Definitive Set Semantics for LTL3

Rayhana Amjad, Rob van Glabbeek, Liam O'Connor

March 17, 2025

LTL_3 is a multi-valued variant of Linear-time Temporal Logic for runtime verification applications, originally due to Bauer et al [3]. The semantic descriptions of LTL_3 in previous work are given only in terms of the relationship to conventional LTL. In this submission, which accompanies our EXPRESS/SOS 2024 paper [1], we instead give a full model-based inductive accounting of the semantics of LTL_3, in terms of families of *definitive prefix sets*. We show that our definitive prefix sets are isomorphic to linear-time temporal properties (sets of infinite traces), and thereby show that our semantics of LTL_3 directly correspond to the semantics of conventional LTL. In addition, we formalise the formula progression evaluation technique [2, 4], popularly used in runtime verification and testing contexts, and show its soundness and completeness up to finite traces with respect to our semantics.

Contents

1 Answer-Indexed Families **4**
- 1.1 Example: Propositional logic . . . 4
  - 1.1.1 Propositional logic lemmas . . . 5
  - 1.1.2 Propositional logic equivalence . . . 5

2 Traces and Definitive Prefixes **6**
- 2.1 Traces . . . 6
- 2.2 Prefix Closure . . . 11
- 2.3 Definitive Prefixes . . . 12
- 2.4 Definitive Sets . . . 13
- 2.5 A type for definitive sets . . . 15
- 2.6 Isomorphism of definitive sets and LTL properties . . . 16

3 Linear-time Temporal Logic **24**
- 3.1 Linear temporal logic equivalence . . . 25
- 3.2 Linear temporal logic lemmas . . . 25

4 LTL3: Semantics, Equivalence and Formula Progression **26**
- 4.1 LTL/LTL3 equivalence . . . 27
- 4.2 Equivalence to LTL3 of Bauer et al. . . . 31
- 4.3 Formula Progression . . . 32

theory *AnswerIndexedFamilies*
imports *Main*
begin

1 Answer-Indexed Families

typedecl *'a state*
consts *L* :: ‹*'a state* => *'a set*›
datatype *answer* = *T* | *F*
type-synonym *'a AiF* = ‹*answer* ⇒ *'a set*›

fun *and-AiF* :: ‹*'a AiF* ⇒ *'a AiF* ⇒ *'a AiF*› (**infixl** ‹∧▪› *60*) **where**
  ‹(*a* ∧▪ *b*) *T* = *a T* ∩ *b T*›
| ‹(*a* ∧▪ *b*) *F* = *a F* ∪ *b F*›

fun *or-AiF* :: ‹*'a AiF* ⇒ *'a AiF* ⇒ *'a AiF*› (**infixl** ‹∨▪› *59*) **where**
  ‹(*a* ∨▪ *b*) *T* = *a T* ∪ *b T*›
| ‹(*a* ∨▪ *b*) *F* = *a F* ∩ *b F*›

fun *not-AiF* :: ‹*'a AiF* ⇒ *'a AiF*› (‹¬▪ -›) **where**
  ‹(¬▪ *a*) *T* = *a F*›
| ‹(¬▪ *a*) *F* = *a T*›

fun *univ-AiF* :: ‹*'a AiF*› (‹*T*▪›) **where**
  ‹*T*▪ *T* = *UNIV*›
| ‹*T*▪ *F* = {}›

fun *satisfying-AiF* :: ‹*'a* ⇒ *'a state AiF*› (‹*sat*▪›) **where**
  ‹*sat*▪ *x T* = {*state*. *x* ∈ *L state*}›
| ‹*sat*▪ *x F* = {*state*. *x* ∉ *L state*}›

1.1 Example: Propositional logic

datatype (*atoms-plogic*: *'a*) *plogic* =
    True-plogic (‹*true*$_p$›)
  | *Prop-plogic* ‹*'a*› (‹*prop*$_p$*'*(-*')*›)
  | *Not-plogic* ‹*'a plogic*› (‹*not*$_p$ -› [*85*] *85*)
  | *Or-plogic* ‹*'a plogic*› ‹*'a plogic*› (‹- *or*$_p$ -› [*82*,*82*] *81*)
  | *And-plogic* ‹*'a plogic*› ‹*'a plogic*› (‹- *and*$_p$ -› [*82*,*82*] *81*)

fun *plogic-semantics* :: ‹*'a plogic* ⇒ *'a state AiF*› (‹⟦-⟧$_p$›) **where**
  ‹⟦ *true*$_p$ ⟧$_p$ = *T*▪›
| ‹⟦ *not*$_p$ *φ* ⟧$_p$ = ¬▪ ⟦*φ*⟧$_p$›
| ‹⟦ *prop*$_p$(*a*) ⟧$_p$ = *sat*▪ *a*›
| ‹⟦ *φ or*$_p$ *ψ* ⟧$_p$ = ⟦*φ*⟧$_p$ ∨▪ ⟦*ψ*⟧$_p$›
| ‹⟦ *φ and*$_p$ *ψ* ⟧$_p$ = ⟦*φ*⟧$_p$ ∧▪ ⟦*ψ*⟧$_p$›

definition *false-p* (‹*false*$_p$›) **where**

false-p-def [*simp*]: ‹$false_p = not_p\ true_p$›

definition *implies-p* :: ‹'*a plogic* ⇒ '*a plogic* ⇒ '*a plogic*› (‹- $implies_p$ -› [*81*,*81*] *80*) **where**
implies-p-def[*simp*]: ‹$\varphi\ implies_p\ \psi = (not_p\ \varphi\ or_p\ \psi)$›

1.1.1 Propositional logic lemmas

lemma *AiF-cases*:
assumes ‹$A\ T = B\ T$› **and** ‹$A\ F = B\ F$›
shows ‹$A = B$›
proof (*rule ext*)
fix x **show** ‹$A\ x = B\ x$› **by** (*cases* ‹x›; *simp add*: *assms*)
qed

lemma *or-and-negation*: ‹$[\![\ \varphi\ or_p\ \psi\ ]\!]_p = [\![\ not_p\ ((not_p\ \varphi)\ and_p\ (not_p\ \psi))\ ]\!]_p$›
by (*rule AiF-cases*; *simp*)

lemma *and-or-negation*: ‹$[\![\ \varphi\ and_p\ \psi]\!]_p = [\![\ not_p\ ((not_p\ \varphi)\ or_p\ (not_p\ \psi))\ ]\!]_p$›
by (*rule AiF-cases*; *simp*)

lemma *de-morgan-1*: ‹$[\![\ not_p\ (\varphi\ and_p\ \psi)\ ]\!]_p = [\![\ (not_p\ \varphi)\ or_p\ (not_p\ \psi)\ ]\!]_p$›
by (*rule AiF-cases*; *simp*)

lemma *de-morgan-2*: ‹$[\![\ not_p\ (\varphi\ or_p\ \psi)\ \ ]\!]_p = [\![\ (not_p\ \varphi)\ and_p\ (not_p\ \psi)\ ]\!]_p$›
by (*rule AiF-cases*; *simp*)

1.1.2 Propositional logic equivalence

fun *plogic-semantics'* :: ‹'*a state* ⇒ '*a plogic* ⇒ *bool*› (**infix** ‹$\models_p$› *60*) **where**
‹$\Gamma \models_p true_p = True$›
| ‹$\Gamma \models_p not_p\ \varphi = (\neg\ \Gamma \models_p \varphi)$›
| ‹$\Gamma \models_p prop_p(a) = (a \in L\ \Gamma)$›
| ‹$\Gamma \models_p \varphi\ or_p\ \psi = (\Gamma \models_p \varphi \vee \Gamma \models_p \psi)$›
| ‹$\Gamma \models_p \varphi\ and_p\ \psi = (\Gamma \models_p \varphi \wedge \Gamma \models_p \psi)$›

lemma *plogic-equivalence*:
shows ‹$(\ \ \Gamma\ \ \models_p \varphi \longleftrightarrow \Gamma \in [\![\varphi]\!]_p\ \ T)$›
and ‹$(\neg\ \Gamma\ \ \models_p \varphi \longleftrightarrow \Gamma \in [\![\varphi]\!]_p\ \ F)$›
proof (*induct* ‹φ›)
qed (*auto*)

end
theory *Traces*
imports *Main HOL.Lattices HOL.List*
begin

2 Traces and Definitive Prefixes

2.1 Traces

typedecl Σ
type-synonym *'a finite-trace* = ‹*'a list*›
type-synonym *'a infinite-trace* = ‹*nat* ⇒ *'a*›
datatype *'a trace* = *Finite* ‹*'a finite-trace*› | *Infinite* ‹*'a infinite-trace*›

fun *thead* :: ‹*'a trace* ⇒ *'a*› **where**
  ‹*thead* (*Finite t*) = *t* ! *0*›
| ‹*thead* (*Infinite t*) = *t 0*›

fun *append* :: ‹*'a trace* ⇒ *'a trace* ⇒ *'a trace*› (**infixr** ‹⌢› *80*) **where**
  ‹(*Finite t*) ⌢ (*Infinite ω*) = *Infinite* (λ*n*. *if n* < *length t then t*!*n else ω* (*n* − *length t*))›
| ‹(*Finite t*) ⌢ (*Finite u*) = *Finite* (*t* @ *u*)›
| ‹(*Infinite t*) ⌢ *u* = *Infinite t*›

definition *ε* :: ‹*'a trace*› **where**
  ‹*ε* = *Finite* []›

definition *singleton* :: ‹*'a* ⇒ *'a trace*› **where**
  ‹*singleton σ* = *Finite* [*σ*]›

interpretation *trace*: *monoid-list* ‹(⌢)› ‹*ε*›
proof *unfold-locales*
  fix *a* :: ‹*'a trace*› **show** ‹*ε* ⌢ *a* = *a*›
    by (*cases* ‹*a*›; *simp add*: *ε-def*)
next
  fix *a* :: ‹*'a trace*› **show** ‹*a* ⌢ *ε* = *a*›
    by (*cases* ‹*a*›; *simp add*: *ε-def*)
next
  fix *a b c* :: ‹*'a trace*› **show** ‹(*a* ⌢ *b*) ⌢ *c* = *a* ⌢ (*b* ⌢ *c*)›
    apply (*cases* ‹*a*›; *simp*)
    apply (*cases* ‹*b*›; *simp*)
    apply (*cases* ‹*c*›; *simp*)
    apply (*rule ext*; *simp*)
    by (*smt* (*verit*, *ccfv-threshold*)
        add.commute add-diff-inverse-nat add-less-cancel-left
        nth-append trans-less-add2)
qed

lemma *finite-empty-suffix*:
  assumes ‹*Finite xs* = *Finite xs* ⌢ *t*›

shows ‹$t = \varepsilon$›
using *assms* **by** (*cases* ‹t›) (*simp-all add*: ε-*def*)

lemma *finite-empty-prefix*:
assumes ‹*Finite xs* = t ⌢ *Finite xs*›
shows ‹$t = \varepsilon$›
using *assms* **by** (*cases* ‹t›) (*simp-all add*: ε-*def*)

lemma *finite-finite-suffix*:
assumes ‹*Finite xs* = *Finite ys* ⌢ t›
obtains *zs* **where** ‹t = *Finite zs*›
using *assms* **by** (*cases* ‹t›) (*simp-all*)

lemma *finite-finite-prefix*:
assumes ‹*Finite xs* = t ⌢ *Finite ys*›
obtains *zs* **where** ‹t = *Finite zs*›
using *assms* **by** (*cases* ‹t›) (*simp-all*)

lemma *append-is-empty*:
assumes ‹t ⌢ $u = \varepsilon$›
shows ‹$t = \varepsilon$›
and ‹$u = \varepsilon$›
using *assms* **by** (*simp add*: ε-*def*; *cases* ‹t›; *cases* ‹u›; *simp*)+

fun *ttake* :: ‹*nat* ⇒ *'a trace* ⇒ *'a finite-trace*› **where**
‹*ttake k* (*Finite xs*) = *take k xs*›
| ‹*ttake k* (*Infinite xs*) = *map xs* [*0*..<*k*] ›

definition *itdrop* :: ‹*nat* ⇒ *'a infinite-trace* ⇒ *'a infinite-trace*› **where**
‹*itdrop k xs* = (λ*i*. *xs* (*i* + *k*))›

lemma *itdrop-itdrop*[*simp*]: ‹*itdrop i* (*itdrop j x*) = *itdrop* (*i* + *j*) *x*›
by (*simp add*: *itdrop-def add.commute add.left-commute*)

lemma *itdrop-zero*[*simp*]: ‹*itdrop 0 x* = *x*›
by (*simp add*: *itdrop-def*)

fun *tdrop* :: ‹*nat* ⇒ *'a trace* ⇒ *'a trace*› **where**
‹*tdrop k* (*Finite xs*) = *Finite* (*drop k xs*)›
| ‹*tdrop k* (*Infinite xs*) = *Infinite* (*itdrop k xs*) ›

lemma *ttake-simp*[*simp*]: ‹*ttake* (*length xs*) (*Finite xs* ⌢ t) = *xs*›
by (*cases* ‹t›, *auto intro*: *list-eq-iff-nth-eq*[*THEN iffD2*])

lemma *ttake-tdrop*[*simp*]: ‹*Finite* (*ttake k t*) ⌢ *tdrop k t* = t›
by (*cases* ‹t›, *auto simp*: *itdrop-def*)

definition *prefixes* :: ‹'*a trace* ⇒ '*a trace set*› (‹↓ -› [*80*] *80*) **where**
‹↓ *t* = { *u* | *u v*. *t* = *u* ⌢ *v* }›

definition *extensions* :: ‹'*a trace* ⇒ '*a trace set*› (‹↑ -› [*80*] *80*) **where**
‹↑ *t* = { *t* ⌢ *u* | *u*. *True* }›

lemma *prefixes-extensions*: ‹*t* ∈ ↓ *u* ⟷ *u* ∈ ↑ *t*›
unfolding *prefixes-def extensions-def* **by** *simp*

interpretation *prefixes*: *order* ‹λ *t u*. *t* ∈ ↓ *u*› ‹λ *t u*. *t* ∈ ↓ *u* ∧ *t* ≠ *u*›
proof

fix *x* :: ‹'*a trace*›
show ‹*x* ∈ ↓ *x*›
unfolding *prefixes-def*
by (*simp*, *metis trace.right-neutral*)
next

fix *x y* :: ‹'*a trace*›
show ‹(*x* ∈ ↓ *y* ∧ *x* ≠ *y*) = (*x* ∈ ↓ *y* ∧ ¬ *y* ∈ ↓ *x*)›
unfolding *prefixes-def*
by (*simp*, *metis append.simps*(*3*) *append-is-empty*(*1*) *finite-empty-suffix*
trace.assoc trace.exhaust)
next

fix *x y* :: ‹'*a trace*›
assume *assms*: ‹*x* ∈ ↓ *y*› ‹*y* ∈ ↓ *x*›
show ‹*x* = *y*›
proof (*cases* ‹*y*›)
case *Finite* **note** *yfinite* = *this*
show ‹*?thesis*›
proof (*cases* ‹*x*›)
case *Finite*
with *assms*(*2*) **obtain** *z* **where** ‹*x* = *y* ⌢ *z*›
unfolding *prefixes-def*
by *auto*
with *assms*(*1*) *yfinite* **show** ‹*?thesis*›
unfolding *prefixes-def*
by (*force simp*: *trace.assoc dest*: *finite-empty-suffix append-is-empty*)
qed (*smt* (*verit*, *del-insts*) *CollectD append.simps*(*3*) *assms*(*1*) *prefixes-def*)
qed (*smt* (*verit*, *del-insts*) *CollectD append.simps*(*3*) *assms*(*2*) *prefixes-def*)
next

fix *x y z* :: ‹'*a trace*›
assume ‹*x* ∈ ↓ *y*› ‹*y* ∈ ↓ *z*›
then show ‹*x* ∈ ↓ *z*›
unfolding *prefixes-def* **by** (*force simp*: *trace.assoc*)
qed

lemma *prefixes-empty-least* : ‹$\varepsilon \in \downarrow t$›
by (*simp add*: *prefixes-def*)

lemma *prefixes-infinite-greatest* : ‹*Infinite x* $\in \downarrow t \Longrightarrow t =$ *Infinite x*›
by (*simp add*: *prefixes-def*)

lemma *prefixes-finite* : ‹*Finite xs* $\in \downarrow$ *Finite ys* $\longleftrightarrow$ ($\exists$ *zs*. *ys* = *xs* @ *zs*)›
proof (*rule iffI*)
show ‹*Finite xs* $\in \downarrow$ *Finite ys* $\Longrightarrow \exists$ *zs*. *ys* = *xs* @ *zs*›
using *finite-finite-suffix* **by** (*fastforce simp*: *prefixes-def*)
next
show ‹$\exists$ *zs*. *ys* = *xs* @ *zs* $\Longrightarrow$ *Finite xs* $\in \downarrow$ *Finite ys*›
by (*clarsimp simp*: *prefixes-def*) (*metis Traces.append.simps*(*2*))
qed

lemma *ttake-take* : ‹*take n* (*ttake m t*) = *ttake* (*min n m*) *t*›
by (*cases* ‹*t*›) (*simp-all add*: *min-def take-map*)

lemma *tdrop-tdrop* : ‹*tdrop n* (*tdrop m t*) = *tdrop* (*n* + *m*) *t*›
by (*cases* ‹*t*›) (*simp-all add*: *add.commute add.left-commute*)

lemma *tdrop-mono*: ‹*t* $\in \downarrow$ *u* $\Longrightarrow$ *tdrop k t* $\in \downarrow$ *tdrop k u*›
proof −
{ fix *v* **assume** *A*: ‹*u* = *t* $\frown$ *v*› **then have** ‹$\exists$ *va*. *tdrop k* (*t* $\frown$ *v*) = *tdrop k t* $\frown$ *va* ›
proof (*cases* ‹*t*›; *cases* ‹*v*›)
fix *x1 x2* **assume** ‹*t* = *Finite x1*› **and** ‹*v* = *Finite x2*› **with** *A* **show** ‹*?thesis*›
by (*simp*, *metis Traces.append.simps*(*2*))
next
fix *x1 x2* **assume** ‹*t* = *Finite x1*› **and** ‹*v* = *Infinite x2*› **with** *A*
have ‹*tdrop k* (*t* $\frown$ *v*) = *tdrop k t* $\frown$ *Infinite* (*itdrop* (*k* − *length x1*) *x2*) ›
apply *simp*
apply (*rule ext*)
apply *clarsimp*
apply (*rule conjI*)
apply (*simp add*: *add.commute itdrop-def less-diff-conv*)
by (*smt* (*z3*) *add.commute add-diff-cancel-left′ add-diff-inverse-nat diff-is-0-eq′*
diff-right-commute itdrop-def linorder-not-less nat-less-le)
then show ‹$\exists$ *va*. *tdrop k* (*t* $\frown$ *v*) = *tdrop k t* $\frown$ *va*›
by *auto*
qed *auto* **} note** *A* = *this*
assume ‹*t* $\in \downarrow$ *u*› **with** *A* **show** *?thesis* **unfolding** *prefixes-def* **by** *clarsimp*
qed

lemma *ttake-finite-prefixes* : ‹*Finite xs* $\in \downarrow t \longleftrightarrow$ *xs* = *ttake* (*length xs*) *t*›
proof (*rule iffI*)
show ‹*Finite xs* $\in \downarrow t \Longrightarrow$ *xs* = *ttake* (*length xs*) *t*›

by (*clarsimp simp*: *prefixes-def*)
next
show ‹*xs* = *ttake* (*length xs*) *t* ⟹ *Finite xs* ∈ ↓ *t*›
unfolding *prefixes-def* **using** *ttake-tdrop*
by (*metis* (*full-types*) *mem-Collect-eq*)
qed

lemma *ttake-prefixes* : ‹*a* ≤ *b* ⟹ *Finite* (*ttake a t*) ∈ ↓ *Finite* (*ttake b t*)›
by (*cases* ‹*t*›; *simp add*: *ttake-finite-prefixes min-def take-map*)

lemma *finite-directed*:
assumes ‹ *Finite xs* ∈ ↓ *t* › ‹ *Finite ys* ∈ ↓ *t* ›
shows ‹ ∃ *zs*. (*xs* = *ys* @ *zs*) ∨ (*ys* = *xs* @ *zs*) ›
proof (*cases* ‹*length xs* > *length ys*›)
case *True*
with *assms* **show** ‹*?thesis*›
apply (*simp add*: *ttake-finite-prefixes*)
using *ttake-prefixes*[*simplified prefixes-finite*]
by (*metis less-le-not-le*)
next
case *False*
from *assms this*[*THEN leI*] **show** ‹*?thesis*›
apply (*simp add*: *ttake-finite-prefixes*)
using *ttake-prefixes*[*simplified prefixes-finite*]
by (*metis*)
qed

lemma *prefixes-directed*: ‹*u* ∈ ↓ *t* ⟹ *v* ∈ ↓ *t* ⟹ *u* ∈ ↓ *v* ∨ *v* ∈ ↓ *u*›
proof (*cases* ‹*v*›; *cases* ‹*u*›)
{ fix *a b* **assume** ‹*Finite a* ∈ ↓ *t*› ‹*Finite b* ∈ ↓ *t*›
then have ‹*Finite a* ∈ ↓ *Finite b* ∨ *Finite b* ∈ ↓ *Finite a*›
using *finite-directed prefixes-finite* **by** *blast* **} note** *X* = *this*
fix *a b* **show** ‹*u* ∈ ↓ *t* ⟹ *v* ∈ ↓ *t* ⟹ *v* = *Finite a* ⟹ *u* = *Finite b* ⟹ *u* ∈ ↓ *v* ∨ *v* ∈ ↓ *u*›
using *X* **by** *auto*
qed (*auto simp*: *prefixes-def dest*: *prefixes-infinite-greatest*)

interpretation *extensions*: *order* ‹λ *t u*. *t* ∈ ↑ *u*› ‹λ *t u*. *t* ∈ ↑ *u* ∧ *t* ≠ *u*›
proof
qed (*auto simp*: *prefixes-extensions*[*THEN sym*] *dest*: *prefixes.leD intro*:*prefixes.order.trans*)

lemma *extensions-infinite*[*simp*]: ‹↑ *Infinite xs* = { *Infinite xs* }›
by (*simp add*: *extensions-def*)

lemma *extensions-empty*[*simp*]: ‹↑ ε = *UNIV*›
by (*simp add*: *extensions-def*)

lemma *prefixes-empty*: ‹↓ ε = {ε}›
apply (*clarsimp simp add*: *set-eq-iff ε-def prefixes-def*)

apply (*rule iffI*)
apply (*metis ε-def append-is-empty(1)*)
by (*metis ε-def trace.left-neutral*)

2.2 Prefix Closure

definition *prefix-closure* :: ‹*'a trace set* ⇒ *'a trace set*› (‹↓$_s$ -› [*80*] *80*) **where**
‹↓$_s$ *X* = (⋃ *t* ∈ *X*. *prefixes t*) ›

lemma *prefix-closure-subset*: ‹*X* ⊆ ↓$_s$ *X*›
unfolding *prefix-closure-def*
by *auto*

lemma *prefix-closure-infinite*: ‹*Infinite x* ∈ ↓$_s$ *X* ⟷ *Infinite x* ∈ *X*›
proof
assume ‹*Infinite x* ∈ ↓$_s$ *X*› **then show** ‹*Infinite x* ∈ *X*›
by (*metis UN-E prefix-closure-def prefixes-infinite-greatest*)
next
assume ‹*Infinite x* ∈ *X*› **then show** ‹*Infinite x* ∈ ↓$_s$ *X*›
by (*meson in-mono prefix-closure-subset*)
qed

lemma *prefix-closure-idem*: ‹↓$_s$ ↓$_s$ *X* = ↓$_s$ *X*›
unfolding *prefix-closure-def*
using *prefixes.order.trans* **by** *blast*

lemma *prefix-closure-mono*: ‹*X* ⊆ *Y* ⟹ ↓$_s$ *X* ⊆ ↓$_s$ *Y*›
unfolding *prefix-closure-def*
by *blast*

lemma *prefix-closure-union-distrib*: ‹↓$_s$ (*X* ∪ *Y*) = ↓$_s$ *X* ∪ ↓$_s$ *Y*›
unfolding *prefix-closure-def*
by *simp*

lemma *prefix-closure-Union-distrib*: ‹↓$_s$ (⋃ *S*) = ⋃ (*prefix-closure* ‘ *S*)›
unfolding *prefix-closure-def*
by *simp*

lemma *prefix-closure-Inter*: ‹↓$_s$ (⋂ (*prefix-closure* ‘ *S*)) = ⋂ (*prefix-closure* ‘ *S*) ›
unfolding *prefix-closure-def*
using *prefixes.dual-order.trans* **by** *fastforce*

lemma *prefix-closure-inter*: ‹↓$_s$ (↓$_s$ *X* ∩ ↓$_s$ *Y*) = ↓$_s$ *X* ∩ ↓$_s$ *Y*›
by (*rule prefix-closure-Inter*[**where** *S* = ‹{*X*, *Y*}›, *simplified*])

lemma *prefix-closure-UNIV*: ‹↓$_s$ *UNIV* = *UNIV*›
unfolding *prefix-closure-def* **by** *blast*

lemma *prefix-closure-empty*: ‹↓$_s$ {} = {}›

unfolding *prefix-closure-def* **by** *blast*

lemma *prefix-closure-extensions*: ‹ $\downarrow_s (\uparrow t) = \uparrow t \cup \downarrow t$›
by (*force intro*: *prefix-closure-subset dest*: *prefixes-directed*
simp: *prefixes-extensions*[*THEN sym*] *prefix-closure-def*)

lemma *prefix-closure-prefixes*: ‹$\downarrow_s (\downarrow t) = \downarrow t$›
unfolding *prefix-closure-def*
by (*force intro*: *prefixes.dual-order.trans*)

2.3 Definitive Prefixes

definition *dprefixes* :: ‹*'a trace set* ⇒ *'a trace set*› (‹$\downarrow_d$ -› [*80*] *80*) **where**
‹$\downarrow_d X = \{ t \mid t. \uparrow t \subseteq \downarrow_s X \}$›

lemma *dprefixes-are-prefixes* : ‹$\downarrow_d X \subseteq \downarrow_s X$›
unfolding *dprefixes-def*
using *extensions.order.refl* **by** *blast*

lemma *prefix-closure-dprefixes* : ‹$\downarrow_s (\downarrow_d X) \subseteq \downarrow_s X$›
using *dprefixes-are-prefixes prefix-closure-idem prefix-closure-mono*
by *blast*

lemma *dprefixes-idem*: ‹$\downarrow_d \downarrow_d X = \downarrow_d X$›
proof
show ‹$\downarrow_d \downarrow_d X \subseteq \downarrow_d X$›
using *prefix-closure-dprefixes*
by (*force simp*: *dprefixes-def*)
next
show ‹$\downarrow_d X \subseteq \downarrow_d \downarrow_d X$›
using *extensions.order.trans prefix-closure-subset*
by (*force simp*: *dprefixes-def*)
qed

lemma *dprefixes-contains-extensions*: ‹$t \in \downarrow_d X \Longrightarrow \uparrow t \subseteq \downarrow_d X$›
unfolding *dprefixes-def*
using *extensions.dual-order.trans* **by** *auto*

lemma *dprefixes-infinite*: ‹*Infinite* $x \in \downarrow_d X \longleftrightarrow$ *Infinite* $x \in X$›
proof
show ‹*Infinite* $x \in X \Longrightarrow$ *Infinite* $x \in \downarrow_d X$›
unfolding *dprefixes-def*
using *prefix-closure-subset* **by** *fastforce*
next
show ‹*Infinite* $x \in \downarrow_d X \Longrightarrow$ *Infinite* $x \in X$›
unfolding *dprefixes-def*
by (*clarsimp simp*: *prefix-closure-infinite*)
qed

lemma *dprefixes-UNIV*: ‹$\downarrow_d$ *UNIV* = *UNIV*›
unfolding *dprefixes-def*
using *prefix-closure-UNIV* **by** *force*

lemma *dprefixes-empty*: ‹$\downarrow_d$ {} = {}›
unfolding *dprefixes-def*
using *prefix-closure-empty* **by** *blast*

lemma *dprefixes-Inter-distrib*: ‹$\downarrow_d$ ($\bigcap$ *S*) ⊆ $\bigcap$ (*dprefixes* ‘ *S*)›
unfolding *dprefixes-def prefix-closure-def*
by *auto*

lemma *dprefixes-Inter*: ‹$\downarrow_d$ ($\bigcap$ (*dprefixes* ‘ *S*)) = $\bigcap$ (*dprefixes* ‘ *S*)›
proof
show ‹$\bigcap$ (*dprefixes* ‘ *S*) ⊆ $\downarrow_d$ $\bigcap$ (*dprefixes* ‘ *S*)›
unfolding *dprefixes-def prefix-closure-def*
using *prefixes.order.refl extensions.dual-order.trans*
by *force*
next
show ‹$\downarrow_d$ $\bigcap$ (*dprefixes* ‘ *S*) ⊆ $\bigcap$ (*dprefixes* ‘ *S*)›
using *dprefixes-idem dprefixes-Inter-distrib*
by *blast*
qed

lemma *dprefixes-mono*:
assumes ‹*X* ⊆ *Y*›
shows ‹$\downarrow_d$ *X* ⊆ $\downarrow_d$ *Y*›
using *assms*
apply (*simp add*: *dprefixes-def*)
apply (*simp add*: *prefix-closure-def*)
apply (*rule subsetI*)
using *prefixes-extensions* **by** *blast*

lemma *dprefixes-inter*: ‹$\downarrow_d$ ($\downarrow_d$ *X* ∩ $\downarrow_d$ *Y*) = ($\downarrow_d$ *X* ∩ $\downarrow_d$ *Y*)›
by (*rule dprefixes-Inter*[**where** *S* = ‹{*X*, *Y*}›, *simplified*])

lemma *dprefixes-inter-distrib*: ‹$\downarrow_d$ (*X* ∩ *Y*) ⊆ $\downarrow_d$ *X* ∩ $\downarrow_d$ *Y*›
using *dprefixes-Inter-distrib*[**where** *S* = ‹{*X*, *Y*}›] **by** *auto*

2.4 Definitive Sets

definition *definitive*:: ‹'*a trace set* ⇒ *bool*› **where**
‹*definitive X* ⟷ $\downarrow_d$ *X* = *X*›

lemma *definitive-image*: ‹∀ *X* ∈ *S*. *definitive X* ⟹ *dprefixes* ‘ *S* = *S*›
unfolding *definitive-def* **by** *auto*

lemma *definitive-dprefixes*: ‹*definitive* ($\downarrow_d$ *X*)›
unfolding *definitive-def* **by** (*rule dprefixes-idem*)

lemma *definitive-contains-extensions*: ‹*definitive X* $\Longrightarrow$ *t* $\in$ *X* $\Longrightarrow$ $\uparrow$ *t* $\subseteq$ *X*›
unfolding *definitive-def* **using** *dprefixes-contains-extensions* **by** *blast*

lemma *definitive-UNIV*: ‹*definitive UNIV*›
unfolding *definitive-def* **by** (*rule dprefixes-UNIV*)

lemma *definitive-empty*: ‹*definitive* {}›
unfolding *definitive-def* **by** (*rule dprefixes-empty*)

lemma *definitive-Inter*: ‹$\forall$ *X* $\in$ *S*. *definitive X* $\Longrightarrow$ *definitive* ($\bigcap$ *S*)›
unfolding *definitive-def* **using** *dprefixes-Inter definitive-image*[*simplified definitive-def*]
by *metis*

lemma *definitive-inter*: ‹*definitive X* $\Longrightarrow$ *definitive Y* $\Longrightarrow$ *definitive* (*X* $\cap$ *Y*)›
using *definitive-Inter*[**where** *S* = ‹{*X*, *Y*}›, *simplified*] **by** *blast*

lemma *definitive-infinite-extension*:
assumes ‹*definitive X*› **and** ‹*t* $\in$ *X*›
shows ‹$\exists$ *f*. *Infinite f* $\in$ *X* $\wedge$ *t* $\in$ $\downarrow$ *Infinite f*›
using *assms* **proof** (*cases* ‹*t*›)
case (*Finite xs*) **then show** ‹*?thesis*›
apply (*intro exI*[**where** *x*=‹λ*n*. *if n* < *length xs then xs*!*n else undefined*›])
by (*force simp*: *prefixes-extensions*[*THEN sym*] *prefixes-def*
intro!: *definitive-contains-extensions*[*THEN subsetD*, *OF assms*]
intro: *exI*[**where** *x*=‹*Infinite* (λ-. *undefined*)›])
qed *auto*

lemma *definitive-elemI*:
assumes ‹*definitive X*› ‹$\uparrow$ *t* $\subseteq$ $\downarrow_s$ *X*›
shows ‹*t* $\in$ *X*›
using *assms*
by (*auto simp add*: *definitive-def dprefixes-def*)

definition *dUnion* :: ‹'*a trace set set* $\Rightarrow$ '*a trace set*› (‹$\bigcup_d$›) **where**
‹$\bigcup_d$ *X* = $\downarrow_d$ $\bigcup$ *X*›

abbreviation *dunion* :: ‹'*a trace set* $\Rightarrow$ '*a trace set* $\Rightarrow$ '*a trace set*› (**infixl** ‹$\cup_d$› *65*) **where**
‹*X* $\cup_d$ *Y* $\equiv$ $\bigcup_d$ {*X*, *Y*}›

lemma *dprefixes-dUnion*: ‹$\downarrow_d$ $\bigcup_d$ *S* = $\bigcup_d$ *S*›
by (*simp add*: *dUnion-def dprefixes-idem*)

lemma *definitive-dUnion*: ‹*definitive* ($\bigcup_d$ *S*)›
by (*simp add*: *dprefixes-dUnion definitive-def*)

lemma *dUnion-contains-dprefixes*: ‹$t \in S \Longrightarrow \downarrow_d t \subseteq \bigcup_d S$›
by (*auto simp*: *dUnion-def dprefixes-def prefix-closure-def*)

lemma *dUnion-contains-definitive*: ‹$X \in S \Longrightarrow$ *definitive* $X \Longrightarrow X \subseteq \bigcup_d S$›
unfolding *definitive-def*
using *dUnion-contains-dprefixes* **by** *blast*

lemma *dUnion-empty*[*simp*]: ‹$\bigcup_d \{\} = \{\}$›
unfolding *dUnion-def*
by (*simp add*: *dprefixes-empty*)

lemma *dUnion-least-dprefixes*: ‹$(\bigwedge X.\ X \in S \Longrightarrow X \subseteq \downarrow_d Z) \Longrightarrow \downarrow_d (\bigcup (\mathit{dprefixes}\ `\ S)) \subseteq \downarrow_d Z$›
unfolding *dprefixes-def prefix-closure-def*
by (*simp add*: *subset-iff*, *meson extensions.order-refl prefixes.order.trans*)

lemma *dUnion-least-definitive*:
assumes *all-defn*: ‹$\forall X \in S.$ *definitive* X›
shows ‹$(\bigwedge X.\ X \in S \Longrightarrow X \subseteq Z) \Longrightarrow$ *definitive* $Z \Longrightarrow \downarrow_d \bigcup S \subseteq Z$›
using *definitive-image*[*OF all-defn*,*THEN sym*] *dUnion-least-dprefixes definitive-def*
by *metis*

2.5 A type for definitive sets

typedef *'a dset* = ‹$\{p :: 'a\ \mathit{trace}\ \mathit{set}.\ \mathit{definitive}\ p\ \}$›
using *definitive-UNIV* **by** *blast*

setup-lifting *type-definition-dset*

lift-definition *Inter-dset* :: ‹*'a dset set* ⇒ *'a dset*› (‹$\bigsqcap$›) **is** ‹λ *ss*. $\bigcap$ *ss*›
by (*simp add*: *definitive-Inter*)

abbreviation *inter-dset* :: ‹*'a dset* ⇒ *'a dset* ⇒ *'a dset*› (**infixl** ‹$\sqcap$› *66*) **where**
‹$X \sqcap Y \equiv \bigsqcap \{X, Y\}$›

lift-definition *Union-cset* :: ‹*'a dset set* ⇒ *'a dset*› (‹$\bigsqcup$›) **is** ‹λ *ss*. $\bigcup_d$ *ss*›
by (*rule definitive-dUnion*)

abbreviation *union-dset* :: ‹*'a dset* ⇒ *'a dset* ⇒ *'a dset*› (**infixl** ‹$\sqcup$› *65*) **where**
‹$X \sqcup Y \equiv \bigsqcup \{X, Y\}$›

lift-definition *empty-dset* :: ‹*'a dset*› (‹$\emptyset$›) **is** ‹$\{\}$›
by (*rule definitive-empty*)

lift-definition *univ-dset* :: ‹*'a dset*› (‹$\Sigma\infty$›) **is** ‹*UNIV*›
by (*rule definitive-UNIV*)

lift-definition *subset-dset* :: ‹*'a dset* ⇒ *'a dset* ⇒ *bool*› (**infix** ‹$\sqsubseteq$› *50*) **is** ‹$(\subseteq)$›
done

lift-definition *strict-subset-cset* :: ‹*'a dset ⇒ 'a dset ⇒ bool*› (**infix** ‹⊏› *50*) **is** ‹(⊂)›
done

lift-definition *in-dset* :: ‹*'a trace ⇒ 'a dset ⇒ bool*› **is** ‹(∈)›
done

lift-definition *notin-dset* :: ‹*'a trace ⇒ 'a dset ⇒ bool*› **is** ‹(∉)›
done

lemma *in-dset-ε*: ‹*in-dset ε A ⟹ A = Σ∞*›
apply (*transfer*)
using *definitive-contains-extensions extensions-empty* **by** *blast*

lemma *in-dset-UNIV*: ‹*in-dset x Σ∞*›
by (*transfer*, *simp*)

lemma *in-dset-subset*: ‹*A ⊑ B ⟹ in-dset x A ⟹ in-dset x B*›
by (*transfer*, *auto*)

lemma *in-dset-inter*: ‹*in-dset x A ⟹ in-dset x B ⟹ in-dset x (A ⊓ B)*›
by (*transfer*, *simp*)

interpretation *dset*: *complete-lattice* ‹⨅› ‹⨆› ‹(⊓)› ‹(⊑)› ‹(⊏)› ‹(⊔)› ‹∅› ‹*Σ∞*›
proof (*unfold-locales*;*transfer*)
fix *X Y Z* :: ‹*'a trace set*› **assume** ‹*definitive X*› ‹*definitive Y*› ‹*definitive Z*›
then show ‹ *Y ⊆ X ⟹ Z ⊆ X ⟹ (Y ∪d Z) ⊆ X*›
by (*metis dUnion-def dUnion-least-definitive insert-iff singletonD*)
next
fix *A* :: ‹*'a trace set set*› **and** *Z* :: ‹*'a trace set*›
assume ‹*∀ X∈A. definitive X*› ‹*definitive Z*› ‹(*⋀X. definitive X ⟹ X ∈ A ⟹ X ⊆ Z*)›
then show ‹*⋃d A ⊆ Z*›
by (*simp add*: *dUnion-def dUnion-least-definitive*)
qed (*auto simp*: *dUnion-contains-definitive*)

2.6 Isomorphism of definitive sets and LTL properties

definition *infinites* :: ‹*'a trace set ⇒ 'a infinite-trace set*› **where**
‹*infinites X = (⋃ x ∈ X. case x of Finite xs ⇒ {} | Infinite xs ⇒ {xs})*›

lemma *infinites-alt*: ‹*Infinite ‘ infinites A = A ∩ range Infinite*›
unfolding *set-eq-iff* **proof**
fix *x* **{ assume** ‹ (*x ∈ Infinite ‘ infinites A*)› **hence** ‹ (*x ∈ A ∩ range Infinite*) ›
by (*clarsimp simp*: *infinites-def split*!: *trace.split-asm*)
} moreover { assume ‹ (*x ∈ A ∩ range Infinite*) › **hence** ‹ (*x ∈ Infinite ‘ infinites A*) ›
by (*force simp*: *infinites-def split*!: *trace.split intro*!: *imageI*)

} ultimately show ‹ $(x \in \mathit{Infinite}\ `\ \mathit{infinites}\ A) = (x \in A \cap \mathit{range}\ \mathit{Infinite})$›
by *blast*
qed

lemma *infinites-append-right*: ‹$t \frown \mathit{Infinite}\ \omega \in \mathit{range}\ \mathit{Infinite}$›
by (*cases* ‹*t*›; *auto*)

lemma *infinites-prefix-closure*:
assumes ‹*definitive X*›
shows ‹$\downarrow_s \mathit{Infinite}\ `\ \mathit{infinites}\ X = \downarrow_s X$›
unfolding *prefix-closure-def infinites-def*
using *definitive-infinite-extension*[*OF assms*] *prefixes.order.trans*
by (*force split*: *trace.split-asm*)

lemma *infinites-UNIV*[*simp*]: ‹*infinites UNIV* = *UNIV*›
by (*auto simp*: *infinites-def split*: *trace.split*)

lemma *infinites-empty*[*simp*]: ‹*infinites* {} = {}›
by (*auto simp*: *infinites-def*)

lemma *infinites-Inter*: ‹*infinites* $(\bigcap\ S) = \bigcap\ (\mathit{infinites}\ `\ S)$›
unfolding *infinites-def*
apply (*rule set-eqI*; *rule iffI*)
apply (*force*)
apply (*simp split*: *trace.split trace.split-asm*)
by (*metis InterI trace.distinct*(*1*) *trace.exhaust trace.inject*(*2*))

lemma *infinites-Union*: ‹*infinites* $(\bigcup\ S) = \bigcup\ (\mathit{infinites}\ `\ S)$›
unfolding *infinites-def*
by *auto*

lemma *infinites-dprefixes*: ‹*infinites* $(\downarrow_d X) = \mathit{infinites}\ X$›
unfolding *infinites-def*
by (*force simp*: *dprefixes-infinite split*: *trace.split trace.split-asm*)

lemma *infinites-dprefixes-Infinite*: ‹*infinites* $(\downarrow_d \mathit{Infinite}\ `\ X) = X$›
proof
show ‹*infinites* $(\downarrow_d \mathit{Infinite}\ `\ X) \subseteq X$›
unfolding *infinites-def*
using *prefixes-infinite-greatest*
by (*force split*: *trace.split-asm simp*: *dprefixes-def prefix-closure-def*)
next
show ‹$X \subseteq$ *infinites* $(\downarrow_d \mathit{Infinite}\ `\ X)$›
by (*force simp*: *infinites-def dprefixes-def prefix-closure-def split*: *trace.split*)
qed

lift-definition *property* :: ‹$'a$ *dset* $\Rightarrow$ $'a$ *infinite-trace set*› **is** ‹*infinites*›
done

lift-definition *definitives* :: ‹*'a infinite-trace set* ⇒ *'a dset*› **is** ‹λ*x*. ↓$_d$ (*Infinite* ‘ *x*)›
  by (*rule definitive-dprefixes*)

lemma *property-inverse*: ‹*property* (*definitives X*) = *X*›
  by (*transfer*, *simp add*: *infinites-dprefixes-Infinite*)

lemma *definitives-inverse*: ‹*definitives* (*property X*) = *X*›
proof (*rule dset.order-antisym*)
  show ‹*definitives* (*property X*) ⊑ *X*›
    by (*transfer*, *force simp*: *dprefixes-def infinites-prefix-closure*
                intro: *definitive-elemI*)
next
  show ‹*X* ⊑ *definitives* (*property X*)›
    apply *transfer*
    using *definitive-contains-extensions definitive-infinite-extension*
    by (*force simp*: *dprefixes-def prefix-closure-def infinites-def*)
qed

lemma *definitives-mono*: ‹*A* ⊆ *B* ⟹ *definitives A* ⊑ *definitives B*›
  by (*transfer*, *metis dprefixes-inter-distrib image-mono inf.order-iff le-infE*)

lemma *property-mono*: ‹*A* ⊑ *B* ⟹ *property A* ⊆ *property B*›
  by (*transfer*, *auto simp*: *infinites-def*)

lemma *definitives-reflecting*: ‹*definitives A* ⊑ *definitives B* ⟹ *A* ⊆ *B*›
  using *property-inverse property-mono* **by** *metis*

lemma *completions-reflecting*: ‹*property A* ⊆ *property B* ⟹ *A* ⊑ *B*›
  using *definitives-inverse definitives-mono* **by** *metis*

lemma *property-Inter*: ‹*property* (⨅ *S*) = ⋂ (*property* ‘ *S*)›
  by (*transfer*, *simp add*: *infinites-Inter*)

lemma *property-Union*: ‹*property* (⨆ *S*) = ⋃ (*property* ‘ *S*)›
  by (*transfer*, *simp add*: *dUnion-def infinites-dprefixes infinites-Union*)

interpretation *dset*: *complete-distrib-lattice* ‹⨅› ‹⨆› ‹(⊓)› ‹(⊑)› ‹(⊏)› ‹(⊔)› ‹∅› ‹Σ∞›
  by (*unfold-locales*)
    (*auto intro*: *completions-reflecting simp add*: *property-Inter property-Union INF-SUP-set*)

definition *iprepend* :: ‹*'a infinite-trace set* ⇒ *'a infinite-trace set*› **where**
  ‹*iprepend X* = {*t*. *itdrop 1 t* ∈ *X* }›

lemma *iprepend-itdrop*: ‹*itdrop k x* ∈ *iprepend B* ⟷ *itdrop* (*Suc k*) *x* ∈ *B*›
  by (*simp add*: *iprepend-def*)

lemmas *iprepend-itdrop-0*[*simp*] = *iprepend-itdrop*[**where** k = ‹*0*›,*simplified*]

definition *prepend'* :: ‹*'a trace set* ⇒ *'a trace set*› **where**
‹*prepend' X* = {*t*. *tdrop 1 t* ∈ *X* }›

lemma *trace-uncons-cases* [*case-names Cons Nil*]:
assumes ‹⋀σ *t*. *x* = *singleton* σ ⌢ *t* ⟹ *P*›
and ‹*x* = ε ⟹ *P*›
shows ‹*P*›
proof (*cases* ‹*x*›)
case (*Finite xs*)
then show ‹*?thesis*›
by (*cases* ‹*xs*›;
force simp: *assms*(*2*)[*simplified* ε-*def*]
intro: *assms*(*1*)[**where** *t* = ‹*Finite ts*› **for** *ts*,
simplified singleton-def append.simps List.append.simps])
next
case (*Infinite f*) **note** *A* = *this*
have ‹*f* = (λ*n*. *if n* = *0 then* [*f 0*] ! *n else* (*f* ∘ *Suc*) (*n* − *length* [*f 0*]))›
by (*rule ext*, *simp*)
with *A* **show** ‹*?thesis*›
using *assms*(*1*)[**where** σ = ‹*f 0*› **and** *t* = ‹*Infinite* (*f* ∘ *Suc*)›,
simplified singleton-def append.simps, *simplified*]
by *simp*
qed

lemma *append-prefixes-left*: ‹*a* ∈ ↓ *b* ⟹ *c* ⌢ *a* ∈ ↓ *c* ⌢ *b*›
by (*simp add*: *prefixes-def*) (*metis trace.assoc*)

lemma *tdrop-singleton-append*[*simp*]: ‹*tdrop* (*Suc n*) (*singleton* σ ⌢ *t*) = *tdrop n t*›
by (*cases* ‹*t*›, *simp-all add*: *singleton-def itdrop-def*)
lemma *tdrop-zero*[*simp*]: ‹*tdrop 0 t* = *t*›
by (*cases* ‹*t*›; *simp*)
lemma *tdrop-ε*[*simp*]: ‹*tdrop k* ε = ε›
by (*simp add*: ε-*def*)

lemma *prepend'-prefix-closure*: ‹↓*s* (*prepend' X*) ⊆ *prepend'* (↓*s X*)›
proof (*rule subsetI*)
fix *x*
assume *A*: ‹ *x* ∈ ↓*s prepend' X*›
show ‹*x* ∈ *prepend'* (↓*s X*) ›
proof (*cases* ‹*x*› *rule*: *trace-uncons-cases*)
case (*Cons* σ *t*)
with *A* **show** ‹*?thesis*›
unfolding *prefix-closure-def prepend'-def prefixes-def*
by (*fastforce simp*: *trace.assoc*)
next
case *Nil*
with *A* **show** ‹*?thesis*›

```
    unfolding prefix-closure-def prepend'-def
    by (force simp: prefixes-empty-least)
  qed
qed

lemma prepend'-dprefixes :
assumes ‹definitive X›
shows ‹↓d prepend' X = prepend' X›
proof
  show ‹↓d prepend' X ⊆ prepend' X›
  proof (rule subsetI)
    fix x assume A: ‹x ∈ ↓d prepend' X› show ‹x ∈ prepend' X›
    proof (cases ‹x› rule: trace-uncons-cases)
      case (Cons σ t)
      with A show ‹?thesis›
        unfolding dprefixes-def
        apply (subst assms[simplified definitive-def, THEN sym])
        apply (clarsimp dest!: subset-trans[OF - prepend'-prefix-closure])
        using append-prefixes-left
        by (force simp: dprefixes-def prepend'-def prefix-closure-def subset-iff
                        prefixes-extensions[THEN sym])
    next
      case Nil
      with A show ‹?thesis›
        apply (subst assms[simplified definitive-def, THEN sym])
        apply (clarsimp simp: prefixes-empty-least prefixes-def dprefixes-def
                              prepend'-def prefix-closure-def subset-iff
                              prefixes-extensions[THEN sym])
        by (metis tdrop-singleton-append tdrop-zero trace.assoc)
    qed
  qed
next
  show ‹prepend' X ⊆ ↓d prepend' X›
  proof (rule subsetI)
    fix x assume A: ‹x ∈ prepend' X› show ‹x ∈ ↓d prepend' X›
    proof (cases ‹x› rule: trace-uncons-cases)
      case (Cons σ t)
      with A show ‹?thesis›
        by (clarsimp simp: dprefixes-def prefixes-def prepend'-def
                           prefix-closure-def prefixes-extensions[THEN sym])
           (metis (mono-tags, lifting) assms definitive-contains-extensions
                  mem-Collect-eq prefixes-def prefixes-extensions subset-eq
                  tdrop-singleton-append tdrop-zero trace.assoc)
    next
      case Nil
      with A show ‹?thesis›
        using assms definitive-contains-extensions
        by (force simp: dprefixes-def prepend'-def prefix-closure-def)
    qed
```

qed
qed

lemma *prepend′-definitive* :
assumes ‹*definitive X*›
shows ‹*definitive* (*prepend′ X*)›
unfolding *definitive-def* **using** *assms*
by (*rule prepend′-dprefixes*)

lift-definition *prepend* :: ‹*′a dset* ⇒ *′a dset*› **is** ‹*prepend′*›
by (*rule prepend′-definitive*)

lemma *prepend-Inter*: ‹⋂ (*prepend ‘ S*) = *prepend* (⋂ *S*)›
apply *transfer*
by (*auto simp add*: *prepend′-def*)

lemma *in-dset-prependD*: ‹*in-dset* (*Finite* [*a*] ⌢ *x*) (*prepend A*) ⟹ *in-dset x A*›
by (*transfer*, *metis One-nat-def Traces.singleton-def mem-Collect-eq prepend′-def*
tdrop-singleton-append tdrop-zero)

lemma *in-dset-prependI*: ‹*in-dset x A* ⟹ *in-dset* (*Finite* [*a*] ⌢ *x*) (*prepend A*)›
by (*transfer*, *metis One-nat-def Traces.singleton-def mem-Collect-eq prepend′-def*
tdrop-singleton-append tdrop-zero)

lemma *prepend′-mono*:
assumes ‹*A* ⊆ *B*›
shows ‹*prepend′ A* ⊆ *prepend′ B*›
using *assms* **unfolding** *prepend′-def*
by *blast*

lemma *property-prepend*: ‹*property* (*prepend X*) = *iprepend* (*property X*)›
apply *transfer*
by (*clarsimp simp*: *definitive-def infinites-def prepend′-def*
split!: *trace.split-asm trace.split intro*!: *set-eqI*;
blast)

lemma *iprepend-Union*: ‹⋃ (*iprepend ‘ S*) = *iprepend* (⋃ *S*)›
by *fastforce*

lemma *definitives-inverse-eqI*: ‹*definitives* (*property X*) = *definitives* (*property Y*) ⟹ *X* = *Y*›
by (*simp add*: *definitives-inverse*)

lemma *prepend-Union*: ‹⨆ (*prepend ‘ S*) = *prepend* (⨆ *S*)›
apply (*rule definitives-inverse-eqI*)
apply (*simp add*: *property-Union property-prepend*)
by (*metis UN-extend-simps*(*10*) *iprepend-Union*)

lemma *non-empty-trace*: ‹ *x* ≠ *ε* ⟷ (∃ *σ x′*. *x* = *Finite* [*σ*] ⌢ *x′*)›
apply (*cases ‹x›* *rule*: *trace-uncons-cases*; *clarsimp*)

apply (*metis Traces.singleton-def ε-def append-is-empty(1) not-Cons-self2 trace.inject(1)*)
by (*metis ε-def append-is-empty(1) list.discI trace.inject(1)*)

lemma *thead-append*: ‹$x \neq \varepsilon \Longrightarrow$ *thead* $(x \frown y) =$ *thead* x›
by (*cases* ‹x›; *cases* ‹y›; *simp add*: *ε-def nth-append*)

lemma *thead-prefix*: ‹ $x \in \downarrow y \Longrightarrow x \neq \varepsilon \Longrightarrow$ *thead* $x =$ *thead* y›
apply (*simp add*: *prefixes-def non-empty-trace*)
using *thead-append* [**where** $x =$ ‹*Finite* [-]›, *simplified ε-def*, *simplified*]
by (*metis append-is-empty(1) thead-append*)

lemma *compr′-inter-thead*:
‹$\downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x)\} \cap \downarrow_d \{x.\ \ x \neq \varepsilon \wedge Q\ (\mathit{thead}\ x)\}$
$= \downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge Q\ (\mathit{thead}\ x)\}$›
proof (*rule antisym*)
{ fix $x\ t$
assume ‹$\forall\, t.\ x \in \downarrow t \longrightarrow (\exists\, x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge t \in \downarrow x)$›
and ‹$\forall\, t.\ x \in \downarrow t \longrightarrow (\exists\, x.\ x \neq \varepsilon \wedge Q\ (\mathit{thead}\ x) \wedge t \in \downarrow x)$›
and ‹$x \in \downarrow t$›
then have ‹ $\exists\, x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge Q\ (\mathit{thead}\ x) \wedge t \in \downarrow x$›
by (*cases* ‹$t = \varepsilon$›; *fastforce dest*: *thead-prefix simp*: *prefixes-empty prefixes-empty-least*)
} then show ‹$\downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x)\} \cap \downarrow_d \{x.\ \ x \neq \varepsilon \wedge Q\ (\mathit{thead}\ x)\} \subseteq \downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge Q\ (\mathit{thead}\ x)\}$›
by (*clarsimp simp*: *set-eq-iff subset-iff dprefixes-def prefix-closure-def prefixes-extensions*[*THEN sym*])
next
{ fix x
assume ‹ $\forall\, t.\ x \in \downarrow t \longrightarrow (\exists\, x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge Q\ (\mathit{thead}\ x) \wedge t \in \downarrow x)$›
then have ‹$(\forall\, t.\ x \in \downarrow t \longrightarrow (\exists\, x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge t \in \downarrow x)) \wedge$
$(\forall\, t.\ x \in \downarrow t \longrightarrow (\exists\, x.\ x \neq \varepsilon \wedge Q\ (\mathit{thead}\ x) \wedge t \in \downarrow x))$›
by *fastforce* **}**
then show ‹$\downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x)\} \cap \downarrow_d \{x.\ \ x \neq \varepsilon \wedge Q\ (\mathit{thead}\ x)\} \supseteq \downarrow_d \{x.\ x \neq \varepsilon \wedge P\ (\mathit{thead}\ x) \wedge Q\ (\mathit{thead}\ x)\}$›
by (*clarsimp simp*: *set-eq-iff subset-iff dprefixes-def prefix-closure-def prefixes-extensions*[*THEN sym*])
qed

lift-definition *compr* :: ‹(*'a trace* ⇒ *bool*) ⇒ *'a dset*› **is** ‹$\lambda p.\ \downarrow_d \{x.\ p\ x\ \}$›
by (*rule definitive-dprefixes*)

lift-definition *complement* :: ‹*'a dset* ⇒ *'a dset*› **is** ‹$\lambda p.\ \downarrow_d$ (*range Infinite* − *p*)›
by (*rule definitive-dprefixes*)

lemma *property-complement*[*simp*]: ‹*property* (*complement X*) = *UNIV* − *property X*›
by (*transfer*, *force simp*: *infinites-dprefixes*[*simplified infinites-def*] *infinites-def*
split: *trace.split-asm trace.split*)

end
theory *LinearTemporalLogic*
imports *Traces AnswerIndexedFamilies Main*
begin

3 Linear-time Temporal Logic

datatype (*atoms-ltl*: *'a*) *ltl* =
  True-ltl (‹$true_l$›)
| *Not-ltl* ‹*'a ltl*› (‹not_l -› [*85*] *85*)
| *Prop-ltl* ‹*'a*› (‹$prop_l$ *'*(-*')*›)
| *Or-ltl* ‹*'a ltl*› ‹*'a ltl*› (‹- or_l -› [*82*,*82*] *81*)
| *And-ltl* ‹*'a ltl*› ‹*'a ltl*› (‹- and_l - › [*82*,*82*] *81*)
| *Next-ltl* ‹*'a ltl*› (‹X_l -› [*88*] *87*)
| *Until-ltl* ‹*'a ltl*› ‹*'a ltl*› (‹- U_l -› [*84*,*84*] *83*)

fun *lsatisfying-AiF* :: ‹*'a* ⇒ *'a state infinite-trace AiF*› (‹*lsat*•›) **where**
‹*lsat*• *x T* = {*t*. *x* ∈ *L* (*t* *0*)}› |
‹*lsat*• *x F* = {*t*. *x* ∉ *L* (*t* *0*)}›

fun *x-operator* :: ‹*'a infinite-trace AiF* ⇒ *'a infinite-trace AiF*› (‹*X*▪›) **where**
‹*X*▪ *t T* = {*x* | *x*. *itdrop* *1* *x* ∈ (*t T*)}›|
‹*X*▪ *t F* = {*x* | *x*. *itdrop* *1* *x* ∈ (*t F*)}›

fun *u-operator* :: ‹*'a infinite-trace AiF* ⇒ *'a infinite-trace AiF* ⇒ *'a infinite-trace AiF*› (**infix** ‹*U*▪› *61*) **where**
‹(*a U*▪ *b*) *T* = {*x* | *x*. ∃*k*. (∀ *i*<*k*. *itdrop* *i* *x* ∈ (*a T*)) ∧ *itdrop* *k* *x* ∈ (*b T*)}›|
‹(*a U*▪ *b*) *F* = {*x* | *x*. ∀*k*. (∃ *i*<*k*. *itdrop* *i* *x* ∈ (*a F*)) ∨ *itdrop* *k* *x* ∈ (*b F*)}›

fun *ltl-semantics* :: ‹*'a ltl* ⇒ *'a state infinite-trace AiF*› (‹⟦-⟧$_l$›) **where**
‹⟦ $true_l$ ⟧$_l$ = *T*•›|
‹⟦ not_l φ ⟧$_l$ = ¬▪ ⟦ φ ⟧$_l$›|
‹⟦ $prop_l$(*a*) ⟧$_l$ = *lsat*• *a*›|
‹⟦ φ or_l ψ ⟧$_l$ = ⟦ φ ⟧$_l$ ∨▪ ⟦ ψ ⟧$_l$›|
‹⟦ φ and_l ψ ⟧$_l$ = ⟦ φ ⟧$_l$ ∧▪ ⟦ ψ ⟧$_l$›|
‹⟦ X_l φ ⟧$_l$ = *X*▪ ⟦ φ ⟧$_l$›|
‹⟦ φ U_l ψ ⟧$_l$ = ⟦ φ ⟧$_l$ *U*▪ ⟦ ψ ⟧$_l$›

lemma *excluded-middle-ltl'* :
  shows ‹(Γ ∉ ⟦φ⟧$_l$ *T*) ⟷ (Γ ∈ ⟦φ⟧$_l$ *F*)›
  and ‹(Γ ∉ ⟦φ⟧$_l$ *F*) ⟷ (Γ ∈ ⟦φ⟧$_l$ *T*)›
proof (*induct* ‹φ› *arbitrary*: ‹Γ›)
qed *auto*

lemma *excluded-middle-ltl*: ‹Γ ∈ ⟦φ⟧$_l$ *T* ∨ Γ ∈ ⟦φ⟧$_l$ *F*›
  using *excluded-middle-ltl'* **by** *blast*

definition *false-ltl* (‹$false_l$›) **where**
  false-ltl-def[*simp*]: ‹$false_l$ = not_l $true_l$›

definition *implies-ltl* :: ‹'*a ltl* ⇒ '*a ltl* ⇒ '*a ltl*› (**infix** ‹*implies*$_l$› *80*) **where**
implies-ltl-def[*simp*]: ‹φ *implies*$_l$ ψ = (*not*$_l$ φ *or*$_l$ ψ)›

definition *final-ltl* :: ‹'*a ltl* ⇒ '*a ltl*› (‹F_l -›) **where**
final-ltl-def[*simp*]: ‹(F_l φ) = (*true*$_l$ U_l φ)›

definition *global-ltl* :: ‹'*a ltl* ⇒ '*a ltl*› (‹G_l -›) **where**
global-ltl-def[*simp*]: ‹(G_l φ) = (*not*$_l$ F_l (*not*$_l$ φ))›

3.1 Linear temporal logic equivalence

fun *ltl-semantics'* :: ‹'*a state infinite-trace* ⇒ '*a ltl* ⇒ *bool*› (**infix** ‹$\models_l$› *60*) **where**
‹$\Gamma \models_l$ *true*$_l$ = *True*›
| ‹$\Gamma \models_l$ *not*$_l$ φ = ($\neg$ $\Gamma \models_l \varphi$)›
| ‹$\Gamma \models_l$ *prop*$_l$(a) = ($a \in L\ (\Gamma\ 0)$)›
| ‹$\Gamma \models_l$ φ *or*$_l$ ψ = ($\Gamma \models_l \varphi \vee \Gamma \models_l \psi$)›
| ‹$\Gamma \models_l$ φ *and*$_l$ ψ = ($\Gamma \models_l \varphi \wedge \Gamma \models_l \psi$)›
| ‹$\Gamma \models_l$ (X_l φ) = *itdrop 1* $\Gamma \models_l \varphi$›
| ‹$\Gamma \models_l$ (φ U_l ψ) = ($\exists k.\ (\forall i{<}k.$ *itdrop* $i\ \Gamma \models_l \varphi) \wedge$ *itdrop* $k\ \Gamma \models_l \psi$)›

3.2 Linear temporal logic lemmas

lemma ‹$[\![F_l\ (F_l\ \varphi)]\!]_l = [\![F_l\ \varphi]\!]_l$›
proof (*rule AiF-cases*)
show ‹$[\![F_l\ F_l\ \varphi]\!]_l\ T = [\![F_l\ \varphi]\!]_l\ T$›
apply (*clarsimp intro*!: *set-eqI*, *rule*)
apply *auto*[*1*]
by (*clarsimp*, *metis add-0*)
next
show ‹$[\![F_l\ F_l\ \varphi]\!]_l\ F = [\![F_l\ \varphi]\!]_l\ F$›
apply (*clarsimp intro*!: *set-eqI*, *rule*)
apply (*clarsimp*, *metis add-0*)
by *simp*
qed

lemma *ltl-equivalence*:
shows ‹ $\Gamma \models_l \varphi$ = ($\Gamma \in [\![\varphi]\!]_l\ T$)›
and ‹($\neg$ $\Gamma \models_l \varphi$) = ($\Gamma \in [\![\varphi]\!]_l\ F$)›
proof(*induct* ‹φ› *arbitrary*: ‹Γ›)
qed *auto*

end
theory *LTL3*
imports *Main Traces AnswerIndexedFamilies LinearTemporalLogic*
begin

4 LTL3: Semantics, Equivalence and Formula Progression

type-synonym $'a\ AiF_3 = \langle answer \Rightarrow 'a\ state\ dset\rangle$

primrec *and-AiF*$_3$ :: $\langle 'a\ AiF_3 \Rightarrow 'a\ AiF_3 \Rightarrow 'a\ AiF_3\rangle$ (**infixl** $\langle\wedge_3\cdot\rangle$ *60*) **where**
  $\langle(a \wedge_3\cdot\ b)\ T = a\ T \sqcap b\ T\rangle$
| $\langle(a \wedge_3\cdot\ b)\ F = a\ F \sqcup b\ F\rangle$

primrec *or-AiF*$_3$ :: $\langle 'a\ AiF_3 \Rightarrow 'a\ AiF_3 \Rightarrow 'a\ AiF_3\rangle$ (**infixl** $\langle\vee_3\cdot\rangle$ *59*) **where**
  $\langle(a \vee_3\cdot\ b)\ T = a\ T \sqcup b\ T\rangle$
| $\langle(a \vee_3\cdot\ b)\ F = a\ F \sqcap b\ F\rangle$

fun *not-AiF*$_3$ :: $\langle 'a\ AiF_3 \Rightarrow 'a\ AiF_3\rangle$ ($\langle\neg_3\cdot$ -$\rangle$) **where**
  $\langle(\neg_3\cdot\ a)\ T = a\ F\rangle$
| $\langle(\neg_3\cdot\ a)\ F = a\ T\rangle$

fun *univ-AiF*$_3$ :: $\langle 'a\ AiF_3\rangle$ ($\langle T_3\cdot\rangle$) **where**
  $\langle T_3\cdot\ T = \Sigma\infty\rangle$
| $\langle T_3\cdot\ F = \emptyset\rangle$

fun *lsatisfying-AiF*$_3$ :: $\langle 'a \Rightarrow 'a\ AiF_3\rangle$ ($\langle lsat_3\cdot\rangle$) **where**
  $\langle lsat_3\cdot\ x\ T = compr\ (\lambda t.\ t \neq \varepsilon \wedge x \in L\ (thead\ t))\rangle$
| $\langle lsat_3\cdot\ x\ F = compr\ (\lambda t.\ t \neq \varepsilon \wedge x \notin L\ (thead\ t))\rangle$

fun *x*$_3$*-operator* :: $\langle 'a\ AiF_3 \Rightarrow 'a\ AiF_3\rangle$ ($\langle X_3\cdot\rangle$) **where**
  $\langle X_3\cdot\ t\ T = prepend\ (t\ T)\rangle$
| $\langle X_3\cdot\ t\ F = prepend\ (t\ F)\rangle$

fun *iterate* :: $\langle('a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a \Rightarrow 'a)\rangle$ **where**
$\langle iterate\ f\ 0\ x = x\rangle|$
$\langle iterate\ f\ (Suc\ n)\ x = f\ (iterate\ f\ n\ x)\rangle$

primrec *u*$_3$*-operator* :: $\langle 'a\ AiF_3 \Rightarrow 'a\ AiF_3 \Rightarrow 'a\ AiF_3\rangle$ (**infix** $\langle U_3\cdot\rangle$ *61*) **where**
  $\langle(a\ U_3\cdot\ b)\ T = \bigsqcup\ (\ range\ (\lambda i.\ iterate\ (\lambda x.\ prepend\ x \sqcap a\ T)\ i\ (b\ T)\ ))\rangle$
| $\langle(a\ U_3\cdot\ b)\ F = \bigsqcap\ (\ range\ (\lambda i.\ iterate\ (\lambda x.\ prepend\ x \sqcup a\ F)\ i\ (b\ F)\ ))\rangle$

fun *triv-true* :: $\langle 'a \Rightarrow bool\rangle$ **where**
$\langle triv\text{-}true\ x = (\forall s.\ x{\in}\ L\ s)\rangle$

fun *triv-false* :: $\langle 'a \Rightarrow bool\rangle$ **where**
$\langle triv\text{-}false\ x = (\forall s.\ x{\notin}\ L\ s)\rangle$

fun *nontrivial* :: ‹'*a* ⇒ *bool*› **where**
‹*nontrivial x* = ((∃ *s*. *x*∈ *L s*) ∧ (∃ *t*. *x*∉ *L t*))›

fun *zero-length* :: ‹'*a trace* ⇒ *bool*› **where**
‹*zero-length* (*Finite t*) = (*length t* = *0*)›
| ‹*zero-length* (*Infinite t*) = *False*›

fun *ltl-semantics*$_3$ :: ‹'*a ltl* ⇒ '*a AiF*$_3$› (‹⟦-⟧$_3$›) **where**
‹⟦ *true*$_l$ ⟧$_3$ = T_3•›
| ‹⟦ *not*$_l$ *φ* ⟧$_3$ = ¬$_3$▪ ⟦*φ*⟧$_3$›
| ‹⟦ *prop*$_l$(*a*) ⟧$_3$ = *lsat*$_3$• *a*›
| ‹⟦ *φ or*$_l$ *ψ* ⟧$_3$ = ⟦*φ*⟧$_3$ ∨$_3$▪ ⟦*ψ*⟧$_3$›
| ‹⟦ *φ and*$_l$ *ψ* ⟧$_3$ = ⟦*φ*⟧$_3$ ∧$_3$▪ ⟦*ψ*⟧$_3$›
| ‹⟦ *X*$_l$ *φ* ⟧$_3$ = X_3▪ ⟦*φ*⟧$_3$›
| ‹⟦ *φ U*$_l$ *ψ* ⟧$_3$ = ⟦*φ*⟧$_3$ U_3▪ ⟦*ψ*⟧$_3$›

4.1 LTL/LTL3 equivalence

declare *dset.Inf-insert*[*simp del*]
declare *dset.Sup-insert*[*simp del*]

lemma *itdrop-all-split*:
assumes ‹*x* ∈ *A*› **and** ‹∀ *i*<*k*. *itdrop* (*Suc i*) *x* ∈ *A*›
shows ‹*i* < *Suc k* ⟹ *itdrop i x* ∈ *A*›
using *assms* **proof** (*cases* ‹*i*›)
qed (*auto simp*: *itdrop-def*)

lemma *itdrop-exists-split*[*simp*]:
shows ‹(∃ *i*<*Suc k*. *itdrop i x* ∈ *A*) ⟷ (∃ *i* < *k*. *itdrop* (*Suc i*) *x* ∈ *A*) ∨ *x* ∈ *A*›
proof (*rule iffI*)
{ fix *i*
assume ‹*i* < *Suc k*› ‹*itdrop i x* ∈ *A*› ‹*x* ∉ *A*›
then have ‹∃ *i*<*k*. *itdrop* (*Suc i*) *x* ∈ *A*›
proof (*cases* ‹*i*›)
qed *auto*
} then show ‹ ∃ *i*<*Suc k*. *itdrop i x* ∈ *A* ⟹ (∃ *i*<*k*. *itdrop* (*Suc i*) *x* ∈ *A*) ∨ *x* ∈ *A*› **by** *auto*
next
assume ‹(∃ *i*<*k*. *itdrop* (*Suc i*) *x* ∈ *A*) ∨ *x* ∈ *A*›
then show ‹∃ *i*<*Suc k*. *itdrop i x* ∈ *A*›
by *auto*
qed

lemma *until-iterate* :
‹{*x*. ∃ *k*. (∀ *i*<*k*. *itdrop i x* ∈ *A*) ∧ *itdrop k x* ∈ *B*} = ⋃ (*range* (λ*k*. *iterate* (λ*x*. *iprepend x* ∩ *A*) *k B*))›
proof (*rule set-eqI*; *rule iffI*)
fix *x*
{ fix *k*
assume ‹∀ *i*<*k*. *itdrop i x* ∈ *A* › **and** ‹*itdrop k x* ∈ *B*›

```
  then have ‹x ∈ iterate (λx. iprepend x ∩ A) k B›
  proof (induct ‹k› arbitrary: ‹x›)
    case 0
    then show ‹?case› by simp
  next
    case (Suc k)
    from this(2,3) show ‹?case›
      by (auto intro!: Suc.hyps[where x = ‹itdrop 1 x›, simplified])
  qed }
  then show ‹x ∈ {x. ∃k. (∀i<k. itdrop i x ∈ A) ∧ itdrop k x ∈ B}
        ⟹ x ∈ (⋃k. iterate (λx. iprepend x ∩ A) k B)›
    by blast
next
  fix x
{ fix k
  assume ‹x ∈ iterate (λx. iprepend x ∩ A) k B›
  then have ‹(∀i<k. itdrop i x ∈ A) ∧ itdrop k x ∈ B ›
  proof (induct ‹k› arbitrary: ‹x›)
    case 0
    then show ‹?case› by auto
  next
    case (Suc k)
    from this(2) show ‹?case›
      by (auto dest: Suc.hyps[where x = ‹itdrop 1 x›, simplified]
            intro: itdrop-all-split)
  qed }
  then show ‹x ∈ (⋃k. iterate (λx. iprepend x ∩ A) k B) ⟹ x ∈ {x. ∃k. (∀i<k. itdrop i x ∈
A) ∧ itdrop k x ∈ B}›
    by blast
qed

lemma release-iterate:
  ‹ {u. ∀k. (∃i<k. itdrop i u ∈ A) ∨ itdrop k u ∈ B} = ⋂ (range (λi. iterate (λx. iprepend x
∪ A) i B))›
proof (rule set-eqI; rule iffI)
  fix x
{ fix i assume ‹ ∀k. (∃i<k. itdrop i x ∈ A) ∨ itdrop k x ∈ B›
  then have ‹x ∈ iterate (λx. iprepend x ∪ A) i B›
  proof (induct ‹i› arbitrary: ‹x›)
    case 0
    then show ‹?case› by auto
  next
    case (Suc i)
    show ‹?case›
      apply (clarsimp)
      apply (rule Suc.hyps[where x = ‹itdrop 1 x›, simplified])
      using Suc(2)[THEN spec, where x = ‹Suc -›,simplified]
      by auto
  qed }
```

then show ‹$x \in \{u.\ \forall k.\ (\exists i{<}k.\ itdrop\ i\ u \in A) \vee itdrop\ k\ u \in B\} \Longrightarrow x \in (\bigcap i.\ iterate\ (\lambda x.\ iprepend\ x \cup A)\ i\ B)$›
by *auto*
next
fix x
{ fix k
assume ‹$\forall i.\ x \in iterate\ (\lambda x.\ iprepend\ x \cup A)\ i\ B$›
then have P: ‹$\forall i.\ x \in iterate\ (\lambda x.\ iprepend\ x \cup A)\ i\ B$›
by *blast*
assume ‹$itdrop\ k\ x \notin B$› **with** P **have** ‹$\exists i{<}k.\ itdrop\ i\ x \in A$›
proof (*induct* ‹k› *arbitrary*: ‹x›)
case *0*
then show ‹*?case*› **by** (*simp*, *metis iterate.simps*(*1*))
next
case (*Suc k*)
from *this*(*3*) **show** ‹*?case*›
apply *clarsimp*
apply (*rule Suc.hyps*[**where** x = ‹$itdrop\ 1\ x$›, *simplified*])
using *Suc*(*2*)[*THEN spec*, **where** x = ‹*Suc* -›]
by *auto*
qed }
then show ‹$x \in (\bigcap i.\ iterate\ (\lambda x.\ iprepend\ x \cup A)\ i\ B) \Longrightarrow x \in \{u.\ \forall k.\ (\exists i{<}k.\ itdrop\ i\ u \in A) \vee itdrop\ k\ u \in B\}$›
by *auto*
qed

lemma *property-until-iterate*:
‹$property\ (iterate\ (\lambda x.\ prepend\ x \sqcap A)\ k\ B) = iterate\ (\lambda x.\ iprepend\ x \cap property\ A)\ k\ (property\ B)$›
by (*induct* ‹k›, *auto simp*: *property-Inter property-prepend*)

lemma *property-release-iterate*:
‹$property\ (iterate\ (\lambda x.\ prepend\ x \sqcup A)\ k\ B) = iterate\ (\lambda x.\ iprepend\ x \cup property\ A)\ k\ (property\ B)$›
by (*induct* ‹k›, *auto simp*: *property-Union property-prepend*)

lemma *ltl3-equiv-ltl*:
shows ‹$property\ (\llbracket\ \varphi\ \rrbracket_3\ T) = \llbracket\ \varphi\ \rrbracket_l\ T$›
and ‹$property\ (\llbracket\ \varphi\ \rrbracket_3\ F) = \llbracket\ \varphi\ \rrbracket_l\ F$›
proof (*induct* ‹φ›)
case *True-ltl*
{
case *1*
then show ‹*?case*› **by** (*simp*, *transfer*, *simp*)
next
case *2*
then show ‹*?case*› **by** (*simp*, *transfer*, *simp*)
}
next

```
  case (Not-ltl φ)
  {
    case 1
    then show ‹?case› using Not-ltl by simp
  next
    case 2
    then show ‹?case› using Not-ltl by simp
  }
next
  case (Prop-ltl x)
  {
    case 1
    then show ‹?case›
      apply simp
      apply transfer
      apply (simp add: infinites-dprefixes)
      apply (clarsimp simp add: infinites-def  split: trace.split-asm trace.split)
      apply (rule set-eqI, rule iffI)
       apply (clarsimp  split: trace.split-asm trace.split)
       apply (metis zero-length.cases)
       apply (clarsimp split: trace.split-asm trace.split)
      by (metis Traces.append.simps(3) append-is-empty(2) trace.distinct(1) trace.inject(2))
  next
    case 2
    then show ‹?case›
      apply simp
      apply transfer
      apply (simp add: infinites-dprefixes)
      apply (clarsimp simp add: infinites-def  split: trace.split-asm trace.split)
      apply (rule set-eqI, rule iffI)
       apply (clarsimp  split: trace.split-asm trace.split)
       apply (metis zero-length.cases)
       apply (clarsimp split: trace.split-asm trace.split)
      by (metis Traces.append.simps(3) append-is-empty(2) trace.distinct(1) trace.inject(2))
  }
next
  case (Or-ltl φ1 φ2)
  {
    case 1
    then show ‹?case› by (simp add: property-Union Or-ltl)
  next
    case 2
    then show ‹?case› by (simp add: property-Inter Or-ltl)
  }
next
  case (And-ltl φ1 φ2)
  {
    case 1
    then show ‹?case› by (simp add: property-Inter And-ltl)
```

```
  next
    case 2
    then show ‹?case› by (simp add: property-Union And-ltl)
  }
next
  case (Next-ltl φ)
  {
    case 1
    then show ‹?case› by (simp add: property-prepend Next-ltl iprepend-def)
    next
    case 2
    then show ‹?case› by (simp add: property-prepend Next-ltl iprepend-def)
  }
next
  case (Until-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
     apply (simp add: Until-ltl[THEN sym] property-Union image-Collect property-until-iterate)
      using until-iterate[simplified] by blast
  next
    case 2
    then show ‹?case›
     apply (simp add: Until-ltl[THEN sym] property-Inter image-Collect property-release-iterate)
      using release-iterate[simplified] by metis
  }
qed
```

4.2 Equivalence to LTL3 of Bauer et al.

```
lemma extension-lemma: ‹in-dset t A = (∀ω. t ⌢ Infinite ω ∈ Infinite ‘ property A)›
proof transfer
  fix t and A :: ‹'a trace set›
  assume D: ‹definitive A›
  show ‹t ∈ A = (∀ω. t ⌢ Infinite ω ∈ Infinite ‘ infinites A) ›
  proof (rule iffI)
    assume ‹t ∈ A›
    with D have D': ‹↑ t ⊆ A › by (rule definitive-contains-extensions)
    { fix ω have ‹t ⌢ Infinite ω ∈ A›
        by (rule subsetD[OF D'], force simp add: extensions-def)
    } then have ‹ ∀ω. t ⌢ Infinite ω ∈  A› by auto
    thus  ‹ ∀ω. t ⌢ Infinite ω ∈ Infinite ‘ infinites A›
      by (simp add: infinites-append-right infinites-alt)
  next
    assume ‹ ∀ω. t ⌢ Infinite ω ∈ Infinite ‘ infinites A› then
    have inA: ‹ ∀ω. t ⌢ Infinite ω ∈ A›
      by (simp add: infinites-alt infinites-append-right)
    have ‹↑ t ⊆ ↓s A›
    proof −
```

```
    { fix u
      obtain ω :: ‹'a infinite-trace› where ‹Infinite ω = u ⌢ Infinite undefined›
        by (cases ‹u›; simp)
      then have ‹∃ v. (t ⌢ u) ⌢ v ∈ A›
        using inA[THEN spec, where x = ‹ω›] by (metis trace.assoc)
    } thus ‹?thesis› unfolding extensions-def prefix-closure-def prefixes-def by auto
  qed
  with D show ‹t ∈ A› by (rule definitive-elemI)
 qed
qed

lemma extension:
  shows ‹in-dset t (ltl-semantics3 φ T) = (∀ ω. (t ⌢ Infinite ω) ∈ Infinite ' (ltl-semantics φ
T))›
  and   ‹in-dset t (ltl-semantics3 φ F) = (∀ ω. (t ⌢ Infinite ω) ∈ Infinite ' (ltl-semantics φ F))›
  by (simp-all add: ltl3-equiv-ltl[THEN sym] extension-lemma)
```

4.3 Formula Progression

```
fun progress :: ‹'a ltl ⇒ 'a state ⇒ 'a ltl› where
  ‹progress true_l σ = true_l›
| ‹progress (not_l φ) σ = not_l (progress φ) σ›
| ‹progress (prop_l(a)) σ = (if a ∈ L σ then true_l else not_l true_l)›
| ‹progress (φ or_l ψ) σ = (progress φ σ) or_l (progress ψ σ)›
| ‹progress (φ and_l ψ) σ = (progress φ σ) and_l (progress ψ σ)›
| ‹progress (X_l φ) σ = φ›
| ‹progress (φ U_l ψ) σ = (progress ψ σ) or_l ((progress φ σ) and_l (φ U_l ψ))›

lemma unroll-Union: ‹⨆ (range P) = P 0 ⊔ (⨆ (range (P ∘ Suc)))›
  apply (rule definitives-inverse-eqI)
  apply (simp add: property-Union)
  apply (rule dset.order-antisym)
   apply (clarsimp intro!: definitives-mono; metis not0-implies-Suc)
  by (force intro: definitives-mono)

lemma unroll-Inter: ‹⨅ (range P) = P 0 ⊓ (⨅ (range (P ∘ Suc)))›
  apply (rule definitives-inverse-eqI)
  apply (simp add: property-Inter)
  apply (rule dset.order-antisym)
   apply (force intro: definitives-mono)
  by (clarsimp intro!: definitives-mono; metis not0-implies-Suc)

lemma iterates-nonempty: ‹range (λi. iterate f i X) ≠ {}›
  by blast

lemma until-cont: ‹A ≠ {} ⟹ prepend (⨆ A) ⊓ X = ⨆ ((λx. prepend x ⊓ X) ' A)›
  by (simp add: prepend-Union[THEN sym] dset.SUP-inf)

lemma release-cont: ‹A ≠ {} ⟹ prepend (⨅ A) ⊔ X = ⨅ ((λx. prepend x ⊔ X) ' A)›
```

by (*simp add*: *prepend-Inter*[*THEN sym*] *dset.INF-sup*)

lemma *iterate-unroll-Inter*:
assumes ‹$\bigwedge A.\ A \neq \{\} \Longrightarrow f\ (\bigsqcap\ A) = \bigsqcap\ (f\ `\ A)$›
shows ‹$\bigsqcap\ (range\ (\lambda i.\ iterate\ f\ i\ X)) = X \sqcap f\ (\bigsqcap\ (range\ (\lambda i.\ iterate\ f\ i\ X\ )))$›
apply (*subst unroll-Inter*)
by (*force simp*: *assms*[*OF iterates-nonempty*] *property-Inter intro*: *definitives-inverse-eqI*)

lemma *iterate-unroll-Union*:
assumes ‹$\bigwedge A.\ A \neq \{\} \Longrightarrow f\ (\bigsqcup\ A) = \bigsqcup\ (f\ `\ A)$›
shows ‹$\bigsqcup\ (range\ (\lambda i.\ iterate\ f\ i\ X)) = X \sqcup f\ (\bigsqcup\ (range\ (\lambda i.\ iterate\ f\ i\ X\ )))$›
apply (*subst unroll-Union*)
by (*force simp*: *assms*[*OF iterates-nonempty*] *property-Union intro*: *definitives-inverse-eqI*)

lemma *inf-inf*: ‹$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap (x \sqcap z)$›
by (*simp add*: *dset.inf-assoc dset.inf-left-commute*)

theorem *progression-tf* :
‹*prepend* ($[\![progress\ \varphi\ \sigma\ ]\!]_3\ T$) $\sqcap$ *compr* ($\lambda t.\ t \neq \varepsilon \wedge thead\ t = \sigma$) $\sqsubseteq [\![\ \varphi\ ]\!]_3\ T$ ›
‹*prepend* ($[\![progress\ \varphi\ \sigma\ ]\!]_3\ F$) $\sqcap$ *compr* ($\lambda t.\ t \neq \varepsilon \wedge thead\ t = \sigma$) $\sqsubseteq [\![\ \varphi\ ]\!]_3\ F$ ›
proof (*induct* ‹φ›)
case *True-ltl*
{
case *1*
then show ‹*?case*› **by** *simp*
next
case *2*
then show ‹*?case*› **by** (*simp*, *transfer*, *simp add*: *prepend′-def*)
}
next
case (*Not-ltl* φ)
{
case *1*
then show ‹*?case*› **using** *Not-ltl* **by** *simp*
next
case *2*
then show ‹*?case*› **using** *Not-ltl* **by** *simp*
}
next
case (*Prop-ltl x*)
{
case *1*
then show ‹*?case*›
by (*simp*, *transfer*, *auto simp*: *prepend′-def intro*: *dprefixes-mono*[*THEN subsetD*, *rotated*])
next
case *2*

```
    then show ‹?case›
      by (simp, transfer, auto simp: prepend'-def intro: dprefixes-mono[THEN subsetD, rotated])
  }
next
  case (Or-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
      apply (simp add: prepend-Union[THEN sym])
      using Or-ltl(1, 3)
      by (metis (no-types, lifting) dset.inf-sup-distrib2 dset.sup-mono)
  next
    case 2
    then show ‹?case›
      apply (simp add: prepend-Inter[THEN sym])
      using Or-ltl(2,4)
      by (meson dset.dual-order.refl dset.dual-order.trans dset.inf.coboundedI2
            dset.inf-le1 dset.inf-mono)
  }
next
  case (And-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
      apply (simp add: prepend-Inter[THEN sym])
      using And-ltl(1,3)
      by (meson dset.dual-order.trans dset.inf-le1 dset.inf-le2 dset.le-infI)
  next
    case 2
    then show ‹?case›
      apply (simp add: prepend-Union[THEN sym])
      using And-ltl(2, 4)
      by (metis (no-types, lifting) dset.inf-sup-distrib2 dset.sup-mono)
  }
next
  case (Next-ltl φ)
  {
    case 1
    then show ‹?case› by simp
  next
    case 2
    then show ‹?case› by simp
  }
next
  case (Until-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
      apply (simp only: progress.simps)
```

```
    apply (simp add: prepend-Union[THEN sym] prepend-Inter[THEN sym])
     apply (subst dset.inf-commute)
    apply (subst dset.distrib(3))
    apply (rule dset.order-trans)
     apply (rule dset.sup-mono[OF - dset.order-refl])
    apply (subst dset.inf-commute)
     apply (rule Until-ltl(3))
    apply (subst dset.inf-assoc[THEN sym])
    apply (rule dset.order-trans)
     apply (rule dset.sup-mono[OF dset.order-refl])
     apply (rule dset.inf-mono[OF - dset.order-refl])
     apply (subst dset.inf-commute)
     apply (rule Until-ltl(1))
    apply (subst iterate-unroll-Union[OF until-cont], simp)
    by (simp add: dset.inf.commute prepend-Union)
  next
    case 2
    then show ‹?case›
     apply simp
    apply (subst prepend-Inter[THEN sym] prepend-Union[THEN sym], simp)
    apply (subst dset.inf-commute)
    apply (subst inf-inf)
    apply (rule dset.order-trans)
     apply (rule dset.inf-mono)
      apply (subst dset.inf-commute)
      apply (rule Until-ltl(4))
    apply (simp add: prepend-Union[THEN sym])
    apply (subst dset.distrib(3))
     apply (rule dset.sup-mono)
    apply (subst dset.inf-commute)
      apply (rule Until-ltl(2))
     apply (rule dset.le-infI2, rule dset.order-refl)
    apply (subst iterate-unroll-Inter[OF release-cont,simplified]; simp)
    by (metis dset.inf-le2 dset.sup.commute)
  }
qed

theorem progression-tf':
  ‹⟦ φ ⟧3 T ⊓ compr (λt. t ≠ ε ∧ thead t = σ) ⊑ prepend (⟦progress φ σ ⟧3 T) ›
  ‹⟦ φ ⟧3 F ⊓ compr (λt. t ≠ ε ∧ thead t = σ) ⊑ prepend (⟦progress φ σ ⟧3 F) ›
proof (induct ‹φ›)
  case True-ltl
  {
    case 1
    then show ‹?case› by (simp, transfer, simp add: prepend'-def)
  next
    case 2
    then show ‹?case› by simp
  }
```

```
next
  case (Not-ltl φ)
  {
    case 1
    then show ‹?case› using Not-ltl by simp
  next
    case 2
    then show ‹?case› using Not-ltl by simp
  }
next
  case (Prop-ltl x)
  {
    case 1
    then show ‹?case› apply simp
      apply transfer
      apply clarsimp
      apply (clarsimp simp: prepend'-def)
      apply (subst compr'-inter-thead)
      by (metis (mono-tags, lifting) Collect-empty-eq dprefixes-empty)
  next
    case 2
    then show ‹?case›
      apply simp
      apply transfer
      apply (clarsimp simp: prepend'-def)
      apply (subst compr'-inter-thead)
      by (metis (mono-tags, lifting) Collect-empty-eq dprefixes-empty)
  }
next
  case (Or-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
      apply (simp add: prepend-Union[THEN sym])
      using Or-ltl(1,3)
      by (metis (no-types, lifting) dset.inf-sup-distrib2 dset.sup-mono)
  next
    case 2
    then show ‹?case›
      apply (simp add: prepend-Inter[THEN sym])
      using Or-ltl(2,4)
    by (meson dset.dual-order.refl dset.dual-order.trans dset.inf.coboundedI2 dset.inf-le1 dset.inf-mono)
  }
next
  case (And-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
      apply (simp add: prepend-Inter[THEN sym])
```

```
    using And-ltl(1,3)
    by (meson dset.dual-order.refl dset.dual-order.trans dset.le-inf-iff)
  next
    case 2
    then show ‹?case›
    apply (simp add: prepend-Union[THEN sym])
    using And-ltl(2,4)
    by (metis (no-types,lifting) dset.inf-sup-distrib2 dset.sup-mono)
  }
next
  case (Next-ltl φ)
  {
    case 1
    then show ‹?case› using Next-ltl by simp
  next
    case 2
    then show ‹?case› using Next-ltl by simp
  }
next
  case (Until-ltl φ1 φ2)
  {
    case 1
    then show ‹?case›
    unfolding ltl-semantics3.simps u3-operator.simps
              ltl-semantics.simps progress.simps u-operator.simps or-AiF3.simps and-AiF3.simps
      apply (simp add: full-SetCompr-eq prepend-Union[THEN sym])
      apply (rule dset.order-trans[rotated])
       apply (rule dset.sup-mono [OF - dset.order-refl], rule Until-ltl(3))
      apply (simp add: prepend-Inter[THEN sym])
      apply (rule dset.order-trans[rotated])
       apply (rule dset.sup-mono [OF dset.order-refl])
       apply (rule dset.inf-mono [OF - dset.order-refl])
     apply (rule Until-ltl(1))
    apply (subst iterate-unroll-Union[OF until-cont], simp)
    apply (subst dset.inf-commute)
    apply (subst dset.inf-sup-distrib1)
    apply (simp, rule conjI)
     apply (subst dset.inf-commute)
     apply auto[1]
    by (meson dset.eq-refl dset.inf.boundedI dset.le-infE dset.le-supI2)
  next
    case 2
    then show ‹?case›
    unfolding ltl-semantics3.simps u3-operator.simps
              ltl-semantics.simps progress.simps u-operator.simps or-AiF3.simps and-AiF3.simps
      apply (simp add: full-SetCompr-eq prepend-Inter[THEN sym])
      apply (rule conjI,rule dset.order-trans[rotated])
        apply (rule Until-ltl(4))
       apply (rule dset.inf-mono; simp?)
```

```
      apply (metis iterate.simps(1) dset.Inf-lower rangeI)
     apply (simp add: prepend-Union[THEN sym])
     apply (rule dset.order-trans[rotated])
      apply (rule dset.sup-mono)
       apply (rule Until-ltl(2))
    apply (rule dset.order-refl)
   apply (subst iterate-unroll-Inter[OF release-cont], simp)
   apply (simp add: prepend-Inter[THEN sym] image-image)
   apply (subst dset.inf-assoc)
   apply (subst dset.sup-inf-distrib2)
   apply (rule dset.le-infI2)
   by (simp add: dset.inf.coboundedI1 insert-commute)
  }
qed

theorem progression-tf'-u:
  shows ‹⟦ φ ⟧3 A ⊓ compr (λt. t ≠ ε ∧ thead t = σ) ⊑ prepend (⟦progress φ σ ⟧3 A)›
  by (cases ‹A›; simp add: progression-tf')

theorem progression-tf-u:
  shows ‹prepend (⟦progress φ σ ⟧3 A) ⊓ compr (λt. t ≠ ε ∧ thead t = σ) ⊑ ⟦ φ ⟧3 A›
  by (cases ‹A›; simp add: progression-tf)

lemma fp-compr-helper: ‹in-dset (Finite (a # t)) (compr (λ x. x ≠ ε ∧ thead x = a))›
  apply transfer
  apply (simp add: dprefixes-def subset-iff extensions-def prefix-closure-def prefixes-def)
  by (metis ε-def list.distinct(1) nth-Cons-0 thead.simps(1) thead-append trace.inject(1)
        trace.left-neutral trace.right-neutral)


theorem fp:
shows ‹in-dset (Finite t) (⟦ φ ⟧3 A) ⟷ ⟦ foldl progress φ t ⟧3 A = Σ∞ ›
proof (induct ‹t› arbitrary: ‹φ›)
  case Nil
  then show ‹?case›
    by (rule iffI; simp add: in-dset-ε[simplified ε-def] in-dset-UNIV)
next
  case (Cons a t)
  show ‹?case›
  proof (simp add: Cons[where φ=‹progress φ a›, THEN sym], rule)
    assume ‹in-dset (Finite (a # t)) (⟦φ⟧3 A)›
    then show ‹in-dset (Finite t) (⟦progress φ a⟧3 A)›
      by (force intro: in-dset-prependD in-dset-subset[OF progression-tf'-u]
                       in-dset-inter fp-compr-helper)
  next
    assume ‹in-dset (Finite t) (⟦progress φ a⟧3 A)›
    then show ‹in-dset (Finite (a # t)) (⟦φ⟧3 A)›
      by (force intro: in-dset-subset[OF progression-tf-u] in-dset-inter fp-compr-helper
                       in-dset-prependI[where x = ‹Finite u› for u, simplified])
```

qed
qed

lemma *em-ltl*: ‹⟦ φ ⟧$_l$ T = *UNIV* − (⟦ φ ⟧$_l$ F)›
by (*rule set-eqI*; *clarsimp simp add*: *subset-iff ltl-equivalence*[*THEN sym*])

theorem *em*:
shows ‹⟦ φ ⟧$_3$ T = *complement* (⟦ φ ⟧$_3$ F)›
by (*force intro*: *definitives-inverse-eqI simp*: *ltl3-equiv-ltl em-ltl*)

end

Bibliography

[1] R. Amjad, R. van Glabbeek, and L. O'Connor. Semantics for linear-time temporal logic with finite observations. In *Proceedings of the Combined 31st International Workshop on Expressiveness in Concurrency and 21st Workshop on Structural Operational Semantics, EXPRESS/SOS 2024*, EPTCS, 2024. To appear.

[2] F. Bacchus and F. Kabanza. *Using Temporal Logic to Control Search in a Forward Chaining Planner*, page 141153. IOS Press, 1996.

[3] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering Methodology*, 20(4), sep 2011.

[4] F. Kabanza and S. Thiébaux. Search control in planning for temporally extended goals. In *International Conference on Automated Planning and Scheduling*, pages 130–139. AAAI, 2005.