$ = (*ltlc-to-pltl* φ) U_p (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(\varphi\ R_c\ \psi)$ = (*ltlc-to-pltl* φ) R_p (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(\varphi\ W_c\ \psi)$ = (*ltlc-to-pltl* φ) W_p (*ltlc-to-pltl* ψ)
| *ltlc-to-pltl* $(\varphi\ M_c\ \psi)$ = (*ltlc-to-pltl* φ) M_p (*ltlc-to-pltl* ψ)

lemma *ltlc-to-pltl-semantics* [*simp*]:
  $w \models_p$ (*ltlc-to-pltl* φ) $\longleftrightarrow$ $w \models_c \varphi$
  by (*induction* φ *arbitrary*: w) *simp-all*

1.4.4 Atoms

fun *atoms-pltl* :: $'a$ *pltl* $\Rightarrow$ ($'a$ $\Rightarrow$ *bool*) *set*
where
  atoms-pltl $false_p$ = $\{\}$
| *atoms-pltl* $atom_p(p)$ = $\{p\}$
| *atoms-pltl* $(\varphi\ implies_p\ \psi)$ = *atoms-pltl* $\varphi \cup$ *atoms-pltl* ψ
| *atoms-pltl* $(X_p\ \varphi)$ = *atoms-pltl* φ
| *atoms-pltl* $(\varphi\ U_p\ \psi)$ = *atoms-pltl* $\varphi \cup$ *atoms-pltl* ψ

lemma *atoms-finite* [*iff*]:
  finite (*atoms-pltl* φ)
  by (*induct* φ) *auto*

lemma *atoms-pltl-sugar* [*simp*]:
  atoms-pltl $(not_p\ \varphi)$ = *atoms-pltl* φ
  atoms-pltl $true_p$ = $\{\}$

atoms-pltl (φ or_p ψ) = *atoms-pltl* φ ∪ *atoms-pltl* ψ
atoms-pltl (φ and_p ψ) = *atoms-pltl* φ ∪ *atoms-pltl* ψ
atoms-pltl (F_p φ) = *atoms-pltl* φ
atoms-pltl (G_p φ) = *atoms-pltl* φ
by (*auto simp*: *Not-ltlp-def True-ltlp-def Or-ltlp-def And-ltlp-def Eventually-ltlp-def Always-ltlp-def*)

end

2 Rewrite Rules for LTL Simplification

theory *Rewriting*
imports
LTL HOL−Library.Extended-Nat
begin

This theory provides rewrite rules for the simplification of LTL formulas. It supports:

- Constants Removal
- *Next-ltln*-Normalisation
- Modal Simplification (based on pure eventual, pure universal, or suspendable formulas)
- Syntactic Implication Checking

It reuses parts of LTL_Rewrite.thy (CAVA, LTL_TO_GBA). Furthermore, some rules were taken from [2] and [1]. All functions are defined for *ltln*.

2.1 Constant Eliminating Constructors

definition *mk-and*
where
mk-and x y ≡ *case x of* $false_n$ ⇒ $false_n$ | $true_n$ ⇒ *y* | - ⇒ (*case y of* $false_n$ ⇒ $false_n$ | $true_n$ ⇒ *x* | - ⇒ *x* and_n *y*)

definition *mk-or*
where
mk-or x y ≡ *case x of* $false_n$ ⇒ *y* | $true_n$ ⇒ $true_n$ | - ⇒ (*case y of* $true_n$ ⇒ $true_n$ | $false_n$ ⇒ *x* | - ⇒ *x* or_n *y*)

fun *remove-strong-ops*
where

remove-strong-ops $(x\ U_n\ y)$ = *remove-strong-ops* y
| *remove-strong-ops* $(x\ M_n\ y)$ = $x\ and_n\ y$
| *remove-strong-ops* $(x\ or_n\ y)$ = *remove-strong-ops* $x\ or_n$ *remove-strong-ops* y
| *remove-strong-ops* $x = x$

fun *remove-weak-ops*
where
remove-weak-ops $(x\ R_n\ y)$ = *remove-weak-ops* y
| *remove-weak-ops* $(x\ W_n\ y) = x\ or_n\ y$
| *remove-weak-ops* $(x\ and_n\ y)$ = *remove-weak-ops* $x\ and_n$ *remove-weak-ops* y
| *remove-weak-ops* $x = x$

definition *mk-finally*
where
mk-finally $x \equiv$ *case* x *of* $true_n \Rightarrow true_n \mid false_n \Rightarrow false_n \mid \text{-} \Rightarrow F_n$ (*remove-strong-ops* x)

definition *mk-globally*
where
mk-globally $x \equiv$ *case* x *of* $true_n \Rightarrow true_n \mid false_n \Rightarrow false_n \mid \text{-} \Rightarrow G_n$ (*remove-weak-ops* x)

definition *mk-until*
where
mk-until $x\ y \equiv$ *case* x *of* $false_n \Rightarrow y$
| $true_n \Rightarrow$ *mk-finally* y
| - $\Rightarrow$ (*case* y *of* $true_n \Rightarrow true_n \mid false_n \Rightarrow false_n \mid \text{-} \Rightarrow x\ U_n\ y$)

definition *mk-release*
where
mk-release $x\ y \equiv$ *case* x *of* $true_n \Rightarrow y$
| $false_n \Rightarrow$ *mk-globally* y
| - $\Rightarrow$ (*case* y *of* $true_n \Rightarrow true_n \mid false_n \Rightarrow false_n \mid \text{-} \Rightarrow x\ R_n\ y$)

definition *mk-weak-until*
where
mk-weak-until $x\ y \equiv$ *case* y *of* $true_n \Rightarrow true_n$
| $false_n \Rightarrow$ *mk-globally* x
| - $\Rightarrow$ (*case* x *of* $true_n \Rightarrow true_n \mid false_n \Rightarrow y \mid \text{-} \Rightarrow x\ W_n\ y$)

definition *mk-strong-release*
where

mk-strong-release x y ≡ *case y of false*$_n$ ⇒ *false*$_n$
| *true*$_n$ ⇒ *mk-finally x*
| - ⇒ (*case x of true*$_n$ ⇒ *y* | *false*$_n$ ⇒ *false*$_n$ | - ⇒ *x M*$_n$ *y*)

definition *mk-next*
where
mk-next x ≡ *case x of true*$_n$ ⇒ *true*$_n$ | *false*$_n$ ⇒ *false*$_n$ | - ⇒ *X*$_n$ *x*

definition *mk-next-pow* (‹*X*$_n$ ″›)
where
mk-next-pow n x ≡ *case x of true*$_n$ ⇒ *true*$_n$ | *false*$_n$ ⇒ *false*$_n$ | - ⇒ (*Next-ltln* ^^ *n*) *x*

lemma *mk-and-semantics* [*simp*]:
w ⊨$_n$ *mk-and x y* ⟷ *w* ⊨$_n$ *x and*$_n$ *y*
unfolding *mk-and-def* **by** (*cases x*; *cases y*; *simp*)

lemma *mk-or-semantics* [*simp*]:
w ⊨$_n$ *mk-or x y* ⟷ *w* ⊨$_n$ *x or*$_n$ *y*
unfolding *mk-or-def* **by** (*cases x*; *cases y*; *simp*)

lemma *remove-strong-ops-sound* [*simp*]:
w ⊨$_n$ *F*$_n$ (*remove-strong-ops y*) ⟷ *w* ⊨$_n$ *F*$_n$ *y*
by (*induction y arbitrary*: *w*) (*auto*; *force*)+

lemma *remove-weak-ops-sound* [*simp*]:
w ⊨$_n$ *G*$_n$ (*remove-weak-ops y*) ⟷ *w* ⊨$_n$ *G*$_n$ *y*
by (*induction y arbitrary*: *w*) (*auto*; *force*)+

lemma *mk-finally-semantics* [*simp*]:
w ⊨$_n$ *mk-finally x* ⟷ *w* ⊨$_n$ *F*$_n$ *x*
by (*simp add*: *mk-finally-def del*: *semantics-ltln.simps*(*8*,*9*) *remove-strong-ops.simps split*: *ltln.splits*)

lemma *mk-globally-semantics* [*simp*]:
w ⊨$_n$ *mk-globally x* ⟷ *w* ⊨$_n$ *G*$_n$ *x*
by (*simp add*: *mk-globally-def del*: *semantics-ltln.simps*(*8*,*9*) *remove-weak-ops.simps split*: *ltln.splits*)

lemma *mk-until-semantics* [*simp*]:
w ⊨$_n$ *mk-until x y* ⟷ *w* ⊨$_n$ *x U*$_n$ *y*
proof (*cases x*)

```
  case (True-ltln)
    show ?thesis
      unfolding True-ltln mk-until-def
      by (cases y) auto
next
  case (False-ltln)
    thus ?thesis
      by (force simp: mk-until-def)
qed (cases y; force simp: mk-until-def)+

lemma mk-release-semantics [simp]:
  w ⊨n mk-release x y ⟷ w ⊨n x Rn y
proof (cases x)
  case (False-ltln)
    thus ?thesis
      unfolding False-ltln mk-release-def
      by (cases y) auto
next
  case (True-ltln)
    thus ?thesis
      by (force simp: mk-release-def)
qed (cases y; force simp: mk-release-def)+

lemma mk-weak-until-semantics [simp]:
  w ⊨n mk-weak-until x y ⟷ w ⊨n x Wn y
proof (cases y)
  case (False-ltln)
    thus ?thesis
      unfolding False-ltln mk-weak-until-def
      by (cases x) auto
next
  case (True-ltln)
    thus ?thesis
      by (force simp: mk-weak-until-def)
qed (cases x; force simp: mk-weak-until-def)+

lemma mk-strong-release-semantics [simp]:
  w ⊨n mk-strong-release x y ⟷ w ⊨n x Mn y
proof (cases y)
  case (True-ltln)
    show ?thesis
      unfolding True-ltln mk-strong-release-def
      by (cases x) auto
next
```

case (*False-ltln*)
thus *?thesis*
by (*force simp*: *mk-strong-release-def*)
qed (*cases x*; *force simp*: *mk-strong-release-def*)+

lemma *mk-next-semantics* [*simp*]:
$w \models_n$ *mk-next x* $\longleftrightarrow$ $w \models_n X_n\ x$
unfolding *mk-next-def* **by** (*cases x*; *auto*)

lemma *mk-next-pow-semantics* [*simp*]:
$w \models_n$ *mk-next-pow i x* $\longleftrightarrow$ *suffix i w* $\models_n x$
by (*induction i arbitrary*: *w*; *cases x*)
(*auto simp*: *mk-next-pow-def*)

lemma *mk-next-pow-simp* [*simp*, *code-unfold*]:
mk-next-pow 0 x = *x*
mk-next-pow 1 x = *mk-next x*
by (*cases x*; *simp add*: *mk-next-pow-def mk-next-def*)+

2.2 Constant Propagation

fun *is-constant* :: *'a ltln* ⇒ *bool*
where
is-constant $true_n$ = *True*
| *is-constant* $false_n$ = *True*
| *is-constant* - = *False*

lemma *is-constant-constructorsI*:
is-constant x ⟹ *is-constant y* ⟹ *is-constant* (*mk-and x y*)
¬*is-constant x* ⟹ ¬*is-constant y* ⟹ ¬*is-constant* (*mk-and x y*)
is-constant x ⟹ *is-constant y* ⟹ *is-constant* (*mk-or x y*)
¬*is-constant x* ⟹ ¬*is-constant y* ⟹ ¬*is-constant* (*mk-or x y*)
is-constant x ⟹ *is-constant* (*mk-finally x*)
¬*is-constant x* ⟹ ¬*is-constant* (*mk-finally x*)
is-constant x ⟹ *is-constant* (*mk-globally x*)
¬*is-constant x* ⟹ ¬*is-constant* (*mk-globally x*)
is-constant x ⟹ *is-constant* (*mk-until y x*)
¬*is-constant x* ⟹ ¬*is-constant* (*mk-until y x*)
is-constant x ⟹ *is-constant* (*mk-release y x*)
¬*is-constant x* ⟹ ¬*is-constant* (*mk-release y x*)
is-constant x ⟹ *is-constant y* ⟹ *is-constant* (*mk-weak-until x y*)
¬*is-constant x* ⟹ ¬*is-constant y* ⟹ ¬*is-constant* (*mk-weak-until x y*)
is-constant x ⟹ *is-constant y* ⟹ *is-constant* (*mk-strong-release x y*)
¬*is-constant x* ⟹ ¬*is-constant y* ⟹ ¬*is-constant* (*mk-strong-release x*

y)
is-constant $x \Longrightarrow$ *is-constant* (*mk-next* x)
$\neg$*is-constant* $x \Longrightarrow \neg$*is-constant* (*mk-next* x)
is-constant $x \Longrightarrow$ *is-constant* (*mk-next-pow* n x)
by (*cases* x; *cases* y; *simp add*: *mk-and-def mk-or-def mk-finally-def mk-globally-def mk-until-def mk-release-def mk-weak-until-def mk-strong-release-def mk-next-def mk-next-pow-def*)+

lemma *is-constant-constructors-simps*:
mk-next-pow n x = $false_n \longleftrightarrow x = false_n$
mk-next-pow n x = $true_n \longleftrightarrow x = true_n$
is-constant (*mk-next-pow* n x) $\longleftrightarrow$ *is-constant* x
by (*induction* n) (*cases* x; *simp add*: *mk-next-pow-def*)+

lemma *is-constant-constructors-simps2*:
is-constant (*mk-and* x y) $\longleftrightarrow$ ($x = true_n \wedge y = true_n \vee x = false_n \vee y = false_n$)
is-constant (*mk-or* x y) $\longleftrightarrow$ ($x = false_n \wedge y = false_n \vee x = true_n \vee y = true_n$)
is-constant (*mk-finally* x) $\longleftrightarrow$ *is-constant* x
is-constant (*mk-globally* x) $\longleftrightarrow$ *is-constant* x
is-constant (*mk-until* y x) $\longleftrightarrow$ *is-constant* x
is-constant (*mk-release* y x) $\longleftrightarrow$ *is-constant* x
is-constant (*mk-next* x) $\longleftrightarrow$ *is-constant* x
by ((*cases* x; *cases* y; *simp add*: *mk-and-def*),
(*cases* x; *cases* y; *simp add*: *mk-or-def*),
(*meson is-constant-constructorsI*)+)

lemma *is-constant-constructors-simps3*:
is-constant (*mk-weak-until* x y) $\longleftrightarrow$ ($x = false_n \wedge y = false_n \vee x = true_n \vee y = true_n$)
is-constant (*mk-strong-release* x y) $\longleftrightarrow$ ($x = true_n \wedge y = true_n \vee x = false_n \vee y = false_n$)
by (*cases* x; *cases* y; *simp add*: *mk-weak-until-def mk-strong-release-def is-constant-constructors-simps2*)+

lemma *is-constant-semantics*:
is-constant $\varphi \Longrightarrow ((\forall\, w.\ w \models_n \varphi) \vee \neg(\exists\, w.\ w \models_n \varphi))$
by (*cases* φ) *auto*

lemma *until-constant-simp*:
is-constant $\psi \Longrightarrow w \models_n \varphi\ U_n\ \psi \longleftrightarrow w \models_n \psi$
by (*cases* ψ) *auto*

lemma *release-constant-simp*:
is-constant $\psi \Longrightarrow w \models_n \varphi\ R_n\ \psi \longleftrightarrow w \models_n \psi$
by (*cases* ψ) *auto*

lemma *mk-next-pow-dist*:
mk-next-pow $(i + j)\ \varphi$ = *mk-next-pow* i (*mk-next-pow* $j\ \varphi$)
by (*cases* j; *simp*) (*cases* φ; *simp add*: *mk-next-pow-def funpow-add*; *simp add*: *funpow-swap1*)

lemma *mk-next-pow-until*:
suffix (*min* $i\ j$) $w \models_n$ (*mk-next-pow* $(i - j)\ \varphi$) U_n (*mk-next-pow* $(j - i)\ \psi$) $\longleftrightarrow w \models_n$ (*mk-next-pow* $i\ \varphi$) U_n (*mk-next-pow* $j\ \psi$)
by (*simp add*: *mk-next-pow-dist min-def add.commute*)

lemma *mk-next-pow-release*:
suffix (*min* $i\ j$) $w \models_n$ (*mk-next-pow* $(i - j)\ \varphi$) R_n (*mk-next-pow* $(j - i)\ \psi$) $\longleftrightarrow w \models_n$ (*mk-next-pow* $i\ \varphi$) R_n (*mk-next-pow* $j\ \psi$)
by (*simp add*: *mk-next-pow-dist min-def add.commute*)

2.3 X-Normalisation

The following rewrite functions pulls the X-operator up in the syntax tree. This preprocessing step enables the removal of X-operators in front of suspendable formulas. Furthermore constants are removed as far as possible.

fun *the-enat-0* :: *enat* ⇒ *nat*
where
the-enat-0 i = *i*
| *the-enat-0* ∞ = *0*

lemma *the-enat-0-simp* [*simp*]:
the-enat-0 0 = *0*
the-enat-0 1 = *1*
by (*simp add*: *zero-enat-def one-enat-def*)+

fun *combine* :: (′*a ltln* ⇒ ′*a ltln* ⇒ ′*a ltln*) ⇒ (′*a ltln* ∗ *enat*) ⇒ (′*a ltln* ∗ *enat*) ⇒ (′*a ltln* ∗ *enat*)
where
combine binop (*φ*, *i*) (*ψ*, *j*) = (
let
χ = *binop* (*mk-next-pow* (*the-enat-0* $(i - j)$) *φ*) (*mk-next-pow* (*the-enat-0* $(j - i)$) *ψ*)
in
(*χ*, *if is-constant χ then* ∞ *else min i j*))

lemma *fst-combine*:
fst (*combine binop* (*φ*, *i*) (*ψ*, *j*)) = *binop* (*mk-next-pow* (*the-enat-0* (*i* − *j*)) *φ*) (*mk-next-pow* (*the-enat-0* (*j* − *i*)) *ψ*)
unfolding *combine.simps* **by** (*meson fstI*)

abbreviation *to-ltln* :: ('*a ltln* ∗ *enat*) ⇒ '*a ltln*
where
to-ltln x ≡ *mk-next-pow* (*the-enat-0* (*snd x*)) (*fst x*)

fun *rewrite-X-enat* :: '*a ltln* ⇒ ('*a ltln* ∗ *enat*)
where
rewrite-X-enat true$_n$ = (*true*$_n$, ∞)
| *rewrite-X-enat false*$_n$ = (*false*$_n$, ∞)
| *rewrite-X-enat prop*$_n$(*a*) = (*prop*$_n$(*a*), *0*)
| *rewrite-X-enat nprop*$_n$(*a*) = (*nprop*$_n$(*a*), *0*)
| *rewrite-X-enat* (*φ and*$_n$ *ψ*) = *combine mk-and* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*φ or*$_n$ *ψ*) = *combine mk-or* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*φ U*$_n$ *ψ*) = *combine mk-until* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*φ R*$_n$ *ψ*) = *combine mk-release* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*φ W*$_n$ *ψ*) = *combine mk-weak-until* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*φ M*$_n$ *ψ*) = *combine mk-strong-release* (*rewrite-X-enat φ*) (*rewrite-X-enat ψ*)
| *rewrite-X-enat* (*X*$_n$ *φ*) = (λ(*φ*, *n*). (*φ*, *eSuc n*)) (*rewrite-X-enat φ*)

definition
rewrite-X φ = *to-ltln* (*rewrite-X-enat φ*)

lemma *combine-infinity-invariant*:
assumes *i* = ∞ ⟷ *is-constant x*
assumes *j* = ∞ ⟷ *is-constant y*
shows *combine mk-and* (*x*, *i*) (*y*, *j*) = (*z*, *k*) ⟹ (*k* = ∞ ⟷ *is-constant z*)
and *combine mk-or* (*x*, *i*) (*y*, *j*) = (*z*, *k*) ⟹ (*k* = ∞ ⟷ *is-constant z*)
and *combine mk-until* (*x*, *i*) (*y*, *j*) = (*z*, *k*) ⟹ (*k* = ∞ ⟷ *is-constant z*)
and *combine mk-release* (*x*, *i*) (*y*, *j*) = (*z*, *k*) ⟹ (*k* = ∞ ⟷ *is-constant z*)

and *combine mk-weak-until* $(x,\ i)\ (y,\ j) = (z,\ k) \Longrightarrow (k = \infty \longleftrightarrow$ *is-constant* $z)$
and *combine mk-strong-release* $(x,\ i)\ (y,\ j) = (z,\ k) \Longrightarrow (k = \infty \longleftrightarrow$ *is-constant* $z)$
by (*cases* i; *cases* j; *simp add*: *assms Let-def*; *force intro*: *is-constant-constructorsI*)+

lemma *combine-and-or-semantics*:
assumes $i = \infty \longleftrightarrow$ *is-constant* φ
assumes $j = \infty \longleftrightarrow$ *is-constant* ψ
shows $w \models_n$ *to-ltln* (*combine mk-and* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,$ $i)$ *and*$_n$ *to-ltln* $(\psi,\ j)$
and $w \models_n$ *to-ltln* (*combine mk-or* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,\ i)$ *or*$_n$ *to-ltln* $(\psi,\ j)$
by ((*cases* i; *cases* j; *simp add*: *min-def is-constant-constructors-simps is-constant-constructors-simps2 assms*),
(*cases* ψ; *insert assms*; *auto*),
(*cases* φ; *insert assms*; *auto*),
(*blast elim*!: *is-constant.elims*))+

lemma *combine-until-release-semantics*:
assumes $i = \infty \longleftrightarrow$ *is-constant* φ
assumes $j = \infty \longleftrightarrow$ *is-constant* ψ
shows $w \models_n$ *to-ltln* (*combine mk-until* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,\ i)\ U_n$ *to-ltln* $(\psi,\ j)$
and $w \models_n$ *to-ltln* (*combine mk-release* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,\ i)\ R_n$ *to-ltln* $(\psi,\ j)$
by ((*cases* i; *cases* j; *simp add*: *is-constant-constructors-simps is-constant-constructors-simps2 until-constant-simp release-constant-simp mk-next-pow-until mk-next-pow-release del*: *semantics-ltln.simps*),
(*blast dest*: *is-constant-semantics*),
(*cases* ψ; *simp add*: *assms*),
(*cases* φ; *insert assms*; *auto simp*: *add.commute*))+

lemma *combine-weak-until-strong-release-semantics*:
assumes $i = \infty \longleftrightarrow$ *is-constant* φ
assumes $j = \infty \longleftrightarrow$ *is-constant* ψ
shows $w \models_n$ *to-ltln* (*combine mk-weak-until* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,\ i)\ W_n$ *to-ltln* $(\psi,\ j)$
and $w \models_n$ *to-ltln* (*combine mk-strong-release* $(\varphi,\ i)\ (\psi,\ j)) \longleftrightarrow w \models_n$ *to-ltln* $(\varphi,\ i)\ M_n$ *to-ltln* $(\psi,\ j)$
by ((*cases* i; *cases* j; *simp add*: *min-def is-constant-constructors-simps is-constant-constructors-simps3 del*: *semantics-ltln.simps*),
(*cases* φ; *simp add*: *assms*),

(*cases* ψ; *insert assms*; *auto simp*: *add.commute*))+

lemma *rewrite-X-enat-infinity-invariant*:
snd (*rewrite-X-enat* φ) = ∞ $\longleftrightarrow$ *is-constant* (*fst* (*rewrite-X-enat* φ))
proof (*induction* φ)
case (*And-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF And-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*Or-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF Or-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*Until-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF Until-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*Release-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF Release-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*WeakUntil-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF WeakUntil-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*StrongRelease-ltln* φ ψ)
thus *?case*
by (*simp add*: *combine-infinity-invariant*[*OF StrongRelease-ltln*(*1*,*2*), *unfolded prod.collapse*])
next
case (*Next-ltln* φ)
thus *?case*
by (*simp add*: *split-def*) (*metis eSuc-infinity eSuc-inject*)
qed *auto*

lemma *rewrite-X-enat-correct*:
$w \models_n \varphi \longleftrightarrow w \models_n$ *to-ltln* (*rewrite-X-enat* φ)

proof (*induction* φ *arbitrary*: w)
case (*And-ltln* φ ψ)
thus *?case*
using *combine-and-or-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*] **by** *fastforce*
next
case (*Or-ltln* φ ψ)
thus *?case*
using *combine-and-or-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*] **by** *fastforce*
next
case (*Until-ltln* φ ψ)
thus *?case*
unfolding *rewrite-X-enat.simps combine-until-release-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*, *unfolded prod.collapse*] **by** *fastforce*
next
case (*Release-ltln* φ ψ)
thus *?case*
unfolding *rewrite-X-enat.simps combine-until-release-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*, *unfolded prod.collapse*] **by** *fastforce*
next
case (*WeakUntil-ltln* φ ψ)
thus *?case*
unfolding *rewrite-X-enat.simps combine-weak-until-strong-release-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*, *unfolded prod.collapse*] **by** *fastforce*
next
case (*StrongRelease-ltln* φ ψ)
thus *?case*
unfolding *rewrite-X-enat.simps combine-weak-until-strong-release-semantics*[*OF rewrite-X-enat-infinity-invariant rewrite-X-enat-infinity-invariant*, *unfolded prod.collapse*] **by** *fastforce*
next
case (*Next-ltln* φ)
moreover
have $w \models_n$ *to-ltln* (*rewrite-X-enat* (X_n φ)) $\longleftrightarrow$ *suffix 1* $w \models_n$ *to-ltln* (*rewrite-X-enat* φ)
by (*simp add*: *split-def*; *cases snd* (*rewrite-X-enat* φ) $\neq \infty$)
(*auto simp*: *eSuc-def*, *auto simp*: *rewrite-X-enat-infinity-invariant eSuc-def dest*: *is-constant-semantics*)
ultimately
show *?case*

using *semantics-ltln.simps*(7) **by** *blast*
qed *auto*

lemma *rewrite-X-sound* [*simp*]:
$w \models_n$ *rewrite-X* $\varphi \longleftrightarrow w \models_n \varphi$
using *rewrite-X-enat-correct* **unfolding** *rewrite-X-def Let-def* **by** *auto*

2.4 Pure Eventual, Pure Universal, and Suspendable Formulas

fun *pure-eventual* :: *'a ltln* ⇒ *bool*
where
pure-eventual $true_n$ = *True*
| *pure-eventual* $false_n$ = *True*
| *pure-eventual* (μ and_n μ') = (*pure-eventual* μ ∧ *pure-eventual* μ')
| *pure-eventual* (μ or_n μ') = (*pure-eventual* μ ∧ *pure-eventual* μ')
| *pure-eventual* (μ U_n μ') = ($\mu = true_n$ ∨ *pure-eventual* μ')
| *pure-eventual* (μ R_n μ') = (*pure-eventual* μ ∧ *pure-eventual* μ')
| *pure-eventual* (μ W_n μ') = (*pure-eventual* μ ∧ *pure-eventual* μ')
| *pure-eventual* (μ M_n μ') = (*pure-eventual* μ ∧ *pure-eventual* μ' ∨ $\mu' = true_n$)
| *pure-eventual* (X_n μ) = *pure-eventual* μ
| *pure-eventual* - = *False*

fun *pure-universal* :: *'a ltln* ⇒ *bool*
where
pure-universal $true_n$ = *True*
| *pure-universal* $false_n$ = *True*
| *pure-universal* (ν and_n ν') = (*pure-universal* ν ∧ *pure-universal* ν')
| *pure-universal* (ν or_n ν') = (*pure-universal* ν ∧ *pure-universal* ν')
| *pure-universal* (ν U_n ν') = (*pure-universal* ν ∧ *pure-universal* ν')
| *pure-universal* (ν R_n ν') = ($\nu = false_n$ ∨ *pure-universal* ν')
| *pure-universal* (ν W_n ν') = (*pure-universal* ν ∧ *pure-universal* ν' ∨ $\nu' = false_n$)
| *pure-universal* (ν M_n ν') = (*pure-universal* ν ∧ *pure-universal* ν')
| *pure-universal* (X_n ν) = *pure-universal* ν
| *pure-universal* - = *False*

fun *suspendable* :: *'a ltln* ⇒ *bool*
where
suspendable $true_n$ = *True*
| *suspendable* $false_n$ = *True*
| *suspendable* (ξ and_n ξ') = (*suspendable* ξ ∧ *suspendable* ξ')
| *suspendable* (ξ or_n ξ') = (*suspendable* ξ ∧ *suspendable* ξ')

| *suspendable* (φ U_n ξ) = (φ = $true_n$ $\wedge$ *pure-universal* ξ $\vee$ *suspendable* ξ)
| *suspendable* (φ R_n ξ) = (φ = $false_n$ $\wedge$ *pure-eventual* ξ $\vee$ *suspendable* ξ)
| *suspendable* (φ W_n ξ) = (*suspendable* φ $\wedge$ *suspendable* ξ $\vee$ *pure-eventual* φ $\wedge$ ξ = $false_n$)
| *suspendable* (φ M_n ξ) = (*suspendable* φ $\wedge$ *suspendable* ξ $\vee$ *pure-universal* φ $\wedge$ ξ = $true_n$)
| *suspendable* (X_n ξ) = *suspendable* ξ
| *suspendable* - = *False*

lemma *pure-eventual-left-append*:
pure-eventual $\mu \Longrightarrow w \models_n \mu \Longrightarrow (u \frown w) \models_n \mu$
proof (*induction* μ *arbitrary*: u w)
case (*Until-ltln* μ μ')
moreover
then obtain i **where** *suffix* i $w \models_n \mu'$
by *auto*
hence $\mu = true_n \Longrightarrow$ *?case*
by *simp* (*metis suffix-conc-length suffix-suffix*)
moreover
have *pure-eventual* $\mu' \Longrightarrow (u \frown w) \models_n \mu'$
by (*metis* ‹*suffix i w* $\models_n \mu'$› *Until-ltln*(*2*) *prefix-suffix*)
hence *pure-eventual* $\mu' \Longrightarrow$ *?case*
by *force*
ultimately
show *?case*
by *fastforce*
next
case (*Release-ltln* μ μ')
thus *?case*
by (*cases* $\forall i$. *suffix* i $w \models_n \mu'$; *simp-all*)
(*metis linear suffix-conc-snd gr0I not-less0 prefix-suffix suffix-0*)+
next
case (*WeakUntil-ltln* μ μ')
thus *?case*
by (*cases* $\forall i$. *suffix* i $w \models_n \mu'$; *simp-all*)
(*metis zero-le le0 nat-le-linear prefix-suffix suffix-0 suffix-conc-length suffix-conc-snd suffix-subseq-join*)+
next
case (*StrongRelease-ltln* μ μ')
moreover
then obtain i **where** *suffix* i $w \models_n \mu$ *and*$_n$ μ'
by *auto*
hence $\mu' = true_n \Longrightarrow$ *?case*
by *simp* (*metis suffix-conc-length suffix-suffix*)

moreover
have *pure-eventual* $\mu \Longrightarrow$ *pure-eventual* $\mu' \Longrightarrow (u \frown w) \models_n \mu$ *and*$_n$ μ'
by (*metis* ‹*suffix i w* $\models_n \mu$ *and*$_n$ μ'› *calculation*(*1*) *calculation*(*2*) *prefix-suffix semantics-ltln.simps*(*5*))
hence *pure-eventual* $\mu \Longrightarrow$ *pure-eventual* $\mu' \Longrightarrow$ *?case*
by *force*
ultimately
show *?case*
by *fastforce*
qed (*auto*, *metis diff-zero le-0-eq not-less-eq-eq suffix-conc-length suffix-conc-snd word-split*)

lemma *pure-universal-suffix-closed*:
pure-universal $\nu \Longrightarrow (u \frown w) \models_n \nu \Longrightarrow w \models_n \nu$
proof (*induction* ν *arbitrary*: u w)
case (*Until-ltln* ν ν')
hence $\exists i.$ *suffix* i $(u \frown w) \models_n \nu' \wedge (\forall j{<}i.$ *suffix* j $(u \frown w) \models_n \nu)$
using *semantics-ltln.simps*(*8*) **by** *blast*
thus *?case*
by *simp* (*metis Until-ltln*(*1−3*) *le-0-eq le-eq-less-or-eq le-less-linear prefix-suffix pure-universal.simps*(*5*) *suffix-conc-fst suffix-conc-snd*)
next
case (*Release-ltln* ν ν')
moreover
hence $\forall i.$ *suffix* i $(u \frown w) \models_n \nu' \vee (\exists j{<}i.$ *suffix* j $(u \frown w) \models_n \nu)$
by *simp*
ultimately
show *?case*
by *simp* (*metis semantics-ltln.simps*(*2*) *not-less0 prefix-suffix suffix-0 suffix-conc-length suffix-suffix*)
next
case (*WeakUntil-ltln* ν ν')
moreover
hence $\forall i.$ *suffix* i $(u \frown w) \models_n \nu \vee (\exists j{\leq}i.$ *suffix* j $(u \frown w) \models_n \nu')$
by *simp*
ultimately
show *?case*
by *simp* (*metis* (*full-types*) *le-antisym prefix-suffix semantics-ltln.simps*(*2*) *suffix-0 suffix-conc-length suffix-suffix zero-le*)
next
case (*StrongRelease-ltln* ν ν')
hence $\exists i.$ *suffix* i $(u \frown w) \models_n \nu \wedge (\forall j{\leq}i.$ *suffix* j $(u \frown w) \models_n \nu')$
using *semantics-ltln.simps*(*11*) **by** *blast*
thus *?case*

by *simp* (*metis StrongRelease-ltln*(*1−3*) *diff-is-0-eq nat-le-linear prefix-conc-length prefix-suffix pure-universal.simps*(*8*) *subsequence-length suffix-conc-snd suffix-subseq-join*)
next
case (*Next-ltln* μ)
thus *?case*
by (*metis prefix-suffix pure-universal.simps*(*9*) *semantics-ltln.simps*(*7*) *semiring-normalization-rules*(*24*) *suffix-conc-length suffix-suffix*)
qed *auto*

lemma *suspendable-prefix-invariant*:
suspendable $\xi \Longrightarrow (u \frown w) \models_n \xi \longleftrightarrow w \models_n \xi$
proof (*induction* ξ *arbitrary*: u w)
case (*Until-ltln* ξ ξ')
show *?case*
proof (*cases suspendable* ξ')
case *False*
hence $\xi = true_n$ **and** *pure-universal* ξ'
using *Until-ltln* **by** *simp+*
thus *?thesis*
by (*simp*; *metis* (*no-types*) *linear pure-universal-suffix-closed suffix-conc-fst suffix-conc-length suffix-conc-snd suffix-suffix*)
qed (*simp*; *metis Until-ltln*(*2*) *not-less0 prefix-suffix*)
next
case (*Release-ltln* ξ ξ')
show *?case*
proof (*cases suspendable* ξ')
case *False*
hence $\xi = false_n$ **and** *pure-eventual* ξ'
using *Release-ltln* **by** *simp+*
thus *?thesis*
by (*simp*; *metis* (*no-types*) *le-iff-add add-diff-cancel-left′ linear pure-eventual-left-append suffix-0 suffix-conc-fst suffix-conc-snd*)
qed (*simp*; *metis Release-ltln*(*2*) *not-less0 prefix-suffix*)
next
case (*WeakUntil-ltln* ξ ξ')
show *?case*
proof (*cases suspendable* $\xi \wedge$ *suspendable* ξ')
case *False*
hence $\xi' = false_n$ **and** *pure-eventual* ξ
using *WeakUntil-ltln* **by** *simp+*
thus *?thesis*
by (*simp*; *metis* (*no-types*) *le-iff-add add-diff-cancel-left′ linear pure-eventual-left-append suffix-0 suffix-conc-fst suffix-conc-snd*)

qed (*simp*; *metis* (*full-types*) *WeakUntil-ltln.IH prefix-suffix*)
next
case (*StrongRelease-ltln* ξ ξ')
show *?case*
proof (*cases suspendable* $\xi \wedge$ *suspendable* ξ')
case *False*
hence $\xi' = true_n$ **and** *pure-universal* ξ
using *StrongRelease-ltln* **by** *simp+*
thus *?thesis*
by (*simp*; *metis* (*no-types*) *linear pure-universal-suffix-closed suffix-conc-fst suffix-conc-length suffix-conc-snd suffix-suffix*)
qed (*simp*; *metis* (*full-types*) *StrongRelease-ltln.IH*(*1*) *StrongRelease-ltln.IH*(*2*) *prefix-suffix*)
qed (*simp-all*, *metis prefix-suffix*)

theorem *pure-eventual-until-simp*:
assumes *pure-eventual* μ
shows $w \models_n \varphi\ U_n\ \mu \longleftrightarrow w \models_n \mu$
proof −
have $\bigwedge i.$ *suffix* $i\ w \models_n \mu \Longrightarrow w \models_n \mu$
using *pure-eventual-left-append*[*OF assms*] *prefix-suffix* **by** *metis*
thus *?thesis*
by *force*
qed

theorem *pure-universal-release-simp*:
assumes *pure-universal* ν
shows $w \models_n \varphi\ R_n\ \nu \longleftrightarrow w \models_n \nu$
proof −
have $\bigwedge i.\ w \models_n \nu \Longrightarrow$ *suffix* $i\ w \models_n \nu$
using *pure-universal-suffix-closed*[*OF assms*] *prefix-suffix* **by** *metis*
thus *?thesis*
by *force*
qed

theorem *pure-universal-weak-until-simp*:
assumes *pure-universal* φ **and** *pure-universal* ψ
shows $w \models_n \varphi\ W_n\ \psi \longleftrightarrow w \models_n \varphi\ or_n\ \psi$
proof −
have $\bigwedge i.\ w \models_n \varphi \Longrightarrow$ *suffix* $i\ w \models_n \varphi$ **and** $\bigwedge i.\ w \models_n \psi \Longrightarrow$ *suffix* $i\ w \models_n \psi$
using *assms pure-universal-suffix-closed prefix-suffix* **by** *metis+*
thus *?thesis*
by *force*

qed

theorem *pure-eventual-strong-release-simp*:
assumes *pure-eventual* φ **and** *pure-eventual* ψ
shows $w \models_n \varphi\ M_n\ \psi \longleftrightarrow w \models_n \varphi\ and_n\ \psi$
proof −
have $\bigwedge i.$ *suffix* $i\ w \models_n \varphi \Longrightarrow w \models_n \varphi$ **and** $\bigwedge i.$ *suffix* $i\ w \models_n \psi \Longrightarrow w \models_n \psi$
using *assms pure-eventual-left-append prefix-suffix* **by** *metis+*
thus *?thesis*
by *force*
qed

theorem *suspendable-formula-simp*:
assumes *suspendable* ξ
shows $w \models_n X_n\ \xi \longleftrightarrow w \models_n \xi$ (**is** *?t1*)
and $w \models_n \varphi\ U_n\ \xi \longleftrightarrow w \models_n \xi$ (**is** *?t2*)
and $w \models_n \varphi\ R_n\ \xi \longleftrightarrow w \models_n \xi$ (**is** *?t3*)
proof −
have $\bigwedge i.$ *suffix* $i\ w \models_n \xi \longleftrightarrow w \models_n \xi$
using *suspendable-prefix-invariant*[*OF assms*] *prefix-suffix* **by** *metis*
thus *?t1 ?t2 ?t3*
by *force+*
qed

theorem *suspendable-formula-simp2*:
assumes *suspendable* φ **and** *suspendable* ψ
shows $w \models_n \varphi\ W_n\ \psi \longleftrightarrow w \models_n \varphi\ or_n\ \psi$ (**is** *?t1*)
and $w \models_n \varphi\ M_n\ \psi \longleftrightarrow w \models_n \varphi\ and_n\ \psi$ (**is** *?t2*)
proof −
have $\bigwedge i.$ *suffix* $i\ w \models_n \varphi \longleftrightarrow w \models_n \varphi$ **and** $\bigwedge i.$ *suffix* $i\ w \models_n \psi \longleftrightarrow w \models_n \psi$
using *assms suspendable-prefix-invariant prefix-suffix* **by** *metis+*
thus *?t1 ?t2*
by *force+*
qed

fun *rewrite-modal* :: *'a ltln* ⇒ *'a ltln*
where
rewrite-modal $true_n$ = $true_n$
| *rewrite-modal* $false_n$ = $false_n$
| *rewrite-modal* ($\varphi\ and_n\ \psi$) = (*rewrite-modal* $\varphi\ and_n$ *rewrite-modal* ψ)
| *rewrite-modal* ($\varphi\ or_n\ \psi$) = (*rewrite-modal* $\varphi\ or_n$ *rewrite-modal* ψ)

| *rewrite-modal* (φ U_n ψ) = (*if pure-eventual* ψ $\vee$ *suspendable* ψ *then* *rewrite-modal* ψ *else* (*rewrite-modal* φ U_n *rewrite-modal* ψ))
| *rewrite-modal* (φ R_n ψ) = (*if pure-universal* ψ $\vee$ *suspendable* ψ *then* *rewrite-modal* ψ *else* (*rewrite-modal* φ R_n *rewrite-modal* ψ))
| *rewrite-modal* (φ W_n ψ) = (*if pure-universal* φ $\wedge$ *pure-universal* ψ $\vee$ *suspendable* φ $\wedge$ *suspendable* ψ *then* (*rewrite-modal* φ or_n *rewrite-modal* ψ) *else* (*rewrite-modal* φ W_n *rewrite-modal* ψ))
| *rewrite-modal* (φ M_n ψ) = (*if pure-eventual* φ $\wedge$ *pure-eventual* ψ $\vee$ *suspendable* φ $\wedge$ *suspendable* ψ *then* (*rewrite-modal* φ and_n *rewrite-modal* ψ) *else* (*rewrite-modal* φ M_n *rewrite-modal* ψ))
| *rewrite-modal* (X_n φ) = (*if suspendable* φ *then rewrite-modal* φ *else* X_n (*rewrite-modal* φ))
| *rewrite-modal* φ = φ

lemma *rewrite-modal-sound* [*simp*]:
$w \models_n$ *rewrite-modal* $\varphi \longleftrightarrow w \models_n \varphi$
proof (*induction* φ *arbitrary*: w)
case (*Until-ltln* φ ψ)
thus *?case*
apply (*cases pure-eventual* ψ $\vee$ *suspendable* ψ)
apply (*insert pure-eventual-until-simp*[*of* ψ] *suspendable-formula-simp*[*of* ψ])
apply *fastforce*+
done
next
case (*Release-ltln* φ ψ)
thus *?case*
apply (*cases pure-universal* ψ $\vee$ *suspendable* ψ)
apply (*insert pure-universal-release-simp*[*of* ψ] *suspendable-formula-simp*[*of* ψ])
apply *fastforce*+
done
next
case (*WeakUntil-ltln* φ ψ)
thus *?case*
apply (*cases pure-universal* φ $\wedge$ *pure-universal* ψ $\vee$ *suspendable* φ $\wedge$ *suspendable* ψ)
apply (*insert pure-universal-weak-until-simp*[*of* φ ψ] *suspendable-formula-simp2*[*of* φ ψ])
apply *fastforce*+
done
next
case (*StrongRelease-ltln* φ ψ)
thus *?case*

apply (*cases pure-eventual* φ $\wedge$ *pure-eventual* ψ $\vee$ *suspendable* φ $\wedge$ *suspendable* ψ)
apply (*insert pure-eventual-strong-release-simp*[*of* φ ψ] *suspendable-formula-simp2*[*of* φ ψ])
apply *fastforce*+
done
next
case (*Next-ltln* φ)
thus *?case*
apply (*cases suspendable* φ)
apply (*insert suspendable-formula-simp*[*of* φ])
apply *fastforce*+
done
qed *auto*

lemma *rewrite-modal-size*:
size (*rewrite-modal* φ) $\leq$ *size* φ
by (*induction* φ) *auto*

2.5 Syntactical Implication Based Simplification

inductive *syntactical-implies* :: *'a ltln* $\Rightarrow$ *'a ltln* $\Rightarrow$ *bool* (‹- $\vdash_s$ -› [*80*, *80*])
where
- $\vdash_s$ $true_n$
| $false_n$ $\vdash_s$ -
| $\varphi = \varphi \Longrightarrow \varphi \vdash_s \varphi$

| $\varphi \vdash_s \chi \Longrightarrow (\varphi\ and_n\ \psi) \vdash_s \chi$
| $\psi \vdash_s \chi \Longrightarrow (\varphi\ and_n\ \psi) \vdash_s \chi$
| $\varphi \vdash_s \psi \Longrightarrow \varphi \vdash_s \chi \Longrightarrow \varphi \vdash_s (\psi\ and_n\ \chi)$

| $\varphi \vdash_s \psi \Longrightarrow \varphi \vdash_s (\psi\ or_n\ \chi)$
| $\varphi \vdash_s \chi \Longrightarrow \varphi \vdash_s (\psi\ or_n\ \chi)$
| $\varphi \vdash_s \chi \Longrightarrow \psi \vdash_s \chi \Longrightarrow (\varphi\ or_n\ \psi) \vdash_s \chi$

| $\varphi \vdash_s \chi \Longrightarrow \varphi \vdash_s (\psi\ U_n\ \chi)$
| $\varphi \vdash_s \chi \Longrightarrow \psi \vdash_s \chi \Longrightarrow (\varphi\ U_n\ \psi) \vdash_s \chi$
| $\varphi \vdash_s \chi \Longrightarrow \psi \vdash_s \nu \Longrightarrow (\varphi\ U_n\ \psi) \vdash_s (\chi\ U_n\ \nu)$

| $\chi \vdash_s \varphi \Longrightarrow (\psi\ R_n\ \chi) \vdash_s \varphi$
| $\chi \vdash_s \varphi \Longrightarrow \chi \vdash_s \psi \Longrightarrow \chi \vdash_s (\varphi\ R_n\ \psi)$
| $\varphi \vdash_s \chi \Longrightarrow \psi \vdash_s \nu \Longrightarrow (\varphi\ R_n\ \psi) \vdash_s (\chi\ R_n\ \nu)$

| $(false_n\ R_n\ \varphi) \vdash_s \psi \Longrightarrow (false_n\ R_n\ \varphi) \vdash_s X_n\ \psi$

$| \ \varphi \vdash_s (true_n\ U_n\ \psi) \Longrightarrow (X_n\ \varphi) \vdash_s (true_n\ U_n\ \psi)$
$| \ \varphi \vdash_s \psi \Longrightarrow (X_n\ \varphi) \vdash_s (X_n\ \psi)$

lemma *syntactical-implies-correct*:
$\varphi \vdash_s \psi \Longrightarrow w \models_n \varphi \Longrightarrow w \models_n \psi$
by (*induction arbitrary*: *w rule*: *syntactical-implies.induct*; *auto*; *force*)

fun *rewrite-syn-imp*
where
rewrite-syn-imp $(\varphi\ and_n\ \psi) =$ (
if $\varphi \vdash_s \psi$ *then*
rewrite-syn-imp φ
else if $\psi \vdash_s \varphi$ *then*
rewrite-syn-imp ψ
else if $\varphi \vdash_s (not_n\ \psi) \vee \psi \vdash_s (not_n\ \varphi)$ *then*
$false_n$
else
mk-and (*rewrite-syn-imp* φ) (*rewrite-syn-imp* ψ))
| *rewrite-syn-imp* $(\varphi\ or_n\ \psi) =$ (
if $\varphi \vdash_s \psi$ *then*
rewrite-syn-imp ψ
else if $\psi \vdash_s \varphi$ *then*
rewrite-syn-imp φ
else if $(not_n\ \varphi) \vdash_s \psi \vee (not_n\ \psi) \vdash_s \varphi$ *then*
$true_n$
else
mk-or (*rewrite-syn-imp* φ) (*rewrite-syn-imp* ψ))
| *rewrite-syn-imp* $(\varphi\ U_n\ \psi) =$ (
if $\varphi \vdash_s \psi$ *then*
rewrite-syn-imp ψ
else if $(not_n\ \varphi) \vdash_s \psi$ *then*
mk-finally (*rewrite-syn-imp* ψ)
else
mk-until (*rewrite-syn-imp* φ) (*rewrite-syn-imp* ψ))
| *rewrite-syn-imp* $(\varphi\ R_n\ \psi) =$ (
if $\psi \vdash_s \varphi$ *then*
rewrite-syn-imp ψ
else if $\psi \vdash_s (not_n\ \varphi)$ *then*
mk-globally (*rewrite-syn-imp* ψ)
else
mk-release (*rewrite-syn-imp* φ) (*rewrite-syn-imp* ψ))
| *rewrite-syn-imp* $(X_n\ \varphi) =$ *mk-next* (*rewrite-syn-imp* φ)
| *rewrite-syn-imp* $\varphi = \varphi$

lemma *rewrite-syn-imp-sound*:
  $w \models_n$ *rewrite-syn-imp* $\varphi \longleftrightarrow w \models_n \varphi$
proof (*induction* φ *arbitrary*: w)
  case *And-ltln*
    thus *?case*
     by (*simp add*: *Let-def*; *metis syntactical-implies-correct not*$_n$*-semantics*)
next
  case (*Or-ltln* φ ψ)
    moreover
    have (*not*$_n$ φ) $\vdash_s \psi \Longrightarrow \forall w.\ w \models_n \varphi$ *or*$_n$ ψ
      by (*auto intro*: *syntactical-implies-correct*[*of not*$_n$ φ])
    moreover
    have (*not*$_n$ ψ) $\vdash_s \varphi \Longrightarrow \forall w.\ w \models_n \varphi$ *or*$_n$ ψ
      by (*auto intro*: *syntactical-implies-correct*[*of not*$_n$ ψ])
    ultimately
    show *?case*
      by (*auto intro*: *syntactical-implies-correct*)
next
  case (*Until-ltln* φ ψ)
    moreover
    have $\varphi \vdash_s \psi \Longrightarrow$ *?case*
      by (*force simp add*: *Until-ltln dest*: *syntactical-implies-correct*)
    moreover
    {
     assume *A*: (*not*$_n$ φ) $\vdash_s \psi$ **and** *B*: $\neg\ \varphi \vdash_s \psi$
     hence [*simp*]: *rewrite-syn-imp* (φ U_n ψ) = *mk-finally* (*rewrite-syn-imp* ψ)
       by *simp*
      {
       assume $\exists i.$ *suffix i w* $\models_n \psi$
       moreover
       define i **where** $i \equiv$ *LEAST i. suffix i w* $\models_n \psi$
       ultimately
       have $\forall j < i. \neg$*suffix j w* $\models_n \psi$ **and** *suffix i w* $\models_n \psi$
         by (*blast dest*: *not-less-Least* , *metis LeastI* ‹$\exists i.$ *suffix i w* $\models_n \psi$› *i-def*)
       hence $\forall j < i.$ *suffix j w* $\models_n \varphi$ **and** *suffix i w* $\models_n \psi$
         using *syntactical-implies-correct*[*OF A*] **by** *auto*
      }
      hence *?case*
       by (*simp del*: *rewrite-syn-imp.simps*; *unfold Until-ltln*(*2*)) *blast*
    }
    ultimately
    show *?case*

by *fastforce*
next
case (*Release-ltln* φ ψ)
moreover
have $\psi \vdash_s \varphi \Longrightarrow$ *?case*
by (*force simp add*: *Release-ltln dest*: *syntactical-implies-correct*)
moreover
{
assume *A*: $\psi \vdash_s (not_n\ \varphi)$ **and** *B*: $\neg\ \psi \vdash_s \varphi$
hence [*simp*]: *rewrite-syn-imp* ($\varphi\ R_n\ \psi$) = *mk-globally* (*rewrite-syn-imp* ψ)
by *simp*
{
assume $\exists\, i.\ \neg$*suffix i w* $\models_n \psi$
moreover
define *i* **where** $i \equiv$ *LEAST i.* $\neg$*suffix i w* $\models_n \psi$
ultimately
have $\forall\, j < i.$ *suffix j w* $\models_n \psi$ **and** $\neg$ *suffix i w* $\models_n \psi$
by (*blast dest*: *not-less-Least* , *metis LeastI* ‹$\exists\, i.$ $\neg$*suffix i w* $\models_n \psi$› *i-def*)
hence $\forall\, j < i.$ $\neg$*suffix j w* $\models_n \varphi$ **and** $\neg$ *suffix i w* $\models_n \psi$
using *syntactical-implies-correct*[*OF A*] **by** *auto*
}
hence *?case*
by (*simp del*: *rewrite-syn-imp.simps*; *unfold Release-ltln*(*2*)) *blast*
}
ultimately
show *?case*
by *fastforce*
qed *auto*

2.6 Iterated Rewriting

fun *iterate*
where
iterate f x 0 = *x*
| *iterate f x* (*Suc n*) = (*let x′* = *f x in if x* = *x′ then x else iterate f x′ n*)

definition
rewrite-iter-fast $\varphi \equiv$ *iterate* (*rewrite-modal o rewrite-X*) φ (*size* φ)

definition
rewrite-iter-slow $\varphi \equiv$ *iterate* (*rewrite-syn-imp o rewrite-modal o rewrite-X*) φ (*size* φ)

The rewriting functions defined in the previous subsections can be used as-is. However, in the most cases one wants to iterate these rules until the formula cannot be simplified further. *rewrite-iter-fast* pulls X operators up in the syntax tree and the uses the "modal" simplification rules. *rewrite-iter-slow* additionally tries to simplify the formula using syntactic implication checking.

lemma *iterate-sound*:
assumes $\bigwedge\varphi.\ w \models_n f\ \varphi \longleftrightarrow w \models_n \varphi$
shows $w \models_n$ *iterate* $f\ \varphi\ n \longleftrightarrow w \models_n \varphi$
by (*induction n arbitrary*: φ; *simp add*: *assms Let-def*)

theorem *rewrite-iter-fast-sound* [*simp*]:
$w \models_n$ *rewrite-iter-fast* $\varphi \longleftrightarrow w \models_n \varphi$
using *iterate-sound*[*of - rewrite-modal o rewrite-X*]
unfolding *comp-def rewrite-modal-sound rewrite-X-sound rewrite-iter-fast-def*
by *blast*

theorem *rewrite-iter-slow-sound* [*simp*]:
$w \models_n$ *rewrite-iter-slow* $\varphi \longleftrightarrow w \models_n \varphi$
using *iterate-sound*[*of - rewrite-syn-imp o rewrite-modal o rewrite-X*]
unfolding *comp-def rewrite-modal-sound rewrite-X-sound rewrite-syn-imp-sound rewrite-iter-slow-def*
by *blast*

2.7 Preservation of atoms

lemma *iterate-atoms*:
assumes
$\bigwedge\varphi.$ *atoms-ltln* $(f\ \varphi) \subseteq$ *atoms-ltln* φ
shows
atoms-ltln (*iterate* $f\ \varphi\ n) \subseteq$ *atoms-ltln* φ
by (*induction n arbitrary*: φ) (*auto*, *metis* (*mono-tags*, *lifting*) *assms in-mono*)

lemma *rewrite-modal-atoms*:
atoms-ltln (*rewrite-modal* $\varphi) \subseteq$ *atoms-ltln* φ
by (*induction* φ) *auto*

lemma *mk-and-atoms*:
atoms-ltln (*mk-and* $\varphi\ \psi) \subseteq$ *atoms-ltln* $\varphi \cup$ *atoms-ltln* ψ
by (*auto simp*: *mk-and-def split*: *ltln.splits*)

lemma *mk-or-atoms*:
atoms-ltln (*mk-or* $\varphi\ \psi) \subseteq$ *atoms-ltln* $\varphi \cup$ *atoms-ltln* ψ

by (*auto simp*: *mk-or-def split*: *ltln.splits*)

lemma *remove-strong-ops-atoms*:
atoms-ltln (*remove-strong-ops* φ) $\subseteq$ *atoms-ltln* φ
by (*induction* φ) *auto*

lemma *remove-weak-ops-atoms*:
atoms-ltln (*remove-weak-ops* φ) $\subseteq$ *atoms-ltln* φ
by (*induction* φ) *auto*

lemma *mk-finally-atoms*:
atoms-ltln (*mk-finally* φ) $\subseteq$ *atoms-ltln* φ
by (*auto simp*: *mk-finally-def split*: *ltln.splits*) (*insert remove-strong-ops-atoms*, *fast+*)

lemma *mk-globally-atoms*:
atoms-ltln (*mk-globally* φ) $\subseteq$ *atoms-ltln* φ
by (*auto simp*: *mk-globally-def split*: *ltln.splits*) (*insert remove-weak-ops-atoms*, *fast+*)

lemma *mk-until-atoms*:
atoms-ltln (*mk-until* φ ψ) $\subseteq$ *atoms-ltln* φ $\cup$ *atoms-ltln* ψ
by (*auto simp*: *mk-until-def split*: *ltln.splits*) (*insert mk-finally-atoms*, *fast-force+*)

lemma *mk-release-atoms*:
atoms-ltln (*mk-release* φ ψ) $\subseteq$ *atoms-ltln* φ $\cup$ *atoms-ltln* ψ
by (*auto simp*: *mk-release-def split*: *ltln.splits*) (*insert mk-globally-atoms*, *fastforce+*)

lemma *mk-weak-until-atoms*:
atoms-ltln (*mk-weak-until* φ ψ) $\subseteq$ *atoms-ltln* φ $\cup$ *atoms-ltln* ψ
by (*auto simp*: *mk-weak-until-def split*: *ltln.splits*) (*insert mk-globally-atoms*, *fastforce+*)

lemma *mk-strong-release-atoms*:
atoms-ltln (*mk-strong-release* φ ψ) $\subseteq$ *atoms-ltln* φ $\cup$ *atoms-ltln* ψ
by (*auto simp*: *mk-strong-release-def split*: *ltln.splits*) (*insert mk-finally-atoms*, *fastforce+*)

lemma *mk-next-atoms*:
atoms-ltln (*mk-next* φ) = *atoms-ltln* φ
by (*auto simp*: *mk-next-def split*: *ltln.splits*)

lemma *mk-next-pow-atoms*:
atoms-ltln (*mk-next-pow n* φ) = *atoms-ltln* φ
by (*induction n*) (*auto simp*: *mk-next-pow-def split*: *ltln.splits*)

lemma *combine-atoms*:
assumes
$\bigwedge\varphi\ \psi$. *atoms-ltln* (*f* φ ψ) $\subseteq$ *atoms-ltln* φ $\cup$ *atoms-ltln* ψ
shows
atoms-ltln (*fst* (*combine f x y*)) $\subseteq$ *atoms-ltln* (*fst x*) $\cup$ *atoms-ltln* (*fst y*)
by (*metis assms fst-combine mk-next-pow-atoms prod.collapse*)

lemmas *combine-mk-atoms* =
combine-atoms[*OF mk-and-atoms*]
combine-atoms[*OF mk-or-atoms*]
combine-atoms[*OF mk-until-atoms*]
combine-atoms[*OF mk-release-atoms*]
combine-atoms[*OF mk-weak-until-atoms*]
combine-atoms[*OF mk-strong-release-atoms*]

lemma *rewrite-X-enat-atoms*:
atoms-ltln (*fst* (*rewrite-X-enat* φ)) $\subseteq$ *atoms-ltln* φ
by (*induction* φ) (*simp-all add*: *case-prod-beta*, *insert combine-mk-atoms*, *fast*+)

lemma *rewrite-X-atoms*:
atoms-ltln (*rewrite-X* φ) $\subseteq$ *atoms-ltln* φ
by (*induction* φ) (*simp-all add*: *rewrite-X-def mk-next-pow-atoms case-prod-beta*, *insert combine-mk-atoms*, *fast*+)

lemma *rewrite-syn-imp-atoms*:
atoms-ltln (*rewrite-syn-imp* φ) $\subseteq$ *atoms-ltln* φ
proof (*induction* φ)
case (*And-ltln* φ*1* φ*2*)
then show *?case*
using *mk-and-atoms* **by** *simp fast*
next
case (*Or-ltln* φ*1* φ*2*)
then show *?case*
using *mk-or-atoms* **by** *simp fast*
next
case (*Next-ltln* φ)
then show *?case*
using *mk-next-atoms* **by** *simp fast*
next

case (*Until-ltln* $\varphi 1$ $\varphi 2$)
then show *?case*
using *mk-finally-atoms mk-until-atoms* **by** *simp fast*
next
case (*Release-ltln* $\varphi 1$ $\varphi 2$)
then show *?case*
using *mk-globally-atoms mk-release-atoms* **by** *simp fast*
qed *simp-all*

lemma *rewrite-iter-fast-atoms*:
atoms-ltln (*rewrite-iter-fast* φ) $\subseteq$ *atoms-ltln* φ
proof −
have *1*: $\bigwedge\varphi$. *atoms-ltln* (*rewrite-modal* (*rewrite-X* φ)) $\subseteq$ *atoms-ltln* φ
using *rewrite-modal-atoms rewrite-X-atoms* **by** *force*

show *?thesis*
by (*simp add*: *rewrite-iter-fast-def 1 iterate-atoms*)
qed

lemma *rewrite-iter-slow-atoms*:
atoms-ltln (*rewrite-iter-slow* φ) $\subseteq$ *atoms-ltln* φ
proof −
have *1*: $\bigwedge\varphi$. *atoms-ltln* (*rewrite-syn-imp* (*rewrite-modal* (*rewrite-X* φ)))
$\subseteq$ *atoms-ltln* φ
using *rewrite-syn-imp-atoms rewrite-modal-atoms rewrite-X-atoms* **by**
force

show *?thesis*
by (*simp add*: *rewrite-iter-slow-def 1 iterate-atoms*)
qed

2.8 Simplifier

We now define a convenience wrapper for the rewriting engine

datatype *mode* = *Nop* | *Fast* | *Slow*

fun *simplify* :: *mode* ⇒ *'a ltln* ⇒ *'a ltln*
where
simplify Nop = *id*
| *simplify Fast* = *rewrite-iter-fast*
| *simplify Slow* = *rewrite-iter-slow*

theorem *simplify-correct*:

$w \models_n$ *simplify* $m\ \varphi \longleftrightarrow w \models_n \varphi$
by (*cases m*) *simp+*

lemma *simplify-atoms*:
atoms-ltln (*simplify m* φ) $\subseteq$ *atoms-ltln* φ
by (*cases m*) (*insert rewrite-iter-fast-atoms rewrite-iter-slow-atoms*, *fast-force+*)

2.9 Code Generation

code-pred *syntactical-implies* **.**

export-code *simplify* **checking**

lemma *rewrite-iter-fast* (F_n (G_n (X_n $prop_n(''a'')$))) = (F_n (G_n $prop_n(''a'')$))
by *eval*

lemma *rewrite-iter-fast* (X_n $prop_n(''a'')$ U_n (X_n $nprop_n(''a'')$)) = X_n ($prop_n(''a'')$ U_n $nprop_n(''a'')$)
by *eval*

lemma *rewrite-iter-slow* (X_n $prop_n(''a'')$ U_n (X_n $nprop_n(''a'')$)) = X_n (F_n $nprop_n(''a'')$)
by *eval*

end

3 Equivalence Relations for LTL formulas

theory *Equivalence-Relations*
imports
LTL
begin

3.1 Language Equivalence

definition *ltl-lang-equiv* :: *'a ltln* $\Rightarrow$ *'a ltln* $\Rightarrow$ *bool* (**infix** $\langle\sim_L\rangle$ *75*)
where
$\varphi \sim_L \psi \equiv \forall\, w.\ w \models_n \varphi \longleftrightarrow w \models_n \psi$

lemma *ltl-lang-equiv-equivp*:
equivp ($\sim_L$)
unfolding *ltl-lang-equiv-def*
by (*simp add*: *equivpI reflp-def symp-def transp-def*)

lemma *ltl-lang-equiv-and-true*[*simp*]:
$\varphi_1\ and_n\ \varphi_2 \sim_L true_n \longleftrightarrow \varphi_1 \sim_L true_n \wedge \varphi_2 \sim_L true_n$
unfolding *ltl-lang-equiv-def* **by** *auto*

lemma *ltl-lang-equiv-and-false*[*intro*, *simp*]:
$\varphi_1 \sim_L false_n \Longrightarrow \varphi_1\ and_n\ \varphi_2 \sim_L false_n$
$\varphi_2 \sim_L false_n \Longrightarrow \varphi_1\ and_n\ \varphi_2 \sim_L false_n$
unfolding *ltl-lang-equiv-def* **by** *auto*

lemma *ltl-lang-equiv-or-false*[*simp*]:
$\varphi_1\ or_n\ \varphi_2 \sim_L false_n \longleftrightarrow \varphi_1 \sim_L false_n \wedge \varphi_2 \sim_L false_n$
unfolding *ltl-lang-equiv-def* **by** *auto*

lemma *ltl-lang-equiv-or-const*[*intro*, *simp*]:
$\varphi_1 \sim_L true_n \Longrightarrow \varphi_1\ or_n\ \varphi_2 \sim_L true_n$
$\varphi_2 \sim_L true_n \Longrightarrow \varphi_1\ or_n\ \varphi_2 \sim_L true_n$
unfolding *ltl-lang-equiv-def* **by** *auto*

3.2 Propositional Equivalence

fun *ltl-prop-entailment* :: *'a ltln set* ⇒ *'a ltln* ⇒ *bool* (**infix** ‹$\models_P$› *80*)
where
$\mathcal{A} \models_P true_n = True$
| $\mathcal{A} \models_P false_n = False$
| $\mathcal{A} \models_P \varphi\ and_n\ \psi = (\mathcal{A} \models_P \varphi \wedge \mathcal{A} \models_P \psi)$
| $\mathcal{A} \models_P \varphi\ or_n\ \psi = (\mathcal{A} \models_P \varphi \vee \mathcal{A} \models_P \psi)$
| $\mathcal{A} \models_P \varphi = (\varphi \in \mathcal{A})$

lemma *ltl-prop-entailment-monotonI*[*intro*]:
$S \models_P \varphi \Longrightarrow S \subseteq S' \Longrightarrow S' \models_P \varphi$
by (*induction* φ) *auto*

lemma *ltl-models-equiv-prop-entailment*:
$w \models_n \varphi \longleftrightarrow \{\psi.\ w \models_n \psi\} \models_P \varphi$
by (*induction* φ) *auto*

definition *ltl-prop-equiv* :: *'a ltln* ⇒ *'a ltln* ⇒ *bool* (**infix** ‹$\sim_P$› *75*)
where
$\varphi \sim_P \psi \equiv \forall \mathcal{A}.\ \mathcal{A} \models_P \varphi \longleftrightarrow \mathcal{A} \models_P \psi$

definition *ltl-prop-implies* :: *'a ltln* ⇒ *'a ltln* ⇒ *bool* (**infix** ‹$\longrightarrow_P$› *75*)
where
$\varphi \longrightarrow_P \psi \equiv \forall \mathcal{A}.\ \mathcal{A} \models_P \varphi \longrightarrow \mathcal{A} \models_P \psi$

lemma *ltl-prop-implies-equiv*:
$\varphi \sim_P \psi \longleftrightarrow (\varphi \longrightarrow_P \psi \wedge \psi \longrightarrow_P \varphi)$
unfolding *ltl-prop-equiv-def ltl-prop-implies-def* **by** *meson*

lemma *ltl-prop-equiv-equivp*:
equivp $(\sim_P)$
by (*simp add*: *ltl-prop-equiv-def equivpI reflp-def symp-def transp-def*)

lemma *ltl-prop-equiv-trans*[*trans*]:
$\varphi \sim_P \psi \Longrightarrow \psi \sim_P \chi \Longrightarrow \varphi \sim_P \chi$
by (*simp add*: *ltl-prop-equiv-def*)

lemma *ltl-prop-equiv-true*:
$\varphi \sim_P true_n \longleftrightarrow \{\} \models_P \varphi$
using *bot.extremum ltl-prop-entailment.simps*(*1*) *ltl-prop-equiv-def* **by** *blast*

lemma *ltl-prop-equiv-false*:
$\varphi \sim_P false_n \longleftrightarrow \neg\ UNIV \models_P \varphi$
by (*meson ltl-prop-entailment.simps*(*2*) *ltl-prop-entailment-monotonI ltl-prop-equiv-def top-greatest*)

lemma *ltl-prop-equiv-true-implies-true*:
$x \sim_P true_n \Longrightarrow x \longrightarrow_P y \Longrightarrow y \sim_P true_n$
by (*simp add*: *ltl-prop-equiv-def ltl-prop-implies-def*)

lemma *ltl-prop-equiv-false-implied-by-false*:
$y \sim_P false_n \Longrightarrow x \longrightarrow_P y \Longrightarrow x \sim_P false_n$
by (*simp add*: *ltl-prop-equiv-def ltl-prop-implies-def*)

lemma *ltl-prop-implication-implies-ltl-implication*:
$w \models_n \varphi \Longrightarrow \varphi \longrightarrow_P \psi \Longrightarrow w \models_n \psi$
using *ltl-models-equiv-prop-entailment ltl-prop-implies-def* **by** *blast*

lemma *ltl-prop-equiv-implies-ltl-lang-equiv*:
$\varphi \sim_P \psi \Longrightarrow \varphi \sim_L \psi$
using *ltl-lang-equiv-def ltl-prop-implication-implies-ltl-implication ltl-prop-implies-equiv* **by** *blast*

lemma *ltl-prop-equiv-lt-ltl-lang-equiv*[*simp*]:
$(\sim_P) \leq (\sim_L)$
using *ltl-prop-equiv-implies-ltl-lang-equiv* **by** *blast*

3.3 Constants Equivalence

datatype *tvl* = *Yes* | *No* | *Maybe*

definition *eval-and* :: *tvl* ⇒ *tvl* ⇒ *tvl*
where
eval-and φ ψ =
(*case* (φ, ψ) *of*
(*Yes*, *Yes*) ⇒ *Yes*
| (*No*, -) ⇒ *No*
| (-, *No*) ⇒ *No*
| - ⇒ *Maybe*)

definition *eval-or* :: *tvl* ⇒ *tvl* ⇒ *tvl*
where
eval-or φ ψ =
(*case* (φ, ψ) *of*
(*No*, *No*) ⇒ *No*
| (*Yes*, -) ⇒ *Yes*
| (-, *Yes*) ⇒ *Yes*
| - ⇒ *Maybe*)

fun *eval* :: *'a ltln* ⇒ *tvl*
where
eval true$_n$ = *Yes*
| *eval false*$_n$ = *No*
| *eval* (φ *and*$_n$ ψ) = *eval-and* (*eval* φ) (*eval* ψ)
| *eval* (φ *or*$_n$ ψ) = *eval-or* (*eval* φ) (*eval* ψ)
| *eval* φ = *Maybe*

lemma *eval-and-const*[*simp*]:
eval-and φ ψ = *No* ⟷ φ = *No* ∨ ψ = *No*
eval-and φ ψ = *Yes* ⟷ φ = *Yes* ∧ ψ = *Yes*
unfolding *eval-and-def*
by (*cases* φ; *cases* ψ, *auto*)+

lemma *eval-or-const*[*simp*]:
eval-or φ ψ = *Yes* ⟷ φ = *Yes* ∨ ψ = *Yes*
eval-or φ ψ = *No* ⟷ φ = *No* ∧ ψ = *No*
unfolding *eval-or-def*
by (*cases* φ; *cases* ψ, *auto*)+

lemma *eval-prop-entailment*:
eval φ = *Yes* ⟷ {} ⊨$_P$ φ

eval φ = *No* $\longleftrightarrow$ $\neg$ *UNIV* $\models_P$ φ
by (*induction* φ) *auto*

definition *ltl-const-equiv* :: $'a$ *ltln* $\Rightarrow$ $'a$ *ltln* $\Rightarrow$ *bool* (**infix** $\langle\sim_C\rangle$ *75*)
where
$\varphi \sim_C \psi \equiv \varphi = \psi \vee (\mathit{eval}\ \varphi = \mathit{eval}\ \psi \wedge \mathit{eval}\ \psi \neq \mathit{Maybe})$

lemma *ltl-const-equiv-equivp*:
equivp $(\sim_C)$
unfolding *ltl-const-equiv-def*
by (*intro equivpI reflpI sympI transpI*) *auto*

lemma *ltl-const-equiv-const*:
$\varphi \sim_C \mathit{true}_n \longleftrightarrow \mathit{eval}\ \varphi = \mathit{Yes}$
$\varphi \sim_C \mathit{false}_n \longleftrightarrow \mathit{eval}\ \varphi = \mathit{No}$
unfolding *ltl-const-equiv-def* **by** *force+*

lemma *ltl-const-equiv-and-const*[*simp*]:
$\varphi_1\ \mathit{and}_n\ \varphi_2 \sim_C \mathit{true}_n \longleftrightarrow \varphi_1 \sim_C \mathit{true}_n \wedge \varphi_2 \sim_C \mathit{true}_n$
$\varphi_1\ \mathit{and}_n\ \varphi_2 \sim_C \mathit{false}_n \longleftrightarrow \varphi_1 \sim_C \mathit{false}_n \vee \varphi_2 \sim_C \mathit{false}_n$
unfolding *ltl-const-equiv-const* **by** *force+*

lemma *ltl-const-equiv-or-const*[*simp*]:
$\varphi_1\ \mathit{or}_n\ \varphi_2 \sim_C \mathit{true}_n \longleftrightarrow \varphi_1 \sim_C \mathit{true}_n \vee \varphi_2 \sim_C \mathit{true}_n$
$\varphi_1\ \mathit{or}_n\ \varphi_2 \sim_C \mathit{false}_n \longleftrightarrow \varphi_1 \sim_C \mathit{false}_n \wedge \varphi_2 \sim_C \mathit{false}_n$
unfolding *ltl-const-equiv-const* **by** *force+*

lemma *ltl-const-equiv-other*[*simp*]:
$\varphi \sim_C \mathit{prop}_n(a) \longleftrightarrow \varphi = \mathit{prop}_n(a)$
$\varphi \sim_C \mathit{nprop}_n(a) \longleftrightarrow \varphi = \mathit{nprop}_n(a)$
$\varphi \sim_C X_n\ \psi \longleftrightarrow \varphi = X_n\ \psi$
$\varphi \sim_C \psi_1\ U_n\ \psi_2 \longleftrightarrow \varphi = \psi_1\ U_n\ \psi_2$
$\varphi \sim_C \psi_1\ R_n\ \psi_2 \longleftrightarrow \varphi = \psi_1\ R_n\ \psi_2$
$\varphi \sim_C \psi_1\ W_n\ \psi_2 \longleftrightarrow \varphi = \psi_1\ W_n\ \psi_2$
$\varphi \sim_C \psi_1\ M_n\ \psi_2 \longleftrightarrow \varphi = \psi_1\ M_n\ \psi_2$
using *ltl-const-equiv-def* **by** *fastforce+*

lemma *ltl-const-equiv-no-const-singleton*:
eval ψ = *Maybe* $\Longrightarrow$ $\varphi \sim_C \psi \Longrightarrow \varphi = \psi$
unfolding *ltl-const-equiv-def* **by** *fastforce*

lemma *ltl-const-equiv-implies-prop-equiv*:
$\varphi \sim_C \mathit{true}_n \longleftrightarrow \varphi \sim_P \mathit{true}_n$
$\varphi \sim_C \mathit{false}_n \longleftrightarrow \varphi \sim_P \mathit{false}_n$

unfolding *ltl-const-equiv-const eval-prop-entailment ltl-prop-equiv-def*
by *auto*

lemma *ltl-const-equiv-no-const-prop-equiv*:
$eval\ \psi = Maybe \Longrightarrow \varphi \sim_C \psi \Longrightarrow \varphi \sim_P \psi$
using *ltl-const-equiv-no-const-singleton equivp-reflp*[*OF ltl-prop-equiv-equivp*]
by *blast*

lemma *ltl-const-equiv-implies-ltl-prop-equiv*:
$\varphi \sim_C \psi \Longrightarrow \varphi \sim_P \psi$
proof (*induction* ψ)
case (*And-ltln* $\psi 1$ $\psi 2$)

show *?case*
proof (*cases eval* ($\psi 1$ and_n $\psi 2$))
case *Yes*

then have $\varphi \sim_C true_n$
by (*meson And-ltln.prems equivp-transp ltl-const-equiv-const*(*1*) *ltl-const-equiv-equivp*)
then show *?thesis*
by (*metis* (*full-types*) *Yes ltl-const-equiv-const*(*1*) *ltl-const-equiv-implies-prop-equiv*(*1*) *ltl-prop-equiv-trans ltl-prop-implies-equiv*)
next
case *No*

then have $\varphi \sim_C false_n$
by (*meson And-ltln.prems equivp-transp ltl-const-equiv-const*(*2*) *ltl-const-equiv-equivp*)
then show *?thesis*
by (*metis* (*full-types*) *No ltl-const-equiv-const*(*2*) *ltl-const-equiv-implies-prop-equiv*(*2*) *ltl-prop-equiv-trans ltl-prop-implies-equiv*)
next
case *Maybe*

then show *?thesis*
using *And-ltln.prems ltl-const-equiv-no-const-prop-equiv* **by** *force*
qed
next
case (*Or-ltln* $\psi 1$ $\psi 2$)

then show *?case*
proof (*cases eval* ($\psi 1$ or_n $\psi 2$))
case *Yes*

then have $\varphi \sim_C true_n$

by (*meson Or-ltln.prems equivp-transp ltl-const-equiv-const*(*1*) *ltl-const-equiv-equivp*)
then show *?thesis*
by (*metis* (*full-types*) *Yes ltl-const-equiv-const*(*1*) *ltl-const-equiv-implies-prop-equiv*(*1*) *ltl-prop-equiv-trans ltl-prop-implies-equiv*)
next
case *No*

then have $\varphi \sim_C \mathit{false}_n$
by (*meson Or-ltln.prems equivp-transp ltl-const-equiv-const*(*2*) *ltl-const-equiv-equivp*)
then show *?thesis*
by (*metis* (*full-types*) *No ltl-const-equiv-const*(*2*) *ltl-const-equiv-implies-prop-equiv*(*2*) *ltl-prop-equiv-trans ltl-prop-implies-equiv*)
next
case *Maybe*

then show *?thesis*
using *Or-ltln.prems ltl-const-equiv-no-const-prop-equiv* **by** *force*
qed
qed (*simp-all add*: *ltl-const-equiv-implies-prop-equiv equivp-reflp*[*OF ltl-prop-equiv-equivp*])

lemma *ltl-const-equiv-lt-ltl-prop-equiv*[*simp*]:
$(\sim_C) \leq (\sim_P)$
using *ltl-const-equiv-implies-ltl-prop-equiv* **by** *blast*

3.4 Quotient types

quotient-type $'a\ \mathit{ltln}_L = 'a\ \mathit{ltln}\ /\ (\sim_L)$
by (*rule ltl-lang-equiv-equivp*)

instantiation $\mathit{ltln}_L :: (\mathit{type})\ \mathit{equal}$
begin

lift-definition $\mathit{ltln}_L\text{-}\mathit{eq\text{-}test} :: 'a\ \mathit{ltln}_L \Rightarrow 'a\ \mathit{ltln}_L \Rightarrow \mathit{bool}$ **is** $\lambda x\ y.\ x \sim_L y$
by (*metis* ltln_L*.abs-eq-iff*)

definition
eq_L: *equal-class.equal* $\equiv$ ltln_L*-eq-test*

instance
by (*standard*; *simp add*: eq_L ltln_L*-eq-test.rep-eq*, *metis Quotient-*ltln_L *Quotient-rel-rep*)

end

quotient-type *'a ltln$_P$ = 'a ltln / ($\sim_P$)*
by (*rule ltl-prop-equiv-equivp*)

instantiation *ltln$_P$* :: (*type*) *equal*
begin

lift-definition *ltln$_P$-eq-test* :: *'a ltln$_P$ ⇒ 'a ltln$_P$ ⇒ bool* **is** *λx y. x ∼$_P$ y*
by (*metis ltln$_P$.abs-eq-iff*)

definition
eq$_P$: *equal-class.equal ≡ ltln$_P$-eq-test*

instance
by (*standard*; *simp add*: *eq$_P$ ltln$_P$-eq-test.rep-eq*, *metis Quotient-ltln$_P$ Quotient-rel-rep*)

end

quotient-type *'a ltln$_C$ = 'a ltln / ($\sim_C$)*
by (*rule ltl-const-equiv-equivp*)

instantiation *ltln$_C$* :: (*type*) *equal*
begin

lift-definition *ltln$_C$-eq-test* :: *'a ltln$_C$ ⇒ 'a ltln$_C$ ⇒ bool* **is** *λx y. x ∼$_C$ y*
by (*metis ltln$_C$.abs-eq-iff*)

definition
eq$_C$: *equal-class.equal ≡ ltln$_C$-eq-test*

instance
by (*standard*; *simp add*: *eq$_C$ ltln$_C$-eq-test.rep-eq*, *metis Quotient-ltln$_C$ Quotient-rel-rep*)

end

3.5 Cardinality of propositional quotient sets

definition *sat-models* :: *'a ltln$_P$ ⇒ 'a ltln set set*
where
sat-models φ = {A. A ⊨$_P$ rep-ltln$_P$ φ}

lemma *Rep-Abs-prop-entailment*[*simp*]:
  $\mathcal{A} \models_P$ *rep-ltln*$_P$ (*abs-ltln*$_P$ φ) = $\mathcal{A} \models_P \varphi$
  by (*metis Quotient3-ltln*$_P$ *Quotient3-rep-abs ltl-prop-equiv-def*)

lemma *sat-models-Abs*:
  $\mathcal{A} \in$ *sat-models* (*abs-ltln*$_P$ φ) = $\mathcal{A} \models_P \varphi$
  by (*simp add*: *sat-models-def*)

lemma *sat-models-inj*:
  inj sat-models
proof (*rule injI*)
  fix φ ψ :: *'a ltln*$_P$
  assume *sat-models* φ = *sat-models* ψ

  then have *rep-ltln*$_P$ φ $\sim_P$ *rep-ltln*$_P$ ψ
    unfolding *sat-models-def ltl-prop-equiv-def* **by** *force*

  then show $\varphi = \psi$
    by (*meson Quotient3-ltln*$_P$ *Quotient3-rel-rep*)
qed

fun *prop-atoms* :: *'a ltln* ⇒ *'a ltln set*
where
  prop-atoms true$_n$ = {}
| *prop-atoms false*$_n$ = {}
| *prop-atoms* (φ *and*$_n$ ψ) = *prop-atoms* φ ∪ *prop-atoms* ψ
| *prop-atoms* (φ *or*$_n$ ψ) = *prop-atoms* φ ∪ *prop-atoms* ψ
| *prop-atoms* φ = {φ}

fun *nested-prop-atoms* :: *'a ltln* ⇒ *'a ltln set*
where
  nested-prop-atoms true$_n$ = {}
| *nested-prop-atoms false*$_n$ = {}
| *nested-prop-atoms* (φ *and*$_n$ ψ) = *nested-prop-atoms* φ ∪ *nested-prop-atoms* ψ
| *nested-prop-atoms* (φ *or*$_n$ ψ) = *nested-prop-atoms* φ ∪ *nested-prop-atoms* ψ
| *nested-prop-atoms* (X_n φ) = {X_n φ} ∪ *nested-prop-atoms* φ
| *nested-prop-atoms* (φ U_n ψ) = {φ U_n ψ} ∪ *nested-prop-atoms* φ ∪ *nested-prop-atoms* ψ
| *nested-prop-atoms* (φ R_n ψ) = {φ R_n ψ} ∪ *nested-prop-atoms* φ ∪ *nested-prop-atoms* ψ
| *nested-prop-atoms* (φ W_n ψ) = {φ W_n ψ} ∪ *nested-prop-atoms* φ ∪

nested-prop-atoms ψ
| *nested-prop-atoms* $(\varphi\ M_n\ \psi) = \{\varphi\ M_n\ \psi\} \cup$ *nested-prop-atoms* $\varphi \cup$ *nested-prop-atoms* ψ
| *nested-prop-atoms* $\varphi = \{\varphi\}$

lemma *prop-atoms-nested-prop-atoms*:
prop-atoms $\varphi \subseteq$ *nested-prop-atoms* φ
by (*induction* φ) *auto*

lemma *prop-atoms-subfrmlsn*:
prop-atoms $\varphi \subseteq$ *subfrmlsn* φ
by (*induction* φ) *auto*

lemma *nested-prop-atoms-subfrmlsn*:
nested-prop-atoms $\varphi \subseteq$ *subfrmlsn* φ
by (*induction* φ) *auto*

lemma *prop-atoms-notin*[*simp*]:
$true_n \notin$ *prop-atoms* φ
$false_n \notin$ *prop-atoms* φ
$\varphi_1\ and_n\ \varphi_2 \notin$ *prop-atoms* φ
$\varphi_1\ or_n\ \varphi_2 \notin$ *prop-atoms* φ
by (*induction* φ) *auto*

lemma *nested-prop-atoms-notin*[*simp*]:
$true_n \notin$ *nested-prop-atoms* φ
$false_n \notin$ *nested-prop-atoms* φ
$\varphi_1\ and_n\ \varphi_2 \notin$ *nested-prop-atoms* φ
$\varphi_1\ or_n\ \varphi_2 \notin$ *nested-prop-atoms* φ
by (*induction* φ) *auto*

lemma *prop-atoms-finite*:
finite (*prop-atoms* φ)
by (*induction* φ) *auto*

lemma *nested-prop-atoms-finite*:
finite (*nested-prop-atoms* φ)
by (*induction* φ) *auto*

lemma *prop-atoms-entailment-iff*:
$\varphi \in$ *prop-atoms* $\psi \Longrightarrow \mathcal{A} \models_P \varphi \longleftrightarrow \varphi \in \mathcal{A}$
by (*induction* φ) *auto*

lemma *prop-atoms-entailment-inter*:

prop-atoms $\varphi \subseteq P \Longrightarrow (\mathcal{A} \cap P) \models_P \varphi = \mathcal{A} \models_P \varphi$
by (*induction* φ) *auto*

lemma *nested-prop-atoms-entailment-inter*:
nested-prop-atoms $\varphi \subseteq P \Longrightarrow (\mathcal{A} \cap P) \models_P \varphi = \mathcal{A} \models_P \varphi$
by (*induction* φ) *auto*

lemma *sat-models-inter-inj-helper*:
assumes
prop-atoms $\varphi \subseteq P$
and
prop-atoms $\psi \subseteq P$
and
sat-models (*abs-ltln*$_P$ φ) $\cap$ *Pow* P = *sat-models* (*abs-ltln*$_P$ ψ) $\cap$ *Pow* P
shows
$\varphi \sim_P \psi$
proof −
from *assms* **have** $\forall \mathcal{A}.\ (\mathcal{A} \cap P) \models_P \varphi \longleftrightarrow (\mathcal{A} \cap P) \models_P \psi$
by (*auto simp*: *sat-models-Abs*)

with *assms* **show** $\varphi \sim_P \psi$
by (*simp add*: *prop-atoms-entailment-inter ltl-prop-equiv-def*)
qed

lemma *sat-models-inter-inj*:
inj-on ($\lambda\varphi$. *sat-models* $\varphi \cap$ *Pow* P) {*abs-ltln*$_P$ φ |φ. *prop-atoms* $\varphi \subseteq P$}
by (*auto simp*: *inj-on-def sat-models-inter-inj-helper ltln*$_P$*.abs-eq-iff*)

lemma *sat-models-pow-pow*:
{*sat-models* (*abs-ltln*$_P$ φ) $\cap$ *Pow* P | φ. *prop-atoms* $\varphi \subseteq P$} $\subseteq$ *Pow* (*Pow* P)
by (*auto simp*: *sat-models-def*)

lemma *sat-models-finite*:
finite $P \Longrightarrow$ *finite* {*sat-models* (*abs-ltln*$_P$ φ) $\cap$ *Pow* P | φ. *prop-atoms* φ $\subseteq P$}
using *sat-models-pow-pow finite-subset* **by** *fastforce*

lemma *sat-models-card*:
finite $P \Longrightarrow$ *card* ({*sat-models* (*abs-ltln*$_P$ φ) $\cap$ *Pow* P | φ. *prop-atoms* φ $\subseteq P$}) $\leq$ 2 ^ 2 ^ *card* P
by (*metis* (*mono-tags*, *lifting*) *sat-models-pow-pow Pow-def card-Pow card-mono finite-Collect-subsets*)

lemma *image-filter*:
f ‘ {g a | a. P a} = {f (g a) | a. P a}
by *blast*

lemma *prop-equiv-finite*:
finite P ⟹ finite {abs-ltln$_P$ ψ | ψ. prop-atoms ψ ⊆ P}
by (*auto simp*: *image-filter sat-models-finite finite-imageD*[*OF - sat-models-inter-inj*])

lemma *prop-equiv-card*:
finite P ⟹ card {abs-ltln$_P$ ψ | ψ. prop-atoms ψ ⊆ P} ≤ 2 ^ 2 ^ card P
by (*auto simp*: *image-filter sat-models-card card-image*[*OF sat-models-inter-inj*, *symmetric*])

lemma *prop-equiv-subset*:
{abs-ltln$_P$ ψ |ψ. nested-prop-atoms ψ ⊆ P} ⊆ {abs-ltln$_P$ ψ |ψ. prop-atoms ψ ⊆ P}
using *prop-atoms-nested-prop-atoms* **by** *blast*

lemma *prop-equiv-finite′*:
finite P ⟹ finite {abs-ltln$_P$ ψ | ψ. nested-prop-atoms ψ ⊆ P}
using *prop-equiv-finite prop-equiv-subset finite-subset* **by** *fast*

lemma *prop-equiv-card′*:
finite P ⟹ card {abs-ltln$_P$ ψ | ψ. nested-prop-atoms ψ ⊆ P} ≤ 2 ^ 2 ^ card P
by (*metis* (*mono-tags*, *lifting*) *prop-equiv-card prop-equiv-subset prop-equiv-finite card-mono le-trans*)

3.6 Substitution

fun *subst* :: *'a ltln ⇒ ('a ltln ⇀ 'a ltln) ⇒ 'a ltln*
where
subst true$_n$ m = true$_n$
| *subst false$_n$ m = false$_n$*
| *subst (φ and$_n$ ψ) m = subst φ m and$_n$ subst ψ m*
| *subst (φ or$_n$ ψ) m = subst φ m or$_n$ subst ψ m*
| *subst φ m = (case m φ of Some ψ ⇒ ψ | None ⇒ φ)*

Based on Uwe Schoening's Translation Lemma (Logic for CS, p. 54)

lemma *ltl-prop-equiv-subst-S*:
S ⊨$_P$ subst φ m = ((S − dom m) ∪ {χ | χ χ′. χ ∈ dom m ∧ m χ = Some χ′ ∧ S ⊨$_P$ χ′}) ⊨$_P$ φ

by (*induction* φ) (*auto split*: *option.split*)

lemma *subst-respects-ltl-prop-entailment*:
$\varphi \longrightarrow_P \psi \Longrightarrow$ *subst* φ $m \longrightarrow_P$ *subst* ψ m
$\varphi \sim_P \psi \Longrightarrow$ *subst* φ $m \sim_P$ *subst* ψ m
unfolding *ltl-prop-equiv-def ltl-prop-implies-def ltl-prop-equiv-subst-S* **by** *blast*+

lemma *eval-subst*:
eval φ = *Yes* $\Longrightarrow$ *eval* (*subst* φ m) = *Yes*
eval φ = *No* $\Longrightarrow$ *eval* (*subst* φ m) = *No*
by (*meson empty-subsetI eval-prop-entailment ltl-prop-entailment-monotonI ltl-prop-equiv-subst-S subset-UNIV*)+

lemma *subst-respects-ltl-const-entailment*:
$\varphi \sim_C \psi \Longrightarrow$ *subst* φ $m \sim_C$ *subst* ψ m
unfolding *ltl-const-equiv-def*
by (*cases eval* ψ) (*metis eval-subst*(*1*), *metis eval-subst*(*2*), *blast*)

3.7 Order of Equivalence Relations

locale *ltl-equivalence* =
fixes
eq :: *'a ltln* $\Rightarrow$ *'a ltln* $\Rightarrow$ *bool* (**infix** $\langle\sim\rangle$ *75*)
assumes
eq-equivp: *equivp* ($\sim$)
and
ge-const-equiv: ($\sim_C$) $\leq$ ($\sim$)
and
le-lang-equiv: ($\sim$) $\leq$ ($\sim_L$)
begin

lemma *eq-implies-ltl-equiv*:
$\varphi \sim \psi \Longrightarrow w \models_n \varphi = w \models_n \psi$
using *le-lang-equiv ltl-lang-equiv-def* **by** *blast*

lemma *const-implies-eq*:
$\varphi \sim_C \psi \Longrightarrow \varphi \sim \psi$
using *ge-const-equiv* **by** *blast*

lemma *eq-implies-lang*:
$\varphi \sim \psi \Longrightarrow \varphi \sim_L \psi$
using *le-lang-equiv* **by** *blast*

lemma *eq-refl*[*simp*]:
$\varphi \sim \varphi$
by (*meson eq-equivp equivp-reflp*)

lemma *eq-sym*[*sym*]:
$\varphi \sim \psi \Longrightarrow \psi \sim \varphi$
by (*meson eq-equivp equivp-symp*)

lemma *eq-trans*[*trans*]:
$\varphi \sim \psi \Longrightarrow \psi \sim \chi \Longrightarrow \varphi \sim \chi$
by (*meson eq-equivp equivp-transp*)

end

interpretation *ltl-lang-equivalence*: *ltl-equivalence* ($\sim_L$)
using *ltl-lang-equiv-equivp ltl-const-equiv-lt-ltl-prop-equiv ltl-prop-equiv-lt-ltl-lang-equiv*
by *unfold-locales blast+*

interpretation *ltl-prop-equivalence*: *ltl-equivalence* ($\sim_P$)
using *ltl-prop-equiv-equivp ltl-const-equiv-lt-ltl-prop-equiv ltl-prop-equiv-lt-ltl-lang-equiv*
by *unfold-locales blast+*

interpretation *ltl-const-equivalence*: *ltl-equivalence* ($\sim_C$)
using *ltl-const-equiv-equivp ltl-const-equiv-lt-ltl-prop-equiv ltl-prop-equiv-lt-ltl-lang-equiv*
by *unfold-locales blast+*

end

4 Disjunctive Normal Form of LTL formulas

theory *Disjunctive-Normal-Form*
imports
LTL Equivalence-Relations HOL−Library.FSet
begin

We use the propositional representation of LTL formulas to define the minimal disjunctive normal form of our formulas. For this purpose we define the minimal product $\otimes_m$ and union $\cup_m$. In the end we show that for a set $\mathcal{A}$ of literals, $\mathcal{A} \models_P \varphi$ if, and only if, there exists a subset of $\mathcal{A}$ in the minimal DNF of φ.

4.1 Definition of Minimum Sets

definition (**in** *ord*) *min-set* :: *'a set* ⇒ *'a set* **where**
min-set X = {*y* ∈ *X*. ∀ *x* ∈ *X*. *x* ≤ *y* ⟶ *x* = *y*}

lemma *min-set-iff*:
x ∈ *min-set X* ⟷ *x* ∈ *X* ∧ (∀ *y* ∈ *X*. *y* ≤ *x* ⟶ *y* = *x*)
unfolding *min-set-def* **by** *blast*

lemma *min-set-subset*:
min-set X ⊆ *X*
by (*auto simp*: *min-set-def*)

lemma *min-set-idem*[*simp*]:
min-set (*min-set X*) = *min-set X*
by (*auto simp*: *min-set-def*)

lemma *min-set-empty*[*simp*]:
min-set {} = {}
using *min-set-subset* **by** *blast*

lemma *min-set-singleton*[*simp*]:
min-set {*x*} = {*x*}
by (*auto simp*: *min-set-def*)

lemma *min-set-finite*:
finite X ⟹ *finite* (*min-set X*)
by (*simp add*: *min-set-def*)

lemma *min-set-obtains-helper*:
A ∈ *B* ⟹ ∃ *C*. *C* |⊆| *A* ∧ *C* ∈ *min-set B*
proof (*induction fcard A arbitrary*: *A rule*: *less-induct*)
case *less*

then have (∀ *A'*. *A'* ∉ *B* ∨ ¬ *A'* |⊆| *A* ∨ *A'* = *A*) ∨ (∃ *A'*. *A'* |⊆| *A* ∧ *A'* ∈ *min-set B*)
by (*metis* (*no-types*) *dual-order.trans order.not-eq-order-implies-strict pfsubset-fcard-mono*)

then show *?case*
using *less.prems min-set-def* **by** *auto*
qed

lemma *min-set-obtains*:

assumes $A \in B$
obtains C **where** $C \mid\subseteq\mid A$ **and** $C \in$ *min-set* B
using *min-set-obtains-helper assms* **by** *metis*

4.2 Minimal operators on sets

definition *product* :: *'a fset set* ⇒ *'a fset set* ⇒ *'a fset set* (**infixr** ‹⊗› *65*)
where $A \otimes B = \{a \mid\cup\mid b \mid a\ b.\ a \in A \wedge b \in B\}$

definition *min-product* :: *'a fset set* ⇒ *'a fset set* ⇒ *'a fset set* (**infixr** ‹$\otimes_m$› *65*)
where $A \otimes_m B =$ *min-set* $(A \otimes B)$

definition *min-union* :: *'a fset set* ⇒ *'a fset set* ⇒ *'a fset set* (**infixr** ‹$\cup_m$› *65*)
where $A \cup_m B =$ *min-set* $(A \cup B)$

definition *product-set* :: *'a fset set set* ⇒ *'a fset set* (‹$\bigotimes$›)
where $\bigotimes X =$ *Finite-Set.fold product* $\{\{||\}\}$ X

definition *min-product-set* :: *'a fset set set* ⇒ *'a fset set* (‹$\bigotimes_m$›)
where $\bigotimes_m X =$ *Finite-Set.fold min-product* $\{\{||\}\}$ X

lemma *min-product-idem*[*simp*]:
$A \otimes_m A =$ *min-set* A
by (*auto simp*: *min-product-def product-def min-set-def*) *fastforce*

lemma *min-union-idem*[*simp*]:
$A \cup_m A =$ *min-set* A
by (*simp add*: *min-union-def*)

lemma *product-empty*[*simp*]:
$A \otimes \{\} = \{\}$
$\{\} \otimes A = \{\}$
by (*simp-all add*: *product-def*)

lemma *min-product-empty*[*simp*]:
$A \otimes_m \{\} = \{\}$
$\{\} \otimes_m A = \{\}$
by (*simp-all add*: *min-product-def*)

lemma *min-union-empty*[*simp*]:

$A \cup_m \{\} = \textit{min-set}\ A$
$\{\} \cup_m A = \textit{min-set}\ A$
by (*simp-all add*: *min-union-def*)

lemma *product-empty-singleton*[*simp*]:
$A \otimes \{\{||\}\} = A$
$\{\{||\}\} \otimes A = A$
by (*simp-all add*: *product-def*)

lemma *min-product-empty-singleton*[*simp*]:
$A \otimes_m \{\{||\}\} = \textit{min-set}\ A$
$\{\{||\}\} \otimes_m A = \textit{min-set}\ A$
by (*simp-all add*: *min-product-def*)

lemma *product-singleton-singleton*:
$A \otimes \{\{|x|\}\} = \textit{finsert}\ x\ `\ A$
$\{\{|x|\}\} \otimes A = \textit{finsert}\ x\ `\ A$
unfolding *product-def* **by** *blast*+

lemma *product-mono*:
$A \subseteq B \Longrightarrow A \otimes C \subseteq B \otimes C$
$B \subseteq C \Longrightarrow A \otimes B \subseteq A \otimes C$
unfolding *product-def* **by** *auto*

lemma *product-finite*:
$\textit{finite}\ A \Longrightarrow \textit{finite}\ B \Longrightarrow \textit{finite}\ (A \otimes B)$
by (*simp add*: *product-def finite-image-set2*)

lemma *min-product-finite*:
$\textit{finite}\ A \Longrightarrow \textit{finite}\ B \Longrightarrow \textit{finite}\ (A \otimes_m B)$
by (*metis min-product-def product-finite min-set-finite*)

lemma *min-union-finite*:
$\textit{finite}\ A \Longrightarrow \textit{finite}\ B \Longrightarrow \textit{finite}\ (A \cup_m B)$
by (*simp add*: *min-union-def min-set-finite*)

lemma *product-set-infinite*[*simp*]:
$\textit{infinite}\ X \Longrightarrow \bigotimes\ X = \{\{||\}\}$
by (*simp add*: *product-set-def*)

lemma *min-product-set-infinite*[*simp*]:

infinite $X \Longrightarrow \bigotimes_m X = \{\{||\}\}$
by (*simp add*: *min-product-set-def*)

lemma *product-comm*:
$A \otimes B = B \otimes A$
unfolding *product-def* **by** *blast*

lemma *min-product-comm*:
$A \otimes_m B = B \otimes_m A$
unfolding *min-product-def*
by (*simp add*: *product-comm*)

lemma *min-union-comm*:
$A \cup_m B = B \cup_m A$
unfolding *min-union-def*
by (*simp add*: *sup.commute*)

lemma *product-iff*:
$x \in A \otimes B \longleftrightarrow (\exists\, a \in A.\ \exists\, b \in B.\ x = a \mid\cup\mid b)$
unfolding *product-def* **by** *blast*

lemma *min-product-iff*:
$x \in A \otimes_m B \longleftrightarrow (\exists\, a \in A.\ \exists\, b \in B.\ x = a \mid\cup\mid b) \wedge (\forall\, a \in A.\ \forall\, b \in B.\ a \mid\cup\mid b \mid\subseteq\mid x \longrightarrow a \mid\cup\mid b = x)$
unfolding *min-product-def min-set-iff product-iff product-def* **by** *blast*

lemma *min-union-iff*:
$x \in A \cup_m B \longleftrightarrow x \in A \cup B \wedge (\forall\, a \in A.\ a \mid\subseteq\mid x \longrightarrow a = x) \wedge (\forall\, b \in B.\ b \mid\subseteq\mid x \longrightarrow b = x)$
unfolding *min-union-def min-set-iff* **by** *blast*

lemma *min-set-min-product-helper*:
$x \in (\textit{min-set}\ A) \otimes_m B \longleftrightarrow x \in A \otimes_m B$
proof
fix x
assume $x \in (\textit{min-set}\ A) \otimes_m B$

then obtain $a\ b$ **where** $a \in \textit{min-set}\ A$ **and** $b \in B$ **and** $x = a \mid\cup\mid b$ **and** *1*: $\forall\, a \in \textit{min-set}\ A.\ \forall\, b \in B.\ a \mid\cup\mid b \mid\subseteq\mid x \longrightarrow a \mid\cup\mid b = x$
unfolding *min-product-iff* **by** *blast*

moreover

{
fix a' b'
assume $a' \in A$ **and** $b' \in B$ **and** $a' \mid\cup\mid b' \mid\subseteq\mid x$

then obtain a'' **where** $a'' \mid\subseteq\mid a'$ **and** $a'' \in$ *min-set* A
using *min-set-obtains* **by** *metis*

then have $a'' \mid\cup\mid b' = x$
by (*metis* (*full-types*) *1* ‹$b' \in B$› ‹$a' \mid\cup\mid b' \mid\subseteq\mid x$› *dual-order.trans le-sup-iff*)

then have $a' \mid\cup\mid b' = x$
using ‹$a' \mid\cup\mid b' \mid\subseteq\mid x$› ‹$a'' \mid\subseteq\mid a'$› **by** *blast*
}

ultimately show $x \in A \otimes_m B$
by (*metis min-product-iff min-set-iff*)
next
fix x
assume $x \in A \otimes_m B$

then have *1*: $x \in A \otimes B$ **and** $\forall\, y \in A \otimes B.\ y \mid\subseteq\mid x \longrightarrow y = x$
unfolding *min-product-def min-set-iff* **by** *simp+*

then have *2*: $\forall\, y{\in}$*min-set* $A \otimes B.\ y \mid\subseteq\mid x \longrightarrow y = x$
by (*metis product-iff min-set-iff*)

then have $x \in$ *min-set* $A \otimes B$
by (*metis 1 funion-mono min-set-obtains order-refl product-iff*)

then show $x \in$ *min-set* $A \otimes_m B$
by (*simp add*: *2 min-product-def min-set-iff*)
qed

lemma *min-set-min-product*[*simp*]:
(*min-set* A) $\otimes_m B = A \otimes_m B$
$A \otimes_m$ (*min-set* B) $= A \otimes_m B$
using *min-product-comm min-set-min-product-helper* **by** *blast+*

lemma *min-set-min-union*[*simp*]:
(*min-set* A) $\cup_m B = A \cup_m B$
$A \cup_m$ (*min-set* B) $= A \cup_m B$

proof (*unfold min-union-def min-set-def*, *safe*)
show $\bigwedge x\ xa\ xb.\ [\![\forall\, xa{\in}\{y \in A.\ \forall\, x{\in}A.\ x \mid{\subseteq}\mid y \longrightarrow x = y\} \cup B.\ xa \mid{\subseteq}\mid x \longrightarrow xa = x;\ x \in B;\ xa \mid{\subseteq}\mid x;\ xb \mid{\in}\mid x;\ xa \in A]\!] \Longrightarrow xb \mid{\in}\mid xa$
by (*metis* (*mono-tags*) *UnCI dual-order.trans fequalityI min-set-def min-set-obtains*)
next
show $\bigwedge x\ xa\ xb.\ [\![\forall\, xa{\in}A \cup \{y \in B.\ \forall\, x{\in}B.\ x \mid{\subseteq}\mid y \longrightarrow x = y\}.\ xa \mid{\subseteq}\mid x \longrightarrow xa = x;\ x \in A;\ xa \mid{\subseteq}\mid x;\ xb \mid{\in}\mid x;\ xa \in B]\!] \Longrightarrow xb \mid{\in}\mid xa$
by (*metis* (*mono-tags*) *UnCI dual-order.trans fequalityI min-set-def min-set-obtains*)
qed *blast+*

lemma *product-assoc*[*simp*]:
$(A \otimes B) \otimes C = A \otimes (B \otimes C)$
proof (*unfold product-def*, *safe*)
fix *a b c*
assume $a \in A$ **and** $c \in C$ **and** $b \in B$
then have $b \mid{\cup}\mid c \in \{b \mid{\cup}\mid c \mid b\ c.\ b \in B \wedge c \in C\}$
by *blast*
then show $\exists\, a'\ bc.\ a \mid{\cup}\mid b \mid{\cup}\mid c = a' \mid{\cup}\mid bc \wedge a' \in A \wedge bc \in \{b \mid{\cup}\mid c \mid b\ c.\ b \in B \wedge c \in C\}$
using ‹$a \in A$› **by** (*metis* (*no-types*) *inf-sup-aci*(*5*) *sup-left-commute*)
qed (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq sup-assoc*)

lemma *min-product-assoc*[*simp*]:
$(A \otimes_m B) \otimes_m C = A \otimes_m (B \otimes_m C)$
unfolding *min-product-def*[*of A B*] *min-product-def*[*of B C*]
by *simp* (*simp add*: *min-product-def*)

lemma *min-union-assoc*[*simp*]:
$(A \cup_m B) \cup_m C = A \cup_m (B \cup_m C)$
unfolding *min-union-def*[*of A B*] *min-union-def*[*of B C*]
by *simp* (*simp add*: *min-union-def sup-assoc*)

lemma *min-product-comp*:
$a \in A \Longrightarrow b \in B \Longrightarrow \exists\, c.\ c \mid{\subseteq}\mid (a \mid{\cup}\mid b) \wedge c \in A \otimes_m B$
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq min-product-def product-def min-set-obtains*)

lemma *min-union-comp*:
$a \in A \Longrightarrow \exists\, c.\ c \mid{\subseteq}\mid a \wedge c \in A \cup_m B$
by (*metis Un-iff min-set-obtains min-union-def*)

interpretation *product-set-thms*: *Finite-Set.comp-fun-commute product*
proof *unfold-locales*
  have $\bigwedge x\ y\ z.\ x \otimes (y \otimes z) = y \otimes (x \otimes z)$
    by (*simp only*: *product-assoc*[*symmetric*]) (*simp only*: *product-comm*)

  then show $\bigwedge x\ y.\ (\otimes)\ y \circ (\otimes)\ x = (\otimes)\ x \circ (\otimes)\ y$
    by *fastforce*
qed

interpretation *min-product-set-thms*: *Finite-Set.comp-fun-idem min-product*
proof *unfold-locales*
  have $\bigwedge x\ y\ z.\ x \otimes_m (y \otimes_m z) = y \otimes_m (x \otimes_m z)$
   by (*simp only*: *min-product-assoc*[*symmetric*]) (*simp only*: *min-product-comm*)

  then show $\bigwedge x\ y.\ (\otimes_m)\ y \circ (\otimes_m)\ x = (\otimes_m)\ x \circ (\otimes_m)\ y$
    by *fastforce*
next
  have $\bigwedge x\ y.\ x \otimes_m (x \otimes_m y) = x \otimes_m y$
    by (*simp add*: *min-product-assoc*[*symmetric*])

  then show $\bigwedge x.\ (\otimes_m)\ x \circ (\otimes_m)\ x = (\otimes_m)\ x$
    by *fastforce*
qed

interpretation *min-union-set-thms*: *Finite-Set.comp-fun-idem min-union*
proof *unfold-locales*
  have $\bigwedge x\ y\ z.\ x \cup_m (y \cup_m z) = y \cup_m (x \cup_m z)$
   by (*simp only*: *min-union-assoc*[*symmetric*]) (*simp only*: *min-union-comm*)

  then show $\bigwedge x\ y.\ (\cup_m)\ y \circ (\cup_m)\ x = (\cup_m)\ x \circ (\cup_m)\ y$
    by *fastforce*
next
  have $\bigwedge x\ y.\ x \cup_m (x \cup_m y) = x \cup_m y$
    by (*simp add*: *min-union-assoc*[*symmetric*])

  then show $\bigwedge x.\ (\cup_m)\ x \circ (\cup_m)\ x = (\cup_m)\ x$
    by *fastforce*
qed

lemma *product-set-empty*[*simp*]:

$\bigotimes\ \{\} = \{\{||\}\}$
$\bigotimes\ \{\{\}\} = \{\}$
$\bigotimes\ \{\{\{||\}\}\} = \{\{||\}\}$
by (*simp-all add*: *product-set-def*)

lemma *min-product-set-empty*[*simp*]:
$\bigotimes_m\ \{\} = \{\{||\}\}$
$\bigotimes_m\ \{\{\}\} = \{\}$
$\bigotimes_m\ \{\{\{||\}\}\} = \{\{||\}\}$
by (*simp-all add*: *min-product-set-def*)

lemma *product-set-code*[*code*]:
$\bigotimes\ (set\ xs) = fold\ product\ (remdups\ xs)\ \{\{||\}\}$
by (*simp add*: *product-set-def product-set-thms.fold-set-fold-remdups*)

lemma *min-product-set-code*[*code*]:
$\bigotimes_m\ (set\ xs) = fold\ min\text{-}product\ (remdups\ xs)\ \{\{||\}\}$
by (*simp add*: *min-product-set-def min-product-set-thms.fold-set-fold-remdups*)

lemma *product-set-insert*[*simp*]:
$finite\ X \Longrightarrow \bigotimes\ (insert\ x\ X) = x \otimes (\bigotimes\ (X - \{x\}))$
unfolding *product-set-def product-set-thms.fold-insert-remove* **..**

lemma *min-product-set-insert*[*simp*]:
$finite\ X \Longrightarrow \bigotimes_m\ (insert\ x\ X) = x \otimes_m (\bigotimes_m\ X)$
unfolding *min-product-set-def min-product-set-thms.fold-insert-idem* **..**

lemma *min-product-subseteq*:
$x \in A \otimes_m B \Longrightarrow \exists\, a.\ a \mid\subseteq\mid x \wedge a \in A$
by (*metis funion-upper1 min-product-iff*)

lemma *min-product-set-subseteq*:
$finite\ X \Longrightarrow x \in \bigotimes_m\ X \Longrightarrow A \in X \Longrightarrow \exists\, a \in A.\ a \mid\subseteq\mid x$
by (*induction X rule*: *finite-induct*) (*blast*, *metis finite-insert insert-absorb min-product-set-insert min-product-subseteq*)

lemma *min-set-product-set*:
$\bigotimes_m\ A = min\text{-}set\ (\bigotimes\ A)$
by (*cases finite A*, *induction A rule*: *finite-induct*) (*simp-all add*: *min-product-set-def product-set-def*, *metis min-product-def*)

lemma *min-product-min-set*[*simp*]:
$min\text{-}set\ (A \otimes_m B) = A \otimes_m B$

by (*simp add*: *min-product-def*)

lemma *min-union-min-set*[*simp*]:
min-set $(A \cup_m B) = A \cup_m B$
by (*simp add*: *min-union-def*)

lemma *min-product-set-min-set*[*simp*]:
finite $X \Longrightarrow$ *min-set* $(\bigotimes_m X) = \bigotimes_m X$
by (*induction X rule*: *finite-induct*, *auto simp add*: *min-product-set-def min-set-iff*)

lemma *min-set-min-product-set*[*simp*]:
finite $X \Longrightarrow \bigotimes_m$ (*min-set* ‘ X) $= \bigotimes_m X$
by (*induction X rule*: *finite-induct*) *simp-all*

lemma *min-product-set-union*[*simp*]:
finite $X \Longrightarrow$ *finite* $Y \Longrightarrow \bigotimes_m (X \cup Y) = (\bigotimes_m X) \otimes_m (\bigotimes_m Y)$
by (*induction X rule*: *finite-induct*) *simp-all*

lemma *product-set-finite*:
$(\bigwedge x.\ x \in X \Longrightarrow$ *finite* $x) \Longrightarrow$ *finite* $(\bigotimes X)$
by (*cases finite X*, *rotate-tac*, *induction X rule*: *finite-induct*) (*simp-all add*: *product-set-def*, *insert product-finite*, *blast*)

lemma *min-product-set-finite*:
$(\bigwedge x.\ x \in X \Longrightarrow$ *finite* $x) \Longrightarrow$ *finite* $(\bigotimes_m X)$
by (*cases finite X*, *rotate-tac*, *induction X rule*: *finite-induct*) (*simp-all add*: *min-product-set-def*, *insert min-product-finite*, *blast*)

4.3 Disjunctive Normal Form

fun *dnf* :: *'a ltln* ⇒ *'a ltln fset set*
where
dnf $true_n = \{\{||\}\}$
| *dnf* $false_n = \{\}$
| *dnf* $(\varphi\ and_n\ \psi) =$ (*dnf* φ) $\otimes$ (*dnf* ψ)
| *dnf* $(\varphi\ or_n\ \psi) =$ (*dnf* φ) $\cup$ (*dnf* ψ)
| *dnf* $\varphi = \{\{|\varphi|\}\}$

fun *min-dnf* :: *'a ltln* ⇒ *'a ltln fset set*
where
min-dnf $true_n = \{\{||\}\}$
| *min-dnf* $false_n = \{\}$

| *min-dnf* (φ *and*$_n$ ψ) = (*min-dnf* φ) $\otimes_m$ (*min-dnf* ψ)
| *min-dnf* (φ *or*$_n$ ψ) = (*min-dnf* φ) $\cup_m$ (*min-dnf* ψ)
| *min-dnf* φ = {{|φ|}}

lemma *dnf-min-set*:
min-dnf φ = *min-set* (*dnf* φ)
by (*induction* φ) (*simp-all*, *simp-all only*: *min-product-def min-union-def*)

lemma *dnf-finite*:
finite (*dnf* φ)
by (*induction* φ) (*auto simp*: *product-finite*)

lemma *min-dnf-finite*:
finite (*min-dnf* φ)
by (*induction* φ) (*auto simp*: *min-product-finite min-union-finite*)

lemma *dnf-Abs-fset*[*simp*]:
fset (*Abs-fset* (*dnf* φ)) = *dnf* φ
by (*simp add*: *dnf-finite Abs-fset-inverse*)

lemma *min-dnf-Abs-fset*[*simp*]:
fset (*Abs-fset* (*min-dnf* φ)) = *min-dnf* φ
by (*simp add*: *min-dnf-finite Abs-fset-inverse*)

lemma *dnf-prop-atoms*:
$\Phi \in$ *dnf* $\varphi \Longrightarrow$ *fset* $\Phi \subseteq$ *prop-atoms* φ
by (*induction* φ *arbitrary*: Φ) (*auto simp*: *product-def*)

lemma *min-dnf-prop-atoms*:
$\Phi \in$ *min-dnf* $\varphi \Longrightarrow$ *fset* $\Phi \subseteq$ *prop-atoms* φ
using *dnf-min-set dnf-prop-atoms min-set-subset* **by** *blast*

lemma *min-dnf-atoms-dnf*:
$\Phi \in$ *min-dnf* $\psi \Longrightarrow \varphi \in$ *fset* $\Phi \Longrightarrow$ *dnf* φ = {{|φ|}}
proof (*induction* φ)
case *True-ltln*
then show *?case*
using *min-dnf-prop-atoms prop-atoms-notin*(*1*) **by** *blast*
next
case *False-ltln*
then show *?case*
using *min-dnf-prop-atoms prop-atoms-notin*(*2*) **by** *blast*
next
case (*And-ltln* φ*1* φ*2*)

then show *?case*
using *min-dnf-prop-atoms prop-atoms-notin*(*3*) **by** *force*
next
case (*Or-ltln* $\varphi 1$ $\varphi 2$)
then show *?case*
using *min-dnf-prop-atoms prop-atoms-notin*(*4*) **by** *force*
qed *auto*

lemma *min-dnf-min-set*[*simp*]:
min-set (*min-dnf* φ) = *min-dnf* φ
by (*induction* φ) (*simp-all add*: *min-set-def min-product-def min-union-def*, *blast*+)

lemma *min-dnf-iff-prop-assignment-subset*:
$\mathcal{A} \models_P \varphi \longleftrightarrow (\exists\, B.\ fset\ B \subseteq \mathcal{A} \wedge B \in min\text{-}dnf\ \varphi)$
proof
assume $\mathcal{A} \models_P \varphi$

then show $\exists\, B.\ fset\ B \subseteq \mathcal{A} \wedge B \in min\text{-}dnf\ \varphi$
proof (*induction* φ *arbitrary*: $\mathcal{A}$)
case (*And-ltln* φ_1 φ_2)

then obtain B_1 B_2 **where** *1*: $fset\ B_1 \subseteq \mathcal{A} \wedge B_1 \in min\text{-}dnf\ \varphi_1$ **and** *2*: $fset\ B_2 \subseteq \mathcal{A} \wedge B_2 \in min\text{-}dnf\ \varphi_2$
by *fastforce*

then obtain C **where** C $|\subseteq|$ B_1 $|\cup|$ B_2 **and** $C \in min\text{-}dnf\ \varphi_1 \otimes_m min\text{-}dnf\ \varphi_2$
using *min-product-comp* **by** *metis*

then show *?case*
by (*metis 1 2 le-sup-iff min-dnf.simps*(*3*) *sup.absorb-iff1 sup-fset.rep-eq*)
next
case (*Or-ltln* φ_1 φ_2)

{
assume $\mathcal{A} \models_P \varphi_1$

then obtain B **where** *1*: $fset\ B \subseteq \mathcal{A} \wedge B \in min\text{-}dnf\ \varphi_1$
using *Or-ltln* **by** *fastforce*

then obtain C **where** C $|\subseteq|$ B **and** $C \in min\text{-}dnf\ \varphi_1 \cup_m min\text{-}dnf\ \varphi_2$
using *min-union-comp* **by** *metis*

then have *?case*
by (*metis 1 dual-order.trans less-eq-fset.rep-eq min-dnf.simps*(*4*))
}

moreover

{
assume $\mathcal{A} \models_P \varphi_2$

then obtain B **where** *2*: *fset* $B \subseteq \mathcal{A} \wedge B \in$ *min-dnf* φ_2
using *Or-ltln* **by** *fastforce*

then obtain C **where** $C \mid\subseteq\mid B$ **and** $C \in$ *min-dnf* $\varphi_1 \cup_m$ *min-dnf* φ_2
using *min-union-comp min-union-comm* **by** *metis*

then have *?case*
by (*metis 2 dual-order.trans less-eq-fset.rep-eq min-dnf.simps*(*4*))
}

ultimately show *?case*
using *Or-ltln.prems* **by** *auto*
qed *simp-all*
next
assume $\exists B.$ *fset* $B \subseteq \mathcal{A} \wedge B \in$ *min-dnf* φ

then obtain B **where** *fset* $B \subseteq \mathcal{A}$ **and** $B \in$ *min-dnf* φ
by *auto*

then have *fset* $B \models_P \varphi$
by (*induction* φ *arbitrary*: B) (*auto simp*: *min-set-def min-product-def product-def min-union-def*)

then show $\mathcal{A} \models_P \varphi$
using ‹*fset* $B \subseteq \mathcal{A}$› **by** *blast*
qed

lemma *ltl-prop-implies-min-dnf*:
$\varphi \longrightarrow_P \psi = (\forall A \in$ *min-dnf* $\varphi. \exists B \in$ *min-dnf* $\psi. B \mid\subseteq\mid A)$
by (*meson less-eq-fset.rep-eq ltl-prop-implies-def min-dnf-iff-prop-assignment-subset order-refl dual-order.trans*)

lemma *ltl-prop-equiv-min-dnf*:

$\varphi \sim_P \psi = (\textit{min-dnf}\ \varphi = \textit{min-dnf}\ \psi)$
proof
assume $\varphi \sim_P \psi$

then have $\bigwedge x.\ x \in \textit{min-set}\ (\textit{min-dnf}\ \varphi) \longleftrightarrow x \in \textit{min-set}\ (\textit{min-dnf}\ \psi)$
unfolding *ltl-prop-implies-equiv ltl-prop-implies-min-dnf min-set-iff*
by *fastforce*

then show $\textit{min-dnf}\ \varphi = \textit{min-dnf}\ \psi$
by *auto*
qed (*simp add*: *ltl-prop-equiv-def min-dnf-iff-prop-assignment-subset*)

lemma *min-dnf-rep-abs*[*simp*]:
$\textit{min-dnf}\ (\textit{rep-ltln}_P\ (\textit{abs-ltln}_P\ \varphi)) = \textit{min-dnf}\ \varphi$
by (*simp add*: *ltl-prop-equiv-min-dnf*[*symmetric*] *Quotient3-ltln*$_P$ *rep-abs-rsp-left*)

4.4 Folding of and_n and or_n over Finite Sets

definition $\textit{And}_n :: 'a\ \textit{ltln set} \Rightarrow 'a\ \textit{ltln}$
where
$\textit{And}_n\ \Phi \equiv \textit{SOME}\ \varphi.\ \textit{fold-graph And-ltln True-ltln}\ \Phi\ \varphi$

definition $\textit{Or}_n :: 'a\ \textit{ltln set} \Rightarrow 'a\ \textit{ltln}$
where
$\textit{Or}_n\ \Phi \equiv \textit{SOME}\ \varphi.\ \textit{fold-graph Or-ltln False-ltln}\ \Phi\ \varphi$

lemma *fold-graph-*$\textit{And}_n$:
$\textit{finite}\ \Phi \Longrightarrow \textit{fold-graph And-ltln True-ltln}\ \Phi\ (\textit{And}_n\ \Phi)$
unfolding $\textit{And}_n$*-def* **by** (*rule someI2-ex*[*OF finite-imp-fold-graph*])

lemma *fold-graph-*$\textit{Or}_n$:
$\textit{finite}\ \Phi \Longrightarrow \textit{fold-graph Or-ltln False-ltln}\ \Phi\ (\textit{Or}_n\ \Phi)$
unfolding $\textit{Or}_n$*-def* **by** (*rule someI2-ex*[*OF finite-imp-fold-graph*])

lemma $\textit{Or}_n$*-empty*[*simp*]:
$\textit{Or}_n\ \{\} = \textit{False-ltln}$
by (*metis empty-fold-graphE finite.emptyI fold-graph-*$\textit{Or}_n$)

lemma $\textit{And}_n$*-empty*[*simp*]:
$\textit{And}_n\ \{\} = \textit{True-ltln}$
by (*metis empty-fold-graphE finite.emptyI fold-graph-*$\textit{And}_n$)

interpretation *dnf-union-thms*: *Finite-Set.comp-fun-commute* $\lambda\varphi.\ (\cup)\ (f\ \varphi)$

by *unfold-locales fastforce*

interpretation *dnf-product-thms*: *Finite-Set.comp-fun-commute* $\lambda\varphi.\ (\otimes)\ (f\ \varphi)$
by *unfold-locales* (*simp add*: *product-set-thms.comp-fun-commute*)

— Copied from locale *comp-fun-commute*
lemma *fold-graph-finite*:
assumes *fold-graph f z A y*
shows *finite A*
using *assms* **by** *induct simp-all*

Taking the DNF of And_n and Or_n is the same as folding over the individual DNFs.

lemma And_n*-dnf*:
$finite\ \Phi \Longrightarrow dnf\ (And_n\ \Phi) = Finite\text{-}Set.fold\ (\lambda\varphi.\ (\otimes)\ (dnf\ \varphi))\ \{\{||\}\}\ \Phi$
proof (*drule fold-graph-*And_n, *induction rule*: *fold-graph.induct*)
case (*insertI x A y*)

then have *finite A*
using *fold-graph-finite* **by** *fast*

then show *?case*
using *insertI* **by** *auto*
qed *simp*

lemma Or_n*-dnf*:
$finite\ \Phi \Longrightarrow dnf\ (Or_n\ \Phi) = Finite\text{-}Set.fold\ (\lambda\varphi.\ (\cup)\ (dnf\ \varphi))\ \{\}\ \Phi$
proof (*drule fold-graph-*Or_n, *induction rule*: *fold-graph.induct*)
case (*insertI x A y*)

then have *finite A*
using *fold-graph-finite* **by** *fast*

then show *?case*
using *insertI* **by** *auto*
qed *simp*

And_n and Or_n are injective on finite sets.

lemma And_n*-inj*:
inj-on And_n $\{s.\ finite\ s\}$
proof (*standard*, *simp*)
fix *x y* :: *'a ltln set*
assume *finite x* **and** *finite y*

```
  then have 1: fold-graph And-ltln True-ltln x (And_n x) and 2: fold-graph
And-ltln True-ltln y (And_n y)
    using fold-graph-And_n by blast+

  assume And_n x = And_n y

  with 1 show x = y
  proof (induction rule: fold-graph.induct)
    case emptyI
    then show ?case
      using 2 fold-graph.cases by force
  next
    case (insertI x A y)
    with 2 show ?case
    proof (induction arbitrary: x A y rule: fold-graph.induct)
      case (insertI x A y)
      then show ?case
        by (metis fold-graph.cases insertI1 ltln.distinct(7) ltln.inject(3))
    qed blast
  qed
qed

lemma Or_n-inj:
  inj-on Or_n {s. finite s}
proof (standard, simp)
  fix x y :: 'a ltln set
  assume finite x and finite y

  then have 1: fold-graph Or-ltln False-ltln x (Or_n x) and 2: fold-graph
Or-ltln False-ltln y (Or_n y)
    using fold-graph-Or_n by blast+

  assume Or_n x = Or_n y

  with 1 show x = y
  proof (induction rule: fold-graph.induct)
    case emptyI
    then show ?case
      using 2 fold-graph.cases by force
  next
    case (insertI x A y)
    with 2 show ?case
    proof (induction arbitrary: x A y rule: fold-graph.induct)
```

case (*insertI x A y*)
then show *?case*
by (*metis fold-graph.cases insertI1 ltln.distinct*(*27*) *ltln.inject*(*4*))
qed *blast*
qed
qed

The semantics of And_n and Or_n can be expressed using quantifiers.

lemma And_n*-semantics*:
finite $\Phi \Longrightarrow w \models_n And_n\ \Phi \longleftrightarrow (\forall \varphi \in \Phi.\ w \models_n \varphi)$
proof −
assume *finite* Φ
have $\bigwedge \psi.$ *fold-graph And-ltln True-ltln* $\Phi\ \psi \Longrightarrow w \models_n \psi \longleftrightarrow (\forall \varphi \in \Phi.\ w \models_n \varphi)$
by (*rule fold-graph.induct*) *auto*
then show *?thesis*
using And_n*-graph-And*$_n$[*OF* ‹*finite* Φ›] **by** *simp*
qed

lemma Or_n*-semantics*:
finite $\Phi \Longrightarrow w \models_n Or_n\ \Phi \longleftrightarrow (\exists \varphi \in \Phi.\ w \models_n \varphi)$
proof −
assume *finite* Φ
have $\bigwedge \psi.$ *fold-graph Or-ltln False-ltln* $\Phi\ \psi \Longrightarrow w \models_n \psi \longleftrightarrow (\exists \varphi \in \Phi.\ w \models_n \varphi)$
by (*rule fold-graph.induct*) *auto*
then show *?thesis*
using Or_n*-graph-Or*$_n$[*OF* ‹*finite* Φ›] **by** *simp*
qed

lemma And_n*-prop-semantics*:
finite $\Phi \Longrightarrow \mathcal{A} \models_P And_n\ \Phi \longleftrightarrow (\forall \varphi \in \Phi.\ \mathcal{A} \models_P \varphi)$
proof −
assume *finite* Φ
have $\bigwedge \psi.$ *fold-graph And-ltln True-ltln* $\Phi\ \psi \Longrightarrow \mathcal{A} \models_P \psi \longleftrightarrow (\forall \varphi \in \Phi.\ \mathcal{A} \models_P \varphi)$
by (*rule fold-graph.induct*) *auto*
then show *?thesis*
using And_n*-graph-And*$_n$[*OF* ‹*finite* Φ›] **by** *simp*
qed

lemma Or_n*-prop-semantics*:
finite $\Phi \Longrightarrow \mathcal{A} \models_P Or_n\ \Phi \longleftrightarrow (\exists \varphi \in \Phi.\ \mathcal{A} \models_P \varphi)$
proof −

assume *finite* Φ
have $\bigwedge\psi$. *fold-graph Or-ltln False-ltln* Φ $\psi \Longrightarrow \mathcal{A} \models_P \psi \longleftrightarrow (\exists\, \varphi \in \Phi.$ $\mathcal{A} \models_P \varphi)$
by (*rule fold-graph.induct*) *auto*
then show *?thesis*
using *fold-graph-*Or_n[*OF* ‹*finite* Φ›] **by** *simp*
qed

lemma Or_n-And_n-*image-semantics*:
assumes *finite* $\mathcal{A}$ **and** $\bigwedge\Phi.\ \Phi \in \mathcal{A} \Longrightarrow$ *finite* Φ
shows $w \models_n Or_n\ (And_n\ `\ \mathcal{A}) \longleftrightarrow (\exists\, \Phi \in \mathcal{A}.\ \forall\, \varphi \in \Phi.\ w \models_n \varphi)$
proof −
have $w \models_n Or_n\ (And_n\ `\ \mathcal{A}) \longleftrightarrow (\exists\, \Phi \in \mathcal{A}.\ w \models_n And_n\ \Phi)$
using Or_n-*semantics assms* **by** *auto*
then show *?thesis*
using And_n-*semantics assms* **by** *fast*
qed

lemma Or_n-And_n-*image-prop-semantics*:
assumes *finite* $\mathcal{A}$ **and** $\bigwedge\Phi.\ \Phi \in \mathcal{A} \Longrightarrow$ *finite* Φ
shows $\mathcal{I} \models_P Or_n\ (And_n\ `\ \mathcal{A}) \longleftrightarrow (\exists\, \Phi \in \mathcal{A}.\ \forall\, \varphi \in \Phi.\ \mathcal{I} \models_P \varphi)$
proof −
have $\mathcal{I} \models_P Or_n\ (And_n\ `\ \mathcal{A}) \longleftrightarrow (\exists\, \Phi \in \mathcal{A}.\ \mathcal{I} \models_P And_n\ \Phi)$
using Or_n-*prop-semantics assms* **by** *blast*
then show *?thesis*
using And_n-*prop-semantics assms* **by** *metis*
qed

4.5 DNF to LTL conversion

definition *ltln-of-dnf* :: *'a ltln fset set* ⇒ *'a ltln*
where
ltln-of-dnf $\mathcal{A} =$ $Or_n\ (And_n\ `$ *fset* $`\ \mathcal{A})$

lemma *ltln-of-dnf-semantics*:
assumes *finite* $\mathcal{A}$
shows $w \models_n$ *ltln-of-dnf* $\mathcal{A} \longleftrightarrow (\exists\, \Phi \in \mathcal{A}.\ \forall\, \varphi.\ \varphi\ |\in|\ \Phi \longrightarrow w \models_n \varphi)$
proof −
have $w \models_n$ *ltln-of-dnf* $\mathcal{A} \longleftrightarrow (\exists\, \Phi \in$ *fset* $`\ \mathcal{A}.\ \forall\, \varphi \in \Phi.\ w \models_n \varphi)$
unfolding *ltln-of-dnf-def*
proof (*rule* Or_n-And_n-*image-semantics*)
show *finite* (*fset* $`\ \mathcal{A}$)
using *assms* **by** *blast*
next

```
    show ⋀Φ. Φ ∈ fset ‘ 𝒜 ⟹ finite Φ
      by auto
  qed

  then show ?thesis
    by (metis image-iff)
qed

lemma ltln-of-dnf-prop-semantics:
  assumes finite 𝒜
  shows ℐ ⊨_P ltln-of-dnf 𝒜 ⟷ (∃Φ ∈ 𝒜. ∀φ. φ |∈| Φ ⟶ ℐ ⊨_P φ)
proof −
  have ℐ ⊨_P ltln-of-dnf 𝒜 ⟷ (∃Φ ∈ fset ‘ 𝒜. ∀φ ∈ Φ. ℐ ⊨_P φ)
    unfolding ltln-of-dnf-def
  proof (rule Or_n-And_n-image-prop-semantics)
    show finite (fset ‘ 𝒜)
      using assms by blast
  next
    show ⋀Φ. Φ ∈ fset ‘ 𝒜 ⟹ finite Φ
      by auto
  qed

  then show ?thesis
    by (metis image-iff)
qed

lemma ltln-of-dnf-prop-equiv:
  ltln-of-dnf (min-dnf φ) ∼_P φ
  unfolding ltl-prop-equiv-def
proof
  fix 𝒜
  have 𝒜 ⊨_P ltln-of-dnf (min-dnf φ) ⟷ (∃Φ ∈ min-dnf φ. ∀φ. φ |∈| Φ
⟶ 𝒜 ⊨_P φ)
    using ltln-of-dnf-prop-semantics min-dnf-finite by metis
  also have ... ⟷ (∃Φ ∈ min-dnf φ. fset Φ ⊆ 𝒜)
    by (metis min-dnf-prop-atoms prop-atoms-entailment-iff subset-eq)
  also have ... ⟷ 𝒜 ⊨_P φ
    using min-dnf-iff-prop-assignment-subset by blast
  finally show 𝒜 ⊨_P ltln-of-dnf (min-dnf φ) = 𝒜 ⊨_P φ .
qed

lemma min-dnf-ltln-of-dnf[simp]:
  min-dnf (ltln-of-dnf (min-dnf φ)) = min-dnf φ
  using ltl-prop-equiv-min-dnf ltln-of-dnf-prop-equiv by blast
```

4.6 Substitution in DNF formulas

definition *subst-clause* :: *'a ltln fset* ⇒ (*'a ltln* ⇀ *'a ltln*) ⇒ *'a ltln fset set*
where
subst-clause Φ *m* = ⨂*m* {*min-dnf* (*subst* φ *m*) | φ. φ ∈ *fset* Φ}

definition *subst-dnf* :: *'a ltln fset set* ⇒ (*'a ltln* ⇀ *'a ltln*) ⇒ *'a ltln fset set*
where
subst-dnf A *m* = (⋃ Φ ∈ A. *subst-clause* Φ *m*)

lemma *subst-clause-empty*[*simp*]:
subst-clause {||} *m* = {{||}}
by (*simp add*: *subst-clause-def*)

lemma *subst-dnf-empty*[*simp*]:
subst-dnf {} *m* = {}
by (*simp add*: *subst-dnf-def*)

lemma *subst-clause-inner-finite*:
finite {*min-dnf* (*subst* φ *m*) | φ. φ ∈ Φ} **if** *finite* Φ
using *that* **by** *simp*

lemma *subst-clause-finite*:
finite (*subst-clause* Φ *m*)
unfolding *subst-clause-def*
by (*auto intro*: *min-dnf-finite min-product-set-finite*)

lemma *subst-dnf-finite*:
finite A ⟹ *finite* (*subst-dnf* A *m*)
unfolding *subst-dnf-def* **using** *subst-clause-finite* **by** *blast*

lemma *subst-dnf-mono*:
A ⊆ B ⟹ *subst-dnf* A *m* ⊆ *subst-dnf* B *m*
unfolding *subst-dnf-def* **by** *blast*

lemma *subst-clause-min-set*[*simp*]:
min-set (*subst-clause* Φ *m*) = *subst-clause* Φ *m*
unfolding *subst-clause-def* **by** *simp*

lemma *subst-clause-finsert*[*simp*]:
subst-clause (*finsert* φ Φ) *m* = (*min-dnf* (*subst* φ *m*)) ⊗*m* (*subst-clause* Φ *m*)
proof −
have {*min-dnf* (*subst* ψ *m*) | ψ. ψ ∈ *fset* (*finsert* φ Φ)}

$= insert\ (min\text{-}dnf\ (subst\ \varphi\ m))\ \{min\text{-}dnf\ (subst\ \psi\ m) \mid \psi.\ \psi \in fset\ \Phi\}$
by *auto*

then show *?thesis*
by (*simp add*: *subst-clause-def*)
qed

lemma *subst-clause-funion*[*simp*]:
$subst\text{-}clause\ (\Phi\ |\cup|\ \Psi)\ m = (subst\text{-}clause\ \Phi\ m) \otimes_m (subst\text{-}clause\ \Psi\ m)$
proof (*induction* Ψ)
case (*insert x F*)
then show *?case*
using *min-product-set-thms.fun-left-comm* **by** *fastforce*
qed *simp*

For the proof of correctness, we redefine the $(\otimes)$ operator on lists.

definition *list-product* :: *'a list set* ⇒ *'a list set* ⇒ *'a list set* (**infixl** ‹$\otimes_l$› *65*)
where
$A \otimes_l B = \{a\ @\ b \mid a\ b.\ a \in A \wedge b \in B\}$

lemma *list-product-fset-of-list*[*simp*]:
fset-of-list ‘ $(A \otimes_l B) = (fset\text{-}of\text{-}list\ `\ A) \otimes (fset\text{-}of\text{-}list\ `\ B)$
unfolding *list-product-def product-def image-def* **by** *fastforce*

lemma *list-product-finite*:
finite $A \Longrightarrow$ *finite* $B \Longrightarrow$ *finite* $(A \otimes_l B)$
unfolding *list-product-def* **by** (*simp add*: *finite-image-set2*)

lemma *list-product-iff*:
$x \in A \otimes_l B \longleftrightarrow (\exists\, a\ b.\ a \in A \wedge b \in B \wedge x = a\ @\ b)$
unfolding *list-product-def* **by** *blast*

lemma *list-product-assoc*[*simp*]:
$A \otimes_l (B \otimes_l C) = A \otimes_l B \otimes_l C$
unfolding *set-eq-iff list-product-iff* **by** *fastforce*

Furthermore, we introduct DNFs where the clauses are represented as lists.

fun *list-dnf* :: *'a ltln* ⇒ *'a ltln list set*
where
$list\text{-}dnf\ true_n = \{[]\}$
| $list\text{-}dnf\ false_n = \{\}$
| $list\text{-}dnf\ (\varphi\ and_n\ \psi) = (list\text{-}dnf\ \varphi) \otimes_l (list\text{-}dnf\ \psi)$
| $list\text{-}dnf\ (\varphi\ or_n\ \psi) = (list\text{-}dnf\ \varphi) \cup (list\text{-}dnf\ \psi)$

| *list-dnf* $\varphi = \{[\varphi]\}$

definition *list-dnf-to-dnf* :: $'a$ *list set* $\Rightarrow$ $'a$ *fset set*
where
list-dnf-to-dnf X = *fset-of-list* ‘ *X*

lemma *list-dnf-to-dnf-list-dnf*[*simp*]:
list-dnf-to-dnf (*list-dnf* φ) = *dnf* φ
by (*induction* φ) (*simp-all add*: *list-dnf-to-dnf-def image-Un*)

lemma *list-dnf-finite*:
finite (*list-dnf* φ)
by (*induction* φ) (*simp-all add*: *list-product-finite*)

We use this to redefine *subst-clause* and *subst-dnf* on list DNFs.

definition *subst-clause'* :: $'a$ *ltln list* $\Rightarrow$ ($'a$ *ltln* $\rightharpoonup$ $'a$ *ltln*) $\Rightarrow$ $'a$ *ltln list set*
where
subst-clause' Φ *m* = *fold* ($\lambda\varphi$ *acc. acc* $\otimes_l$ *list-dnf* (*subst* φ *m*)) Φ $\{[]\}$

definition *subst-dnf'* :: $'a$ *ltln list set* $\Rightarrow$ ($'a$ *ltln* $\rightharpoonup$ $'a$ *ltln*) $\Rightarrow$ $'a$ *ltln list set*
where
subst-dnf' $\mathcal{A}$ *m* = ($\bigcup \Phi \in \mathcal{A}$. *subst-clause'* Φ *m*)

lemma *subst-clause'-finite*:
finite (*subst-clause'* Φ *m*)
by (*induction* Φ *rule*: *rev-induct*) (*simp-all add*: *subst-clause'-def list-dnf-finite list-product-finite*)

lemma *subst-clause'-nil*[*simp*]:
subst-clause' [] *m* = $\{[]\}$
by (*simp add*: *subst-clause'-def*)

lemma *subst-clause'-cons*[*simp*]:
subst-clause' (*xs* @ [*x*]) *m* = *subst-clause' xs m* $\otimes_l$ *list-dnf* (*subst x m*)
by (*simp add*: *subst-clause'-def*)

lemma *subst-clause'-append*[*simp*]:
subst-clause' (*A* @ *B*) *m* = *subst-clause' A m* $\otimes_l$ *subst-clause' B m*
proof (*induction B rule*: *rev-induct*)
case (*snoc x xs*)
then show *?case*
by *simp* (*metis append-assoc subst-clause'-cons*)
qed(*simp add*: *list-product-def*)

lemma *subst-dnf'-iff*:
x ∈ *subst-dnf'* *A* *m* ⟷ (∃ Φ ∈ *A*. *x* ∈ *subst-clause'* Φ *m*)
by (*simp add*: *subst-dnf'-def*)

lemma *subst-dnf'-product*:
subst-dnf' (*A* ⊗$_l$ *B*) *m* = (*subst-dnf'* *A* *m*) ⊗$_l$ (*subst-dnf'* *B* *m*) (**is** *?lhs* = *?rhs*)
proof (*unfold set-eq-iff*, *safe*)
fix *x*
assume *x* ∈ *?lhs*

then obtain Φ **where** Φ ∈ *A* ⊗$_l$ *B* **and** *x* ∈ *subst-clause'* Φ *m*
unfolding *subst-dnf'-iff* **by** *blast*

then obtain *a* *b* **where** *a* ∈ *A* **and** *b* ∈ *B* **and** Φ = *a* @ *b*
unfolding *list-product-def* **by** *blast*

then have *x* ∈ (*subst-clause'* *a* *m*) ⊗$_l$ (*subst-clause'* *b* *m*)
using ‹*x* ∈ *subst-clause'* Φ *m*› **by** *simp*

then obtain *a'* *b'* **where** *a'* ∈ *subst-clause'* *a* *m* **and** *b'* ∈ *subst-clause'* *b* *m* **and** *x* = *a'* @ *b'*
unfolding *list-product-iff* **by** *blast*

then have *a'* ∈ *subst-dnf'* *A* *m* **and** *b'* ∈ *subst-dnf'* *B* *m*
unfolding *subst-dnf'-iff* **using** ‹*a* ∈ *A*› ‹*b* ∈ *B*› **by** *auto*

then have ∃ *a*∈*subst-dnf'* *A* *m*. ∃ *b*∈*subst-dnf'* *B* *m*. *x* = *a* @ *b*
using ‹*x* = *a'* @ *b'*› **by** *blast*

then show *x* ∈ *?rhs*
unfolding *list-product-iff* **by** *blast*
next
fix *x*
assume *x* ∈ *?rhs*

then obtain *a* *b* **where** *a* ∈ *subst-dnf'* *A* *m* **and** *b* ∈ *subst-dnf'* *B* *m* **and** *x* = *a* @ *b*
unfolding *list-product-iff* **by** *blast*

then obtain *a'* *b'* **where** *a'* ∈ *A* **and** *b'* ∈ *B* **and** *a*: *a* ∈ *subst-clause'* *a'* *m* **and** *b*: *b* ∈ *subst-clause'* *b'* *m*
unfolding *subst-dnf'-iff* **by** *blast*

then have $x \in (\mathit{subst\text{-}clause}'\ a'\ m) \otimes_l (\mathit{subst\text{-}clause}'\ b'\ m)$
unfolding *list-product-iff* **using** ‹$x = a$ @ b› **by** *blast*

moreover

have a' @ $b' \in A \otimes_l B$
unfolding *list-product-iff* **using** ‹$a' \in A$› ‹$b' \in B$› **by** *blast*

ultimately show *?lhs*
unfolding *subst-dnf'-iff* **by** *force*
qed

lemma *subst-dnf'-list-dnf*:
subst-dnf' (*list-dnf* φ) m = *list-dnf* (*subst* φ m)
proof (*induction* φ)
case (*And-ltln* $\varphi 1$ $\varphi 2$)
then show *?case*
by (*simp add*: *subst-dnf'-product*)
qed (*simp-all add*: *subst-dnf'-def subst-clause'-def list-product-def*)

lemma *min-set-Union*:
finite $X \Longrightarrow$ *min-set* ($\bigcup$ (*min-set* ‘ X)) = *min-set* ($\bigcup$ X) **for** X :: *'a fset set set*
by (*induction X rule*: *finite-induct*) (*force*, *metis Sup-insert image-insert min-set-min-union min-union-def*)

lemma *min-set-Union-image*:
finite $X \Longrightarrow$ *min-set* ($\bigcup x \in X$. *min-set* (f x)) = *min-set* ($\bigcup x \in X$. f x)
for f :: *'b* ⇒ *'a fset set*
proof −
assume *finite X*

then have ∗: *finite* (f ‘ X) **by** *auto*

with *min-set-Union* **show** *?thesis*
unfolding *image-image* **by** *fastforce*
qed

lemma *subst-clause-fset-of-list*:
subst-clause (*fset-of-list* Φ) m = *min-set* (*list-dnf-to-dnf* (*subst-clause'* Φ m))
unfolding *list-dnf-to-dnf-def subst-clause'-def*

proof (*induction* Φ *rule*: *rev-induct*)
case (*snoc x xs*)
then show *?case*
by *simp* (*metis* (*no-types*, *lifting*) *dnf-min-set list-dnf-to-dnf-def list-dnf-to-dnf-list-dnf min-product-comm min-product-def min-set-min-product*(*1*))
qed *simp*

lemma *min-set-list-dnf-to-dnf-subst-dnf'*:
finite X ⟹ *min-set* (*list-dnf-to-dnf* (*subst-dnf'* *X m*)) = *min-set* (*subst-dnf* (*list-dnf-to-dnf X*) *m*)
by (*simp add*: *subst-dnf'-def subst-dnf-def subst-clause-fset-of-list list-dnf-to-dnf-def min-set-Union-image image-Union*)

lemma *subst-dnf-dnf*:
min-set (*subst-dnf* (*dnf* φ) *m*) = *min-dnf* (*subst* φ *m*)
unfolding *dnf-min-set*
unfolding *list-dnf-to-dnf-list-dnf*[*symmetric*]
unfolding *subst-dnf'-list-dnf*[*symmetric*]
unfolding *min-set-list-dnf-to-dnf-subst-dnf'*[*OF list-dnf-finite*]
by *simp*

This is almost the lemma we need. However, we need to show that the same holds for *min-dnf* φ, too.

lemma *fold-product*:
Finite-Set.fold (λ*x*. (⊗) {{|*x*|}}) {{||}} (*fset x*) = {*x*}
by (*induction x*) (*simp-all*, *simp add*: *product-singleton-singleton*)

lemma *fold-union*:
Finite-Set.fold (λ*x*. (∪) {*x*}) {} (*fset x*) = *fset x*
by (*induction x*) (*simp-all add*: *comp-fun-idem-on.fold-insert-idem*[*OF comp-fun-idem-insert*[*unfolded comp-fun-idem-def'*]])

lemma *fold-union-fold-product*:
assumes *finite X* **and** ⋀Ψ ψ. Ψ ∈ *X* ⟹ ψ ∈ *fset* Ψ ⟹ *dnf* ψ = {{|ψ|}}
shows *Finite-Set.fold* (λ*x*. (∪) (*Finite-Set.fold* (λφ. (⊗) (*dnf* φ)) {{||}} (*fset x*))) {} *X* = *X* (**is** *?lhs* = *X*)
proof −
from *assms* **have** *?lhs* = *Finite-Set.fold* (λ*x*. (∪) (*Finite-Set.fold* (λφ. (⊗) {{|φ|}}) {{||}} (*fset x*))) {} *X*
proof (*induction X rule*: *finite-induct*)
case (*insert* Φ *X*)

from *insert.prems* **have** *1*: ⋀Ψ ψ. ⟦Ψ ∈ *X*; ψ ∈ *fset* Ψ⟧ ⟹ *dnf* ψ = {{|ψ|}}

```
    by force

  from insert.prems have Finite-Set.fold (λφ. (⊗) (dnf φ)) {{||}} (fset
Φ) = Finite-Set.fold (λφ. (⊗) {{|φ|}}) {{||}} (fset Φ)
    by (induction Φ) force+

  with insert 1 show ?case
    by simp
 qed simp

 with ‹finite X› show ?thesis
  unfolding fold-product by (metis fset-to-fset fold-union)
qed

lemma dnf-ltln-of-dnf-min-dnf:
 dnf (ltln-of-dnf (min-dnf φ)) = min-dnf φ
proof −
 have 1: finite (And_n ‘ fset ‘ min-dnf φ)
  using min-dnf-finite by blast

 have 2: inj-on And_n (fset ‘ min-dnf φ)
  by (metis (mono-tags, lifting) And_n-inj f-inv-into-f fset inj-onI inj-on-contraD)

 have 3: inj-on fset (min-dnf φ)
  by (meson fset-inject inj-onI)

 show ?thesis
  unfolding ltln-of-dnf-def
  unfolding Or_n-dnf[OF 1]
  unfolding fold-image[OF 2]
  unfolding fold-image[OF 3]
  unfolding comp-def
  unfolding And_n-dnf[OF finite-fset]
  by (metis fold-union-fold-product min-dnf-finite min-dnf-atoms-dnf)
qed

lemma min-dnf-subst:
  min-set (subst-dnf (min-dnf φ) m) = min-dnf (subst φ m) (is ?lhs =
?rhs)
proof −
 let ?φ′ = ltln-of-dnf (min-dnf φ)

 have ?lhs = min-set (subst-dnf (dnf ?φ′) m)
  unfolding dnf-ltln-of-dnf-min-dnf ..
```

also have ... = *min-dnf* (*subst* ?φ' *m*)
unfolding *subst-dnf-dnf* **..**

also have ... = *min-dnf* (*subst* φ *m*)
using *ltl-prop-equiv-min-dnf ltln-of-dnf-prop-equiv subst-respects-ltl-prop-entailment*(*2*)
by *blast*

finally show *?thesis* **.**
qed

end

5 Code lemmas for abstract definitions

theory *Code-Equations*
imports
LTL Equivalence-Relations
Boolean-Expression-Checkers.Boolean-Expression-Checkers
Boolean-Expression-Checkers.Boolean-Expression-Checkers-AList-Mapping
begin

5.1 Propositional Equivalence

fun *ifex-of-ltl* :: '*a ltln* ⇒ '*a ltln ifex*
where
ifex-of-ltl true$_n$ = *Trueif*
| *ifex-of-ltl false*$_n$ = *Falseif*
| *ifex-of-ltl* (φ *and*$_n$ ψ) = *normif Mapping.empty* (*ifex-of-ltl* φ) (*ifex-of-ltl* ψ) *Falseif*
| *ifex-of-ltl* (φ *or*$_n$ ψ) = *normif Mapping.empty* (*ifex-of-ltl* φ) *Trueif* (*ifex-of-ltl* ψ)
| *ifex-of-ltl* φ = *IF* φ *Trueif Falseif*

lemma *val-ifex*:
val-ifex (*ifex-of-ltl b*) *s* = {*x*. *s x*} ⊨$_P$ *b*
by (*induction b*) (*simp add*: *agree-Nil val-normif*)+

lemma *reduced-ifex*:
reduced (*ifex-of-ltl b*) {}
by (*induction b*) (*simp*; *metis keys-empty reduced-normif*)+

lemma *ifex-of-ltl-reduced-bdt-checker*:
reduced-bdt-checkers ifex-of-ltl (λ*y s*. {*x*. *s x*} ⊨$_P$ *y*)

unfolding *reduced-bdt-checkers-def*
using *val-ifex reduced-ifex* **by** *blast*

lemma *ltl-prop-equiv-impl*[*code*]:
$(\varphi \sim_P \psi) =$ *equiv-test ifex-of-ltl* φ ψ
by (*simp add*: *ltl-prop-equiv-def reduced-bdt-checkers.equiv-test*[*OF ifex-of-ltl-reduced-bdt-checker*]; *fastforce*)

lemma *ltl-prop-implies-impl*[*code*]:
$(\varphi \longrightarrow_P \psi) =$ *impl-test ifex-of-ltl* φ ψ
by (*simp add*: *ltl-prop-implies-def reduced-bdt-checkers.impl-test*[*OF ifex-of-ltl-reduced-bdt-checker*]; *force*)

— Check code export
export-code $(\sim_P)$ $(\longrightarrow_P)$ **checking**

end

6 Example

theory *Example*
imports
../LTL ../Rewriting HOL−Library.Code-Target-Numeral
begin

— The included parser always returns a *String.literal ltlc*. If a different labelling is needed one can use *map-ltlc* to relabel the leafs. In our example we prepend a string to each atom.

definition *rewrite* :: *String.literal ltlc* ⇒ *String.literal ltlc*
where
rewrite ≡ *ltln-to-ltlc o rewrite-iter-slow o ltlc-to-ltln o* (*map-ltlc* (λ*s*. *String.implode* ″*prop*(″ + *s* + *String.implode* ″)″))

— Export rewriting engine (and also constructors)

export-code $true_c$ *Iff-ltlc rewrite* **in** *SML* **file-prefix** ‹*rewrite-example*›

end

References

[1] T. Babiak, M. Kretínský, V. Rehák, and J. Strejcek. LTL to büchi automata translation: Fast and more deterministic. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012.

[2] F. Somenzi and R. Bloem. Efficient büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.