A Verified Proof Checker for Metric First-Order Temporal Logic

Andrei Herasimau Jonathan Julián Huerta y Munive Leonardo Lima
Martin Raszyk Dmitriy Traytel

April 19, 2024

Abstract

Metric first-order temporal logic (MFOTL) is an expressive formalism for specifying temporal and data-dependent constraints on streams of time-stamped, data-carrying events. Recently, we have developed a monitoring algorithm that not only outputs the satisfaction or violation of MFOTL formulas but also explains its verdicts in the form of proof trees [1, 2]. These explanations serve as certificates, and in this entry we verify the correctness of a certificate checker. The checker is used to certify the output of our new, unverified monitoring tool WhyMon. The formalization contains another unverified, executable implementation of an explanation-producing monitoring algorithm used to exemplify our checker.

Contents

1 Traces and Trace Prefixes

1.1 Infinite Traces

coinductive *ssorted* :: $'a$ :: *linorder stream* ⇒ *bool* **where**
shd s ≤ *shd* (*stl s*) ⟹ *ssorted* (*stl s*) ⟹ *ssorted s*

lemma *ssorted_siterate*[*simp*]: (⋀*n*. *n* ≤ *f n*) ⟹ *ssorted* (*siterate f n*)
by (*coinduction arbitrary*: *n*) *auto*

lemma *ssortedD*: *ssorted s* ⟹ *s* !! *i* ≤ *stl s* !! *i*
by (*induct i arbitrary*: *s*) (*auto elim*: *ssorted.cases*)

lemma *ssorted_sdrop*: *ssorted s* ⟹ *ssorted* (*sdrop i s*)
by (*coinduction arbitrary*: *i s*) (*auto elim*: *ssorted.cases ssortedD*)

lemma *ssorted_monoD*: *ssorted s* ⟹ *i* ≤ *j* ⟹ *s* !! *i* ≤ *s* !! *j*
proof (*induct j* − *i arbitrary*: *j*)
case (*Suc x*)
from *Suc*(*1*)[*of j* − *1*] *Suc*(*2*−*4*) *ssortedD*[*of s j* − *1*]
show *?case* **by** (*cases j*) (*auto simp*: *le_Suc_eq Suc_diff_le*)
qed *simp*

lemma *sorted_stake*: *ssorted s* ⟹ *sorted* (*stake i s*)
by (*induct i arbitrary*: *s*)
(*auto elim*: *ssorted.cases simp*: *in_set_conv_nth*
intro!: *ssorted_monoD*[*of _ 0*, *simplified*, *THEN order_trans*, *OF _ ssortedD*])

lemma *ssorted_monoI*: ∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j* ⟹ *ssorted s*
by (*coinduction arbitrary*: *s*)
(*auto dest*: *spec2*[*of _ Suc _ Suc _*] *spec2*[*of _ 0 Suc 0*])

lemma *ssorted_iff_mono*: *ssorted s* ⟷ (∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j*)
using *ssorted_monoI ssorted_monoD* **by** *metis*

lemma *ssorted_iff_le_Suc*: *ssorted s* ⟷ (∀ *i*. *s* !! *i* ≤ *s* !! *Suc i*)

```
  using mono_iff_le_Suc[of snth s] by (simp add: mono_def ssorted_iff_mono)

definition sincreasing s = (∀ x. ∃ i. x < s !! i)

lemma sincreasingI: (⋀x. ∃ i. x < s !! i) ⟹ sincreasing s
  by (simp add: sincreasing_def)

lemma sincreasing_grD:
  fixes x :: 'a :: semilattice_sup
  assumes sincreasing s
  shows ∃ j>i. x < s !! j
proof −
  let ?A = insert x {s !! n | n. n ≤ i}
  from assms obtain j where ∗: Sup_fin ?A < s !! j
    by (auto simp: sincreasing_def)
  then have x < s !! j
    by (rule order.strict_trans1[rotated]) (auto intro: Sup_fin.coboundedI)
  moreover have i < j
  proof (rule ccontr)
    assume ¬ i < j
    then have s !! j ∈ ?A by (auto simp: not_less)
    then have s !! j ≤ Sup_fin ?A
      by (auto intro: Sup_fin.coboundedI)
    with ∗ show False by simp
  qed
  ultimately show ?thesis by blast
qed

lemma sincreasing_siterate_nat[simp]:
  fixes n :: nat
  assumes (⋀n. n < f n)
  shows sincreasing (siterate f n)
unfolding sincreasing_def proof
  fix x
  show ∃ i. x < siterate f n !! i
  proof (induction x)
    case 0
    have 0 < siterate f n !! 1
      using order.strict_trans1[OF le0 assms] by simp
    then show ?case ..
  next
    case (Suc x)
    then obtain i where x < siterate f n !! i ..
    then have Suc x < siterate f n !! Suc i
      using order.strict_trans1[OF _ assms] by (simp del: snth.simps)
    then show ?case ..
  qed
qed

lemma sincreasing_stl: sincreasing s ⟹ sincreasing (stl s) for s :: 'a :: semilattice_sup stream
  by (auto 0 4 simp: gr0_conv_Suc intro!: sincreasingI dest: sincreasing_grD[of s 0])

definition sfinite s = (∀ i. finite (s !! i))

lemma sfiniteI: (⋀i. finite (s !! i)) ⟹ sfinite s
  by (simp add: sfinite_def)

typedef 'a trace = {s :: ('a set × nat) stream. ssorted (smap snd s) ∧ sincreasing (smap snd s) ∧ sfinite
```

```
(smap fst s)}
  by (intro exI[of _ smap (λi. ({}, i)) nats])
    (auto simp: stream.map_comp stream.map_ident sfinite_def cong: stream.map_cong)

setup_lifting type_definition_trace

lift_definition Γ :: 'a trace ⇒ nat ⇒ 'a set is
  λs i. fst (s !! i) .
lift_definition τ :: 'a trace ⇒ nat ⇒ nat is
  λs i. snd (s !! i) .

lemma stream_eq_iff: s = s' ⟷ (∀ n. s !! n = s' !! n)
  by (metis stream.map_cong0 stream_smap_nats)

lemma trace_eqI: (⋀i. Γ σ i = Γ σ' i) ⟹ (⋀i. τ σ i = τ σ' i) ⟹ σ = σ'
  by transfer (auto simp: stream_eq_iff intro!: prod_eqI)

lemma τ_mono[simp]: i ≤ j ⟹ τ s i ≤ τ s j
  by transfer (auto simp: ssorted_iff_mono)

lemma ex_le_τ: ∃ j≥i. x ≤ τ s j
  by (transfer fixing: i x) (auto dest!: sincreasing_grD[of _ i x] less_imp_le)

lemma le_τ_less: τ σ i ≤ τ σ j ⟹ j < i ⟹ τ σ i = τ σ j
  by (simp add: antisym)

lemma less_τD: τ σ i < τ σ j ⟹ i < j
  by (meson τ_mono less_le_not_le not_le_imp_less)

abbreviation Δ s i ≡ τ s i − τ s (i − 1)
```

1.2 Finite Trace Prefixes

```
typedef 'a prefix = {p :: ('a set × nat) list. sorted (map snd p)}
  by (auto intro!: exI[of _ []])

setup_lifting type_definition_prefix

lift_definition pmap_Γ :: ('a set ⇒ 'b set) ⇒ 'a prefix ⇒ 'b prefix is
  λf. map (λ(x, i). (f x, i))
  by (simp add: split_beta comp_def)

lift_definition last_ts :: 'a prefix ⇒ nat is
  λp. (case p of [] ⇒ 0 | _ ⇒ snd (last p)) .

lift_definition first_ts :: nat ⇒ 'a prefix ⇒ nat is
  λn p. (case p of [] ⇒ n | _ ⇒ snd (hd p)) .

lift_definition pnil :: 'a prefix is [] by simp

lift_definition plen :: 'a prefix ⇒ nat is length .

lift_definition psnoc :: 'a prefix ⇒ 'a set × nat ⇒ 'a prefix is
  λp x. if (case p of [] ⇒ 0 | _ ⇒ snd (last p)) ≤ snd x then p @ [x] else []
proof (goal_cases sorted_psnoc)
  case (sorted_psnoc p x)
  then show ?case
    by (induction p) (auto split: if_splits list.splits)
```

qed

instantiation *prefix* :: (*type*) *order* **begin**

lift_definition *less_eq_prefix* :: *'a prefix* ⇒ *'a prefix* ⇒ *bool* **is**
  $\lambda p\ q.\ \exists r.\ q = p\ @\ r$ **.**

definition *less_prefix* :: *'a prefix* ⇒ *'a prefix* ⇒ *bool* **where**
  less_prefix x y = $(x \leq y \wedge \neg\ y \leq x)$

instance
proof (*standard*, *goal_cases less refl trans antisym*)
  case (*less x y*)
  then show *?case* **unfolding** *less_prefix_def* **..**
next
  case (*refl x*)
  then show *?case* **by** *transfer auto*
next
  case (*trans x y z*)
  then show *?case* **by** *transfer auto*
next
  case (*antisym x y*)
  then show *?case* **by** *transfer auto*
qed

end

lemma *psnoc_inject*[*simp*]:
  last_ts p ≤ *snd x* ⟹ *last_ts q* ≤ *snd y* ⟹ *psnoc p x* = *psnoc q y* ⟷ $(p = q \wedge x = y)$
  by *transfer auto*

lift_definition *prefix_of* :: *'a prefix* ⇒ *'a trace* ⇒ *bool* **is** $\lambda p\ s.$ *stake* (*length p*) *s* = *p* **.**

lemma *prefix_of_pnil*[*simp*]: *prefix_of pnil σ*
  by *transfer auto*

lemma *plen_pnil*[*simp*]: *plen pnil* = *0*
  by *transfer auto*

lemma *plen_mono*: $\pi \leq \pi'$ ⟹ *plen* π ≤ *plen* π'
  by *transfer auto*

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc p x*) *s* ⟹ *last_ts p* ≤ *snd x* ⟹
  (*prefix_of p s* ⟹ Γ *s* (*plen p*) = *fst x* ⟹ τ *s* (*plen p*) = *snd x* ⟹ *P*) ⟹ *P*
  by *transfer* (*simp del*: *stake.simps add*: *stake_Suc*)

lemma *le_pnil*[*simp*]: *pnil* ≤ π
  by *transfer auto*

lift_definition *take_prefix* :: *nat* ⇒ *'a trace* ⇒ *'a prefix* **is** *stake*
  by (*auto dest*: *sorted_stake*)

lemma *plen_take_prefix*[*simp*]: *plen* (*take_prefix i σ*) = *i*
  by *transfer auto*

lemma *plen_psnoc*[*simp*]: *last_ts* π ≤ *snd x* ⟹ *plen* (*psnoc* π *x*) = *plen* π + *1*
  by *transfer auto*

lemma *prefix_of_take_prefix*[*simp*]: *prefix_of* (*take_prefix i* σ) σ
by *transfer auto*

lift_definition *pdrop* :: *nat* $\Rightarrow$ '*a prefix* $\Rightarrow$ '*a prefix* **is** *drop*
by (*auto simp*: *drop_map*[*symmetric*] *sorted_wrt_drop*)

lemma *pdrop_0*[*simp*]: *pdrop 0* $\pi = \pi$
by *transfer auto*

lemma *prefix_of_antimono*: $\pi \leq \pi' \Longrightarrow$ *prefix_of* π' *s* $\Longrightarrow$ *prefix_of* π *s*
by *transfer* (*auto simp del*: *stake_add simp add*: *stake_add*[*symmetric*])

lemma *prefix_of_imp_linear*: *prefix_of* π $\sigma \Longrightarrow$ *prefix_of* π' $\sigma \Longrightarrow \pi \leq \pi' \vee \pi' \leq \pi$
proof *transfer*
fix π π' **and** σ :: ('*a set* $\times$ *nat*) *stream*
assume *assms*: *stake* (*length* π) $\sigma = \pi$ *stake* (*length* π') $\sigma = \pi'$
show $(\exists r.\ \pi' = \pi$ @ $r) \vee (\exists r.\ \pi = \pi'$ @ $r)$
proof (*cases length* π *length* π' *rule*: *le_cases*)
case *le*
then have π' = *take* (*length* π) π' @ *drop* (*length* π) π'
by *simp*
moreover have *take* (*length* π) $\pi' = \pi$
using *assms le* **by** (*metis min.absorb1 take_stake*)
ultimately show *?thesis* **by** *auto*
next
case *ge*
then have π = *take* (*length* π') π @ *drop* (*length* π') π
by *simp*
moreover have *take* (*length* π') $\pi = \pi'$
using *assms ge* **by** (*metis min.absorb1 take_stake*)
ultimately show *?thesis* **by** *auto*
qed
qed

lemma τ*_prefix_conv*: *prefix_of p s* $\Longrightarrow$ *prefix_of p s'* $\Longrightarrow$ *i* $<$ *plen p* $\Longrightarrow$ τ *s i* = τ *s' i*
by *transfer* (*simp add*: *stake_nth*[*symmetric*])

lemma Γ*_prefix_conv*: *prefix_of p s* $\Longrightarrow$ *prefix_of p s'* $\Longrightarrow$ *i* $<$ *plen p* $\Longrightarrow$ Γ *s i* = Γ *s' i*
by *transfer* (*simp add*: *stake_nth*[*symmetric*])

lemma *sincreasing_sdrop*:
fixes *s* :: ('*a* :: *semilattice_sup*) *stream*
assumes *sincreasing s*
shows *sincreasing* (*sdrop n s*)
proof (*rule sincreasingI*)
fix *x*
obtain *i* **where** $n < i$ **and** $x <$ *s* !! *i*
using *sincreasing_grD*[*OF assms*] **by** *blast*
then have $x <$ *sdrop n s* !! $(i - n)$
by (*simp add*: *sdrop_snth*)
then show $\exists i.\ x <$ *sdrop n s* !! *i* **..**
qed

lemma *ssorted_shift*:
ssorted (*xs* @− *s*) = (*sorted xs* $\wedge$ *ssorted s* $\wedge$ $(\forall x{\in}$*set xs*. $\forall y{\in}$*sset s*. $x \leq y)$)
proof *safe*
assume $*$: *ssorted* (*xs* @− *s*)
then show *sorted xs*

```
    by (auto simp: ssorted_iff_mono shift_snth sorted_iff_nth_mono split: if_splits)
  from ssorted_sdrop[OF ∗, of length xs] show ssorted s
    by (auto simp: sdrop_shift)
  fix x y assume x ∈ set xs y ∈ sset s
  then obtain i j where i < length xs xs ! i = x s !! j = y
    by (auto simp: set_conv_nth sset_range)
  with ssorted_monoD[OF ∗, of i j + length xs] show x ≤ y by auto
next
  assume sorted xs ssorted s ∀ x∈set xs. ∀ y∈sset s. x ≤ y
  then show ssorted (xs @− s)
  proof (coinduction arbitrary: xs s)
    case (ssorted xs s)
    with ‹ssorted s› show ?case
      by (subst (asm) ssorted.simps) (auto 0 4 simp: neq_Nil_conv shd_sset intro: exI[of _ _ # _])
  qed
qed

lemma sincreasing_shift:
  assumes sincreasing s
  shows sincreasing (xs @− s)
proof (rule sincreasingI)
  fix x
  from assms obtain i where x < s !! i
    unfolding sincreasing_def by blast
  then have x < (xs @− s) !! (length xs + i)
    by simp
  then show ∃ i. x < (xs @− s) !! i ..
qed

lift_definition pts :: 'a prefix ⇒ nat list is map snd .

lemma pts_pmap_Γ[simp]: pts (pmap_Γ f π) = pts π
  by (transfer fixing: f) (simp add: split_beta)
```

1.3 Earliest and Latest Time-Points

```
definition ETP:: 'a trace ⇒ nat ⇒ nat  where
  ETP σ t = (LEAST i. τ σ i ≥ t)

definition LTP:: 'a trace ⇒ nat ⇒ nat where
  LTP σ t = Max {i. (τ σ i) ≤ t}

abbreviation δ σ i j ≡ (τ σ i − τ σ j)

abbreviation ETP_p σ i b ≡ ETP σ ((τ σ i) − b)
abbreviation LTP_p σ i I ≡ min i (LTP σ ((τ σ i) − left I))
abbreviation ETP_f σ i I ≡ max i (ETP σ ((τ σ i) + left I))
abbreviation LTP_f σ i b ≡ LTP σ ((τ σ i) + b)

definition max_opt where
  max_opt a b = (case (a,b) of (Some x, Some y) ⇒ Some (max x y) | _ ⇒ None)

definition LTP_p_safe σ i I = (if τ σ i − left I ≥ τ σ 0 then LTP_p σ i I else 0)

lemma i_ETP_tau: i ≥ ETP σ n ⟷ τ σ i ≥ n
proof
  assume P: i ≥ ETP σ n
  define j where j_def: j ≡ ETP σ n
```

```
  then have i_j: τ σ i ≥ τ σ j using P by auto
  from j_def have τ σ j ≥ n
    unfolding ETP_def using LeastI_ex ex_le_τ by force
  then show τ σ i ≥ n using i_j by auto
next
  assume Q: τ σ i ≥ n
  then show ETP σ n ≤ i unfolding ETP_def
    by (auto simp add: Least_le)
qed

lemma tau_LTP_k:
  assumes τ σ 0 ≤ n LTP σ n < k
  shows τ σ k > n
proof −
  have finite {i. τ σ i ≤ n}
    by (rule ccontr, unfold infinite_nat_iff_unbounded_le mem_Collect_eq)
      (metis Suc_le_eq i_ETP_tau  leD)
  then show ?thesis
    using assms(2) Max.coboundedI linorder_not_less
    unfolding LTP_def by auto
qed

lemma i_LTP_tau:
  assumes n_asm: n ≥ τ σ 0
  shows (i ≤ LTP σ n ⟷ τ σ i ≤ n)
proof
  define A and j where A_def: A ≡ {i. τ σ i ≤ n} and j_def: j ≡ LTP σ n
  assume P: i ≤ LTP σ n
  from n_asm A_def have A_ne: A ≠ {} by auto
  from j_def have i_j: τ σ i ≤ τ σ j using P by auto
  have not_in: k ∉ A if j < k for k
    using n_asm that tau_LTP_k leD
    unfolding A_def j_def by blast
  then have A ⊆ {0..<Suc j}
    using assms not_less_eq
    unfolding A_def j_def
    by fastforce
  then have fin_A: finite A
    using subset_eq_atLeast0_lessThan_finite[of A Suc j]
    by simp
  from A_ne j_def have τ σ j ≤ n
    using Max_in[of A] A_def fin_A
    unfolding LTP_def
    by simp
  then show τ σ i ≤ n using i_j by auto
next
  define A and j where A_def: A ≡ {i. τ σ i ≤ n} and j_def: j ≡ LTP σ n
  assume Q: τ σ i ≤ n
  have not_in: k ∉ A if j < k for k
    using n_asm that tau_LTP_k leD
    unfolding A_def j_def by blast
  then have A ⊆ {0..<Suc j}
    using assms not_less_eq
    unfolding A_def j_def
    by fastforce
  then have fin_A: finite A
    using subset_eq_atLeast0_lessThan_finite[of A Suc j]
    by simp
```

```
  moreover have i ∈ A using Q A_def by auto
  ultimately show i ≤ LTP σ n
    using Max_ge[of A] A_def
    unfolding LTP_def
    by auto
qed

lemma ETP_δ: i ≥ ETP σ (τ σ l + n) ⟹ δ σ i l ≥ n
proof −
  assume P: i ≥ ETP σ (τ σ l + n)
  then have τ σ i ≥ τ σ l + n by (auto simp add: i_ETP_tau)
  then show ?thesis by auto
qed

lemma ETP_ge: ETP σ (τ σ l + n + 1) > l
proof −
  define j where j_def: j ≡ τ σ l + n + 1
  then have etp_j: τ σ (ETP σ j) ≥ j unfolding ETP_def
    using LeastI_ex ex_le_τ by force
  then have τ σ (ETP σ j) > τ σ l using j_def by auto
  then show ?thesis using j_def less_τD by blast
qed

lemma i_le_LTPi: i ≤ LTP σ (τ σ i)
  using τ_mono i_LTP_tau[of σ τ σ i i]
  by auto

lemma i_le_LTPi_add: i ≤ LTP σ (τ σ i + n)
  using i_le_LTPi
  by (simp add: add_increasing2 i_LTP_tau)

lemma i_le_LTPi_minus:
  assumes τ σ 0 + n ≤ τ σ i i > 0 n > 0
  shows LTP σ (τ σ i − n) < i
  unfolding LTP_def
proof (subst Max_less_iff; (intro ballI; elim CollectE)?)
  show finite {j. τ σ j ≤ τ σ i − n}
    unfolding finite_nat_set_iff_bounded_le
  proof (intro exI[of _ i], safe)
    fix j
    assume τ σ j ≤ τ σ i − n
    with assms(1,3) show j ≤ i
      by (metis add_leD2 add_strict_increasing le_add_diff_inverse less_τD less_or_eq_imp_le)
  qed
next
  from assms(1) show {j. τ σ j ≤ τ σ i − n} ≠ {}
    by (auto simp: le_diff_conv2)
next
  fix j
  assume τ σ j ≤ τ σ i − n
  with assms(1,3) show j < i
    by (metis add_leD2 add_strict_increasing le_add_diff_inverse less_τD)
qed

lemma i_ge_etpi: ETP σ (τ σ i) ≤ i
  using i_ETP_tau by auto
```

2 Metric First-Order Temporal Logic

2.1 Syntax

type_synonym $('n, 'a)$ *event* $= ('n \times 'a\ list)$
type_synonym $('n, 'a)$ *database* $= ('n, 'a)$ *event set*
type_synonym $('n, 'a)$ *prefix* $= ('n \times 'a\ list)$ *prefix*
type_synonym $('n, 'a)$ *trace* $= ('n \times 'a\ list)$ *trace*
type_synonym $('n, 'a)$ *env* $= 'n \Rightarrow 'a$
type_synonym $('n, 'a)$ *envset* $= 'n \Rightarrow 'a\ set$

datatype $(fv\_trm: 'n, 'a)$ *trm* $=$ *is_Var*: *Var* $'n$ $(\mathbf{v})$ | *is_Const*: *Const* $'a$ $(\mathbf{c})$

lemma *in_fv_trm_conv*: $x \in fv\_trm\ t \longleftrightarrow t = \mathbf{v}\ x$
by (*cases t*) *auto*

datatype $('n, 'a)$ *formula* $=$
TT $(\top)$
| *FF* $(\bot)$
| *Eq_Const* $'n$ $'a$ $(\_ \approx \_\ [85, 85]\ 85)$
| *Pred* $'n$ $('n, 'a)$ *trm list* $(\_ \dagger \_\ [85, 85]\ 85)$
| *Neg* $('n, 'a)$ *formula* $(\neg_F \_\ [82]\ 82)$
| *Or* $('n, 'a)$ *formula* $('n, 'a)$ *formula* **infixr** $\vee_F$ *80*
| *And* $('n, 'a)$ *formula* $('n, 'a)$ *formula* **infixr** $\wedge_F$ *80*
| *Imp* $('n, 'a)$ *formula* $('n, 'a)$ *formula* **infixr** $\longrightarrow_F$ *79*
| *Iff* $('n, 'a)$ *formula* $('n, 'a)$ *formula* **infixr** $\longleftrightarrow_F$ *79*
| *Exists* $'n$ $('n, 'a)$ *formula* $(\exists_F \_.\ \_\ [70,70]\ 70)$
| *Forall* $'n$ $('n, 'a)$ *formula* $(\forall_F \_.\ \_\ [70,70]\ 70)$
| *Prev* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{Y}\ \_\ \_\ [1000, 65]\ 65)$
| *Next* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{X}\ \_\ \_\ [1000, 65]\ 65)$
| *Once* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{P}\ \_\ \_\ [1000, 65]\ 65)$
| *Historically* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{H}\ \_\ \_\ [1000, 65]\ 65)$
| *Eventually* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{F}\ \_\ \_\ [1000, 65]\ 65)$
| *Always* $\mathcal{I}$ $('n, 'a)$ *formula* $(\mathbf{G}\ \_\ \_\ [1000, 65]\ 65)$
| *Since* $('n, 'a)$ *formula* $\mathcal{I}$ $('n, 'a)$ *formula* $(\_\ \mathbf{S}\ \_\ \_\ [60,1000,60]\ 60)$
| *Until* $('n, 'a)$ *formula* $\mathcal{I}$ $('n, 'a)$ *formula* $(\_\ \mathbf{U}\ \_\ \_\ [60,1000,60]\ 60)$

primrec $fv :: ('n, 'a)$ *formula* $\Rightarrow 'n\ set$ **where**
$fv\ (r \dagger ts) = \bigcup (fv\_trm\ `\ set\ ts)$
| $fv\ \top = \{\}$
| $fv\ \bot = \{\}$
| $fv\ (x \approx c) = \{x\}$
| $fv\ (\neg_F\ \varphi) = fv\ \varphi$
| $fv\ (\varphi \vee_F \psi) = fv\ \varphi \cup fv\ \psi$
| $fv\ (\varphi \wedge_F \psi) = fv\ \varphi \cup fv\ \psi$
| $fv\ (\varphi \longrightarrow_F \psi) = fv\ \varphi \cup fv\ \psi$
| $fv\ (\varphi \longleftrightarrow_F \psi) = fv\ \varphi \cup fv\ \psi$
| $fv\ (\exists_F x.\ \varphi) = fv\ \varphi - \{x\}$
| $fv\ (\forall_F x.\ \varphi) = fv\ \varphi - \{x\}$
| $fv\ (\mathbf{Y}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\mathbf{X}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\mathbf{P}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\mathbf{H}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\mathbf{F}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\mathbf{G}\ I\ \varphi) = fv\ \varphi$
| $fv\ (\varphi\ \mathbf{S}\ I\ \psi) = fv\ \varphi \cup fv\ \psi$
| $fv\ (\varphi\ \mathbf{U}\ I\ \psi) = fv\ \varphi \cup fv\ \psi$

primrec *consts* $:: ('n, 'a)$ *formula* $\Rightarrow 'a\ set$ **where**

consts $(r \dagger ts) = \{\}$ — terms may also contain constants, but these only filter out values from the trace and do not introduce new values of interest (i.e., do not extend the active domain)
| *consts* $\top = \{\}$
| *consts* $\bot = \{\}$
| *consts* $(x \approx c) = \{c\}$
| *consts* $(\neg_F \varphi) =$ *consts* φ
| *consts* $(\varphi \vee_F \psi) =$ *consts* $\varphi \cup$ *consts* ψ
| *consts* $(\varphi \wedge_F \psi) =$ *consts* $\varphi \cup$ *consts* ψ
| *consts* $(\varphi \longrightarrow_F \psi) =$ *consts* $\varphi \cup$ *consts* ψ
| *consts* $(\varphi \longleftrightarrow_F \psi) =$ *consts* $\varphi \cup$ *consts* ψ
| *consts* $(\exists_F x.\ \varphi) =$ *consts* φ
| *consts* $(\forall_F x.\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{Y}\ I\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{X}\ I\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{P}\ I\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{H}\ I\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{F}\ I\ \varphi) =$ *consts* φ
| *consts* $(\mathbf{G}\ I\ \varphi) =$ *consts* φ
| *consts* $(\varphi\ \mathbf{S}\ I\ \psi) =$ *consts* $\varphi \cup$ *consts* ψ
| *consts* $(\varphi\ \mathbf{U}\ I\ \psi) =$ *consts* $\varphi \cup$ *consts* ψ

lemma *finite_fv_trm*[*simp*]: *finite* (*fv_trm t*)
by (*cases t*) *simp_all*

lemma *finite_fv*[*simp*]: *finite* (*fv* φ)
by (*induction* φ) *simp_all*

lemma *finite_consts*[*simp*]: *finite* (*consts* φ)
by (*induction* φ) *simp_all*

definition *nfv* :: ('*n*, '*a*) *formula* ⇒ *nat* **where**
nfv φ = *card* (*fv* φ)

fun *future_bounded* :: ('*n*, '*a*) *formula* ⇒ *bool* **where**
future_bounded $\top$ = *True*
| *future_bounded* $\bot$ = *True*
| *future_bounded* (_ † _) = *True*
| *future_bounded* (_ ≈ _) = *True*
| *future_bounded* $(\neg_F \varphi)$ = *future_bounded* φ
| *future_bounded* $(\varphi \vee_F \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* ψ)
| *future_bounded* $(\varphi \wedge_F \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* ψ)
| *future_bounded* $(\varphi \longrightarrow_F \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* ψ)
| *future_bounded* $(\varphi \longleftrightarrow_F \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* ψ)
| *future_bounded* $(\exists_F \_.\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\forall_F \_.\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\mathbf{Y}\ I\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\mathbf{X}\ I\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\mathbf{P}\ I\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\mathbf{H}\ I\ \varphi)$ = *future_bounded* φ
| *future_bounded* $(\mathbf{F}\ I\ \varphi)$ = (*future_bounded* $\varphi \wedge$ *right* $I \neq \infty$)
| *future_bounded* $(\mathbf{G}\ I\ \varphi)$ = (*future_bounded* $\varphi \wedge$ *right* $I \neq \infty$)
| *future_bounded* $(\varphi\ \mathbf{S}\ I\ \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* ψ)
| *future_bounded* $(\varphi\ \mathbf{U}\ I\ \psi)$ = (*future_bounded* $\varphi \wedge$ *future_bounded* $\psi \wedge$ *right* $I \neq \infty$)

2.2 Semantics

primrec *eval_trm* :: ('*n*, '*a*) *env* ⇒ ('*n*, '*a*) *trm* ⇒ '*a*(_⟦_⟧ [*70*,*89*] *89*) **where**
eval_trm v (**v** *x*) = *v x*

| *eval_trm v* (**c** *x*) = *x*

lemma *eval_trm_fv_cong*: ∀ *x*∈*fv_trm t*. *v x* = *v′ x* ⟹ *v*⟦*t*⟧ = *v′*⟦*t*⟧
by (*induction t*) *simp_all*

definition *eval_trms* :: ('*n*, '*a*) *env* ⇒ ('*n*, '*a*) *trm list* ⇒ '*a list* (_⟦_⟧ [*70*,*89*] *89*) **where**
eval_trms v ts = *map* (*eval_trm v*) *ts*

lemma *eval_trms_fv_cong*:
∀ *t*∈*set ts*. ∀ *x*∈*fv_trm t*. *v x* = *v′ x* ⟹ *v*⟦*ts*⟧ = *v′*⟦*ts*⟧
using *eval_trm_fv_cong*[*of* _ *v v′*]
by (*auto simp*: *eval_trms_def*)

primrec *eval_trm_set* :: ('*n*, '*a*) *envset* ⇒ ('*n*, '*a*) *trm* ⇒ ('*n*, '*a*) *trm* × '*a set*(_{|_|} [*70*,*89*] *89*) **where**
eval_trm_set vs (**v** *x*) = (**v** *x*, *vs x*)
| *eval_trm_set vs* (**c** *x*) = (**c** *x*, {*x*})

definition *eval_trms_set* :: ('*n*, '*a*) *envset* ⇒ ('*n*, '*a*) *trm list* ⇒ (('*n*, '*a*) *trm* × '*a set*) *list* (_{|_|}
[*70*,*89*] *89*)
where *eval_trms_set vs ts* = *map* (*eval_trm_set vs*) *ts*

lemma *eval_trms_set_Nil*: *vs*{|[]|} = []
by (*simp add*: *eval_trms_set_def*)

lemma *eval_trms_set_Cons*:
vs{|(*t* # *ts*)|} = *vs*{|*t*|} # *vs*{|*ts*|}
by (*simp add*: *eval_trms_set_def*)

primrec *sat* :: ('*n*, '*a*) *trace* ⇒ ('*n*, '*a*) *env* ⇒ *nat* ⇒ ('*n*, '*a*) *formula* ⇒ *bool* (⟨_, _, _⟩ ⊨ _ [*56*, *56*,
56, *56*] *55*) **where**
⟨*σ*, *v*, *i*⟩ ⊨ ⊤ = *True*
| ⟨*σ*, *v*, *i*⟩ ⊨ ⊥ = *False*
| ⟨*σ*, *v*, *i*⟩ ⊨ *r* † *ts* = ((*r*, *v*⟦*ts*⟧) ∈ Γ *σ i*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *x* ≊ *c* = (*v x* = *c*)
| ⟨*σ*, *v*, *i*⟩ ⊨ ¬*F φ* = (¬ ⟨*σ*, *v*, *i*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* ∨*F ψ* = (⟨*σ*, *v*, *i*⟩ ⊨ *φ* ∨ ⟨*σ*, *v*, *i*⟩ ⊨ *ψ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* ∧*F ψ* = (⟨*σ*, *v*, *i*⟩ ⊨ *φ* ∧ ⟨*σ*, *v*, *i*⟩ ⊨ *ψ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* ⟶*F ψ* = (⟨*σ*, *v*, *i*⟩ ⊨ *φ* ⟶ ⟨*σ*, *v*, *i*⟩ ⊨ *ψ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* ⟷*F ψ* = (⟨*σ*, *v*, *i*⟩ ⊨ *φ* ⟷ ⟨*σ*, *v*, *i*⟩ ⊨ *ψ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ ∃*F x*. *φ* = (∃ *z*. ⟨*σ*, *v*(*x* := *z*), *i*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ ∀*F x*. *φ* = (∀ *z*. ⟨*σ*, *v*(*x* := *z*), *i*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **Y** *I φ* = (*case i of 0* ⇒ *False* | *Suc j* ⇒ *mem* (*τ σ i* − *τ σ j*) *I* ∧ ⟨*σ*, *v*, *j*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **X** *I φ* = (*mem* (*τ σ* (*Suc i*) − *τ σ i*) *I* ∧ ⟨*σ*, *v*, *Suc i*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **P** *I φ* = (∃ *j*≤*i*. *mem* (*τ σ i* − *τ σ j*) *I* ∧ ⟨*σ*, *v*, *j*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **H** *I φ* = (∀ *j*≤*i*. *mem* (*τ σ i* − *τ σ j*) *I* ⟶ ⟨*σ*, *v*, *j*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **F** *I φ* = (∃ *j*≥*i*. *mem* (*τ σ j* − *τ σ i*) *I* ∧ ⟨*σ*, *v*, *j*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ **G** *I φ* = (∀ *j*≥*i*. *mem* (*τ σ j* − *τ σ i*) *I* ⟶ ⟨*σ*, *v*, *j*⟩ ⊨ *φ*)
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* **S** *I ψ* = (∃ *j*≤*i*. *mem* (*τ σ i* − *τ σ j*) *I* ∧ ⟨*σ*, *v*, *j*⟩ ⊨ *ψ* ∧ (∀ *k*∈{*j*<..*i*}. ⟨*σ*, *v*, *k*⟩ ⊨ *φ*))
| ⟨*σ*, *v*, *i*⟩ ⊨ *φ* **U** *I ψ* = (∃ *j*≥*i*. *mem* (*τ σ j* − *τ σ i*) *I* ∧ ⟨*σ*, *v*, *j*⟩ ⊨ *ψ* ∧ (∀ *k*∈{*i*..<*j*}. ⟨*σ*, *v*, *k*⟩ ⊨ *φ*))

lemma *sat_fv_cong*: ∀ *x*∈*fv φ*. *v x* = *v′ x* ⟹ ⟨*σ*, *v*, *i*⟩ ⊨ *φ* = ⟨*σ*, *v′*, *i*⟩ ⊨ *φ*
proof (*induct φ arbitrary*: *v v′ i*)
case (*Pred n ts*)
thus *?case*
by (*simp cong*: *map_cong eval_trms_fv_cong*[*rule_format*, *OF Pred*[*simplified*, *rule_format*]]
split: *option.splits*)
next

```
  case (Exists t φ)
  then show ?case unfolding sat.simps
    by (intro iff_exI) (simp add: nth_Cons')
next
  case (Forall t φ)
  then show ?case unfolding sat.simps
    by (intro iff_allI) (simp add: nth_Cons')
qed (auto 10 0 simp: Let_def split: nat.splits intro!: iff_exI eval_trm_fv_cong)

lemma sat_Until_rec: ⟨σ, v, i⟩ ⊨ φ U I ψ ⟷
  (mem 0 I ∧ ⟨σ, v, i⟩ ⊨ ψ ∨
   Δ σ (i + 1) ≤ right I ∧ ⟨σ, v, i⟩ ⊨ φ ∧ ⟨σ, v, i + 1⟩ ⊨ φ U (subtract (Δ σ (i + 1)) I) ψ)
  (is ?L ⟷ ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j: i ≤ j mem (τ σ j − τ σ i) I ⟨σ, v, j⟩ ⊨ ψ ∀k ∈ {i ..< j}. ⟨σ, v, k⟩ ⊨ φ
    by auto
  then show ?R
  proof (cases i = j)
    case False
    with j(1,2) have Δ σ (i + 1) ≤ right I
      by (auto elim: order_trans[rotated] simp: diff_le_mono)
    moreover from False j(1,4) have ⟨σ, v, i⟩ ⊨ φ by auto
    moreover from False j have ⟨σ, v, i + 1⟩ ⊨ φ U (subtract (Δ σ (i + 1)) I) ψ
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
    ultimately show ?thesis by blast
  qed simp
next
  assume Δ: Δ σ (i + 1) ≤ right I and now: ⟨σ, v, i⟩ ⊨ φ and
   next: ⟨σ, v, i + 1⟩ ⊨ φ U (subtract (Δ σ (i + 1)) I) ψ
  from next obtain j where j: i + 1 ≤ j mem (τ σ j − τ σ (i + 1)) (subtract (Δ σ (i + 1)) I)
      ⟨σ, v, j⟩ ⊨ ψ ∀k ∈ {i + 1 ..< j}. ⟨σ, v, k⟩ ⊨ φ
    by auto
  from Δ j(1,2) have mem (τ σ j − τ σ i) I
    by (cases right I) (auto simp: le_diff_conv2)
  with now j(1,3,4) show ?L by (auto simp: le_eq_less_or_eq[of i] intro!: exI[of _ j])
qed auto

lemma sat_Since_rec: ⟨σ, v, i⟩ ⊨ φ S I ψ ⟷
  mem 0 I ∧ ⟨σ, v, i⟩ ⊨ ψ ∨
  (i > 0 ∧ Δ σ i ≤ right I ∧ ⟨σ, v, i⟩ ⊨ φ ∧ ⟨σ, v, i − 1⟩ ⊨ φ S (subtract (Δ σ i) I) ψ)
  (is ?L ⟷ ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j: j ≤ i mem (τ σ i − τ σ j) I ⟨σ, v, j⟩ ⊨ ψ ∀k ∈ {j <.. i}. ⟨σ, v, k⟩ ⊨ φ
    by auto
  then show ?R
  proof (cases i = j)
    case False
    with j(1) obtain k where [simp]: i = k + 1
      by (cases i) auto
    with j(1,2) False have Δ σ i ≤ right I
      by (auto elim: order_trans[rotated] simp: diff_le_mono2 le_Suc_eq)
    moreover from False j(1,4) have ⟨σ, v, i⟩ ⊨ φ by auto
    moreover from False j have ⟨σ, v, i − 1⟩ ⊨ φ S (subtract (Δ σ i) I) ψ
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
    ultimately show ?thesis by auto
  qed simp
```

next
  assume *i*: $0 < i$ **and** Δ: $\Delta\ \sigma\ i \leq$ *right I* **and** *now*: $\langle \sigma, v, i \rangle \models \varphi$ **and**
   prev: $\langle \sigma, v, i - 1 \rangle \models \varphi\ \mathbf{S}\ (\mathit{subtract}\ (\Delta\ \sigma\ i)\ I)\ \psi$
  from *prev* **obtain** *j* **where** *j*: $j \leq i - 1$ *mem* $(\tau\ \sigma\ (i - 1) - \tau\ \sigma\ j)$ $((\mathit{subtract}\ (\Delta\ \sigma\ i)\ I))$
      $\langle \sigma, v, j \rangle \models \psi\ \forall k \in \{j <.. i - 1\}.\ \langle \sigma, v, k \rangle \models \varphi$
    by *auto*
  from Δ *i j*(*1*,*2*) **have** *mem* $(\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
    by (*cases right I*) (*auto simp*: *le_diff_conv2*)
  with *now i j*(*1*,*3*,*4*) **show** *?L* **by** (*auto simp*: *le_Suc_eq gr0_conv_Suc intro*!: *exI*[*of* _ *j*])
qed *auto*

lemma *sat_Since_0*: $\langle \sigma, v, 0 \rangle \models \varphi\ \mathbf{S}\ I\ \psi \longleftrightarrow \mathit{mem}\ 0\ I \wedge \langle \sigma, v, 0 \rangle \models \psi$
  by *auto*

lemma *sat_Since_point*: $\langle \sigma, v, i \rangle \models \varphi\ \mathbf{S}\ I\ \psi \Longrightarrow$
    $(\bigwedge j.\ j \leq i \Longrightarrow \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \Longrightarrow \langle \sigma, v, i \rangle \models \varphi\ \mathbf{S}\ (\mathit{point}\ (\tau\ \sigma\ i - \tau\ \sigma\ j))\ \psi \Longrightarrow P) \Longrightarrow P$
  by (*auto intro*: *diff_le_self*)

lemma *sat_Since_pointD*: $\langle \sigma, v, i \rangle \models \varphi\ \mathbf{S}\ (\mathit{point}\ t)\ \psi \Longrightarrow \mathit{mem}\ t\ I \Longrightarrow \langle \sigma, v, i \rangle \models \varphi\ \mathbf{S}\ I\ \psi$
  by *auto*

lemma *sat_Once_Since*: $\langle \sigma, v, i \rangle \models \mathbf{P}\ I\ \varphi = \langle \sigma, v, i \rangle \models \top\ \mathbf{S}\ I\ \varphi$
  by *auto*

lemma *sat_Once_rec*: $\langle \sigma, v, i \rangle \models \mathbf{P}\ I\ \varphi \longleftrightarrow$
  $\mathit{mem}\ 0\ I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$
  $(i > 0 \wedge \Delta\ \sigma\ i \leq \mathit{right}\ I \wedge \langle \sigma, v, i - 1 \rangle \models \mathbf{P}\ (\mathit{subtract}\ (\Delta\ \sigma\ i)\ I)\ \varphi)$
  unfolding *sat_Once_Since*
  by (*subst sat_Since_rec*) *auto*

lemma *sat_Historically_Once*: $\langle \sigma, v, i \rangle \models \mathbf{H}\ I\ \varphi = \langle \sigma, v, i \rangle \models \neg_F\ (\mathbf{P}\ I\ \neg_F\ \varphi)$
  by *auto*

lemma *sat_Historically_rec*: $\langle \sigma, v, i \rangle \models \mathbf{H}\ I\ \varphi \longleftrightarrow$
  $(\mathit{mem}\ 0\ I \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$
  $(i > 0 \longrightarrow \Delta\ \sigma\ i \leq \mathit{right}\ I \longrightarrow \langle \sigma, v, i - 1 \rangle \models \mathbf{H}\ (\mathit{subtract}\ (\Delta\ \sigma\ i)\ I)\ \varphi)$
  unfolding *sat_Historically_Once sat.simps*(*5*)
  by (*subst sat_Once_rec*) *auto*

lemma *sat_Eventually_Until*: $\langle \sigma, v, i \rangle \models \mathbf{F}\ I\ \varphi = \langle \sigma, v, i \rangle \models \top\ \mathbf{U}\ I\ \varphi$
  by *auto*

lemma *sat_Eventually_rec*: $\langle \sigma, v, i \rangle \models \mathbf{F}\ I\ \varphi \longleftrightarrow$
  $\mathit{mem}\ 0\ I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$
  $(\Delta\ \sigma\ (i + 1) \leq \mathit{right}\ I \wedge \langle \sigma, v, i + 1 \rangle \models \mathbf{F}\ (\mathit{subtract}\ (\Delta\ \sigma\ (i + 1))\ I)\ \varphi)$
  unfolding *sat_Eventually_Until*
  by (*subst sat_Until_rec*) *auto*

lemma *sat_Always_Eventually*: $\langle \sigma, v, i \rangle \models \mathbf{G}\ I\ \varphi = \langle \sigma, v, i \rangle \models \neg_F\ (\mathbf{F}\ I\ \neg_F\ \varphi)$
  by *auto*

lemma *sat_Always_rec*: $\langle \sigma, v, i \rangle \models \mathbf{G}\ I\ \varphi \longleftrightarrow$
  $(\mathit{mem}\ 0\ I \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$
  $(\Delta\ \sigma\ (i + 1) \leq \mathit{right}\ I \longrightarrow \langle \sigma, v, i + 1 \rangle \models \mathbf{G}\ (\mathit{subtract}\ (\Delta\ \sigma\ (i + 1))\ I)\ \varphi)$
  unfolding *sat_Always_Eventually sat.simps*(*5*)
  by (*subst sat_Eventually_rec*) *auto*

bundle *MFOTL_no_notation* **begin**

For bold font, type “backslash” followed by the word “bold”

no_notation *Var* (**v**)
  and *Const* (**c**)

For subscripts type “backslash” followed by “sub”

no_notation *TT* (⊤)
  and *FF* (⊥)
  and *Pred* (_ † _ [*85*, *85*] *85*)
  and *Eq_Const* (_ ≈ _ [*85*, *85*] *85*)
  and *Neg* (¬$_F$ _ [*82*] *82*)
  and *And* (**infixr** ∧$_F$ *80*)
  and *Or* (**infixr** ∨$_F$ *80*)
  and *Imp* (**infixr** ⟶$_F$ *79*)
  and *Iff* (**infixr** ⟷$_F$ *79*)
  and *Exists* (∃$_F$_. _ [*70*,*70*] *70*)
  and *Forall* (∀$_F$_. _ [*70*,*70*] *70*)
  and *Prev* (**Y** _ _ [*1000*, *65*] *65*)
  and *Next* (**X** _ _ [*1000*, *65*] *65*)
  and *Once* (**P** _ _ [*1000*, *65*] *65*)
  and *Eventually* (**F** _ _ [*1000*, *65*] *65*)
  and *Historically* (**H** _ _ [*1000*, *65*] *65*)
  and *Always* (**G** _ _ [*1000*, *65*] *65*)
  and *Since* (_ **S** _ _ [*60*,*1000*,*60*] *60*)
  and *Until* (_ **U** _ _ [*60*,*1000*,*60*] *60*)

no_notation *eval_trm* (_⟦_⟧ [*70*,*89*] *89*)
  and *eval_trms* (_⟦⟦_⟧⟧ [*70*,*89*] *89*)
  and *eval_trm_set* (_⦃_⦄ [*70*,*89*] *89*)
  and *eval_trms_set* (_⦃⦃_⦄⦄ [*70*,*89*] *89*)
  and *sat* (⟨_, _, _⟩ ⊨ _ [*56*, *56*, *56*, *56*] *55*)
  and *Interval.interval* ([_,_])

end

bundle *MFOTL_notation* **begin**

notation *Var* (**v**)
  and *Const* (**c**)

notation *TT* (⊤)
  and *FF* (⊥)
  and *Pred* (_ † _ [*85*, *85*] *85*)
  and *Eq_Const* (_ ≈ _ [*85*, *85*] *85*)
  and *Neg* (¬$_F$ _ [*82*] *82*)
  and *And* (**infixr** ∧$_F$ *80*)
  and *Or* (**infixr** ∨$_F$ *80*)
  and *Imp* (**infixr** ⟶$_F$ *79*)
  and *Iff* (**infixr** ⟷$_F$ *79*)
  and *Exists* (∃$_F$_. _ [*70*,*70*] *70*)
  and *Forall* (∀$_F$_. _ [*70*,*70*] *70*)
  and *Prev* (**Y** _ _ [*1000*, *65*] *65*)
  and *Next* (**X** _ _ [*1000*, *65*] *65*)
  and *Once* (**P** _ _ [*1000*, *65*] *65*)
  and *Eventually* (**F** _ _ [*1000*, *65*] *65*)
  and *Historically* (**H** _ _ [*1000*, *65*] *65*)
  and *Always* (**G** _ _ [*1000*, *65*] *65*)
  and *Since* (_ **S** _ _ [*60*,*1000*,*60*] *60*)
  and *Until* (_ **U** _ _ [*60*,*1000*,*60*] *60*)

notation *eval_trm* (_⟦_⟧ [70,89] 89)
  and *eval_trms* (_⟦_⟧ [70,89] 89)
  and *eval_trm_set* (_⦃_⦄ [70,89] 89)
  and *eval_trms_set* (_⦃⦃_⦄⦄ [70,89] 89)
  and *sat* (⟨_, _, _⟩ ⊨ _ [56, 56, 56, 56] 55)
  and *Interval.interval* ([_,_])

end

unbundle *MFOTL_no_notation*

3 Valued Partitions

lemma *part_list_set_eq_aux1*:
  assumes
    ∀ *i*<*length xs*. ∀ *j*<*length xs*. *i* ≠ *j* ⟶ *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {}
    {} ∉ *fst* ‘ *set xs*
  shows *disjoint* (*fst* ‘ *set xs*) ∧ *distinct* (*map fst xs*)
proof −
  from *assms*(*1*) **have** *disjoint* (*fst* ‘ *set xs*)
    by (*metis disjnt_def in_set_conv_nth pairwise_imageI*)
  moreover have *distinct* (*map fst xs*)
    using *assms*
    by (*smt* (*verit*) *distinct_conv_nth image_iff inf.idem*
      length_map nth_map nth_mem)
  ultimately show *?thesis*
    by *blast*
qed

lemma *part_list_set_eq_aux2*:
  assumes
    disjoint (*fst* ‘ *set xs*)
    distinct (*map fst xs*)
    i < *length xs*
    j < *length xs*
    i ≠ *j*
  shows *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {}
proof −
  from *assms* **have** *fst* (*xs* ! *i*) ∈ *fst* ‘ *set xs*
    and *fst* (*xs* ! *j*) ∈ *fst* ‘ *set xs*
    by *auto*
  moreover have *fst* (*xs* ! *i*) ≠ *fst* (*xs* ! *j*)
    using *assms*(*2*−) *nth_eq_iff_index_eq*
    by *fastforce*
  ultimately show *?thesis*
    using *assms*(*1*) **unfolding** *disjoint_def*
    by *blast*
qed

lemma *part_list_eq*:
  (⋃ *X* ∈ *fst* ‘ *set xs*. *X*) = *UNIV*
    ∧ (∀ *i* < *length xs*. ∀ *j* < *length xs*. *i* ≠ *j*
      ⟶ *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {}) ∧ {} ∉ *fst* ‘ *set xs*
  ⟷ *partition_on UNIV* (*set* (*map fst xs*)) ∧ *distinct* (*map fst xs*)
  using *part_list_set_eq_aux1 part_list_set_eq_aux2*
  unfolding *partition_on_def* **by** (*auto 5 0*)

$'d$: domain (such that the union of $'d$ sets form a partition)

typedef ($'d$, $'a$) *part* = {*xs* :: ($'d$ *set* × $'a$) *list*. *partition_on UNIV* (*set* (*map fst xs*)) ∧ *distinct* (*map fst xs*)}
  by (*rule exI*[*of* _ [(*UNIV*, *undefined*)]]) (*auto simp*: *partition_on_def*)

setup_lifting *type_definition_part*

lift_bnf (*no_warn_wits*, *no_warn_transfer*) (*dead* $'d$, *Vals*: $'a$) *part*
  unfolding *part_list_eq*[*symmetric*]
  by (*auto simp*: *image_iff*)

3.1 *size* setup

lift_definition *subs* :: ($'d$, $'a$) *part* ⇒ $'d$ *set list* **is** *map fst* .

lift_definition *Subs* :: ($'d$, $'a$) *part* ⇒ $'d$ *set set* **is** *set o map fst* .

lift_definition *vals* :: ($'d$, $'a$) *part* ⇒ $'a$ *list* **is** *map snd* .

lift_definition *SubsVals* :: ($'d$, $'a$) *part* ⇒ ($'d$ *set* × $'a$) *set* **is** *set* .

lift_definition *subsvals* :: ($'d$, $'a$) *part* ⇒ ($'d$ *set* × $'a$) *list* **is** *id* .

lift_definition *size_part* :: ($'d$ ⇒ *nat*) ⇒ ($'a$ ⇒ *nat*) ⇒ ($'d$, $'a$) *part* ⇒ *nat* **is** λ*f g*. *size_list* (λ(*x*, *y*). *sum f x* + *g y*) .

instantiation *part* :: (*type*, *type*) *size* **begin**

definition *size_part* **where**
size_part_overloaded_def: *size_part* = *Partition.size_part* (λ_. *0*) (λ_. *0*)

instance ..

end

lemma *size_part_overloaded_simps*[*simp*]: *size x* = *size* (*vals x*)
  unfolding *size_part_overloaded_def*
  by *transfer* (*auto simp*: *size_list_conv_sum_list*)

lemma *part_size_o_map*: *inj h* ⟹ *size_part f g* ∘ *map_part h* = *size_part f* (*g* ∘ *h*)
  by (*rule ext*, *transfer*)
    (*auto simp*: *fun_eq_iff map_prod_def o_def split_beta case_prod_beta*[*abs_def*])

setup ‹
BNF_LFP_Size.register_size_global ***type_name*** ‹*part*› ***const_name*** ‹*size_part*›
  @{*thm size_part_overloaded_def*} @{*thms size_part_def size_part_overloaded_simps*}
  @{*thms part_size_o_map*}
›

lemma *is_measure_size_part*[*measure_function*]: *is_measure f* ⟹ *is_measure g* ⟹ *is_measure* (*size_part f g*)
  by (*rule is_measure_trivial*)

lemma *size_part_estimation*[*termination_simp*]: *x* ∈ *Vals xs* ⟹ *y* < *g x* ⟹ *y* < *size_part f g xs*
  by *transfer* (*auto simp*: *termination_simp*)

lemma *size_part_estimation'*[*termination_simp*]: *x* ∈ *Vals xs* ⟹ *y* ≤ *g x* ⟹ *y* ≤ *size_part f g xs*
  by *transfer* (*auto simp*: *termination_simp*)

lemma *size_part_pointwise*[*termination_simp*]: $(\bigwedge x.\ x \in Vals\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow size\_part\ h\ f\ xs \leq size\_part\ h\ g\ xs$
by *transfer* (*force simp*: *image_iff intro*!: *size_list_pointwise*)

3.2 Functions on Valued Partitions

lemma *Vals_code*[*code*]: *Vals x* = *set* (*map snd* (*Rep_part x*))
by (*force simp*: *Vals_def*)

lemma *Vals_transfer*[*transfer_rule*]: *rel_fun* (*pcr_part* (=) (=)) (=) (*set* ∘ *map snd*) *Vals*
by (*force simp*: *Vals_def rel_fun_def pcr_part_def cr_part_def rel_set_eq prod.rel_eq list.rel_eq*)

lemma *set_vals*[*simp*]: *set* (*vals xs*) = *Vals xs*
by *transfer simp*

lift_definition *part_hd* :: ('*d*, '*a*) *part* ⇒ '*a* **is** *snd* ∘ *hd* .

lift_definition *tabulate* :: '*d list* ⇒ ('*d* ⇒ '*n*) ⇒ '*n* ⇒ ('*d*, '*n*) *part* **is**
λds f z. if distinct ds then if set ds = UNIV then map (*λd.* ({*d*}, *f d*)) *ds else* (− *set ds*, *z*) # *map* (*λd.* ({*d*}, *f d*)) *ds else* [(*UNIV*, *z*)]
by (*auto simp*: *o_def distinct_map inj_on_def partition_on_def disjoint_def*)

lift_definition *lookup_part* :: ('*d*, '*a*) *part* ⇒ '*d* ⇒ '*a* **is** *λxs d. snd* (*the* (*find* (*λ*(*D*, _). *d* ∈ *D*) *xs*)) .

lemma *Vals_tabulate*[*simp*]: *Vals* (*tabulate xs f z*) =
(*if distinct xs then if set xs = UNIV then f* ‘ *set xs else* {*z*} ∪ *f* ‘ *set xs else* {*z*})
by *transfer* (*auto simp*: *image_iff*)

lemma *lookup_part_tabulate*[*simp*]: *lookup_part* (*tabulate xs f z*) *x* =
(*if distinct xs* ∧ *x* ∈ *set xs then f x else z*)
by (*transfer fixing*: *x xs f z*)
(*auto simp*: *find_dropWhile dropWhile_eq_Cons_conv map_eq_append_conv split*: *list.splits*)

lemma *part_hd_Vals*[*simp*]: *part_hd part* ∈ *Vals part*
by *transfer* (*auto simp*: *partition_on_def image_iff intro*!: *hd_in_set*)

lemma *lookup_part_Vals*[*simp*]: *lookup_part part d* ∈ *Vals part*
proof (*transfer*, *goal_cases part*)
case (*part xs d*)
then have *unique*: (∀ *i*<*length xs*. ∀ *j*<*length xs*. *i* ≠ *j* ⟶ *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {})
using *part_list_eq*[*of xs*]
by *simp*
from *part* **obtain** *D x* **where** *D*: (*D*, *x*) ∈ *set xs d* ∈ *D*
unfolding *partition_on_def*
by *fastforce*
with *unique* **have** *find* (*λ*(*D*, _). *d* ∈ *D*) *xs* = *Some* (*D*, *x*)
unfolding *set_eq_iff*
by (*auto simp*: *find_Some_iff in_set_conv_nth split_beta*)
with *D* **show** *?case*
by (*force simp*: *image_iff*)
qed

lemma *lookup_part_SubsVals*: ∃ *D*. *d* ∈ *D* ∧ (*D*, *lookup_part part d*) ∈ *SubsVals part*
proof (*transfer*, *goal_cases part*)
case (*part d xs*)
then have *unique*: (∀ *i*<*length xs*. ∀ *j*<*length xs*. *i* ≠ *j* ⟶ *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {})
using *part_list_eq*[*of xs*]

```
    by simp
  from part obtain D x where D: (D, x) ∈ set xs d ∈ D
    unfolding partition_on_def
    by fastforce
  with unique have find (λ(D, _). d ∈ D) xs = Some (D, x)
    unfolding set_eq_iff
    by (auto simp: find_Some_iff in_set_conv_nth split_beta)
  with D show ?case
    by (force simp: image_iff)
qed

lemma lookup_part_from_subvals: (D, e) ∈ set (subsvals part) ⟹ d ∈ D ⟹ lookup_part part d = e
proof (transfer fixing: D e d, goal_cases)
  case (1 part)
  then show ?case
  proof (cases find (λ(D, _). d ∈ D) part)
    case (Some De)
    from 1 show ?thesis
      unfolding partition_on_def set_eq_iff Some using Some unfolding find_Some_iff
      by (fastforce dest!: spec[of _ d] simp: in_set_conv_nth split_beta dest: part_list_set_eq_aux2)
  qed (auto simp: partition_on_def image_iff find_None_iff)
qed

lemma size_lookup_part_estimation[termination_simp]: size (lookup_part part d) < Suc (size_part
(λ_. 0) size part)
  unfolding less_Suc_eq_le
  by (rule size_part_estimation'[OF _ order_refl]) simp

lemma subsvals_part_estimation[termination_simp]: (D, e) ∈ set (subsvals part) ⟹ size e < Suc
(size_part (λ_. 0) size part)
  unfolding less_Suc_eq_le
  by (rule size_part_estimation'[OF _ order_refl], transfer)
    (force simp: image_iff)

lemma size_part_hd_estimation[termination_simp]: size (part_hd part) < Suc (size_part (λ_. 0) size
part)
  unfolding less_Suc_eq_le
  by (rule size_part_estimation'[OF _ order_refl]) simp

lemma size_last_estimation[termination_simp]: xs ≠ [] ⟹ size (last xs) < size_list size xs
  by (induct xs) auto

lift_definition lookup :: ('d, 'a) part ⇒ 'd ⇒ ('d set × 'a) is λxs d. the (find (λ(D, _). d ∈ D) xs) .

lemma snd_lookup[simp]: snd (lookup part d) = lookup_part part d
  by transfer auto

lemma distinct_disjoint_uniq: distinct xs ⟹ disjoint (set xs) ⟹
  i < j ⟹ j < length xs ⟹ d ∈ xs ! i ⟹ d ∈ xs ! j ⟹ False
  unfolding disjoint_def disjoint_iff
  by (metis (no_types, lifting) order.strict_trans min.strict_order_iff nth_eq_iff_index_eq nth_mem)

lemma partition_on_UNIV_find_Some:
  partition_on UNIV (set (map fst part)) ⟹ distinct (map fst part) ⟹
  ∃ y. find (λ(D, _). d ∈ D) part = Some y
  unfolding partition_on_def set_eq_iff
  by (auto simp: find_Some_iff in_set_conv_nth
      Ball_def image_iff Bex_def split_beta dest: distinct_disjoint_uniq dest!: spec[of _ d]
```

intro!: *exI*[**where** *P*=λ*x*. ∃ *y z*. *P x y z* ∧ *Q x y z* **for** *P Q*, *OF exI*, *OF exI*, *OF conjI*])

lemma *fst_lookup*: *d* ∈ *fst* (*lookup part d*)
proof (*transfer fixing*: *d*, *goal_cases*)
  case (*1 part*)
  then obtain *y* **where** *find* (λ(*D*, _). *d* ∈ *D*) *part* = *Some y* **using** *partition_on_UNIV_find_Some*
    by *fastforce*
  then show *?case*
    by (*auto dest*: *find_Some_iff*[*THEN iffD1*])
qed

lemma *lookup_subsvals*: *lookup part d* ∈ *set* (*subsvals part*)
proof (*transfer fixing*: *d*, *goal_cases*)
  case (*1 part*)
  then obtain *y* **where** *find* (λ(*D*, _). *d* ∈ *D*) *part* = *Some y* **using** *partition_on_UNIV_find_Some*
    by *fastforce*
  then show *?case*
    by (*auto simp*: *in_set_conv_nth dest*: *find_Some_iff*[*THEN iffD1*])
qed

lift_definition *trivial_part* :: *'pt* ⇒ (*'d*, *'pt*) *part* **is** λ*pt*. [(*UNIV*, *pt*)]
  by (*simp add*: *partition_on_space*)

lemma *part_hd_trivial*[*simp*]: *part_hd* (*trivial_part pt*) = *pt*
  unfolding *part_hd_def*
  by (*transfer*) *simp*

lemma *SubsVals_trivial*[*simp*]: *SubsVals* (*trivial_part pt*) = {(*UNIV*, *pt*)}
  unfolding *SubsVals_def*
  by (*transfer*) *simp*

4 Partitioned Decision Trees

datatype (*dead 'd*, *leaves*: *'l*, *vars*: *'n*) *pdt* = *Leaf* (*unleaf*: *'l*) | *Node 'n* (*'d*, (*'d*, *'l*, *'n*) *pdt*) *part*

inductive *vars_order* :: *'n list* ⇒ (*'d*, *'l*, *'n*) *pdt* ⇒ *bool* **where**
  vars_order vs (*Leaf* _)
| ∀ *expl* ∈ *Vals part1*. *vars_order vs expl* ⟹ *vars_order* (*x* # *vs*) (*Node x part1*)
| *vars_order vs* (*Node x part1*) ⟹ *x* ≠ *z* ⟹ *vars_order* (*z* # *vs*) (*Node x part1*)

lemma *vars_order_Node*:
  assumes *distinct xs*
  shows *vars_order xs* (*Node x part*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ (∀ *e* ∈ *Vals part*. *vars_order zs e*))
proof (*safe*, *goal_cases LR RL*)
  case *LR*
  then show *?case*
    by (*induct xs Node x part rule*: *vars_order.induct*)
      (*auto 4 3 intro*: *exI*[*of* _ _ # _])
next
  case (*RL ys zs*)
  with *assms* **show** *?case*
    by (*induct ys arbitrary*: *xs*)
      (*auto intro*: *vars_order.intros*)
qed

fun *distinct_paths* **where**
  distinct_paths (*Leaf* _) = *True*
| *distinct_paths* (*Node x part*) = (∀ *e* ∈ *Vals part*. *x* ∉ *vars e* ∧ *distinct_paths e*)

fun *eval_pdt* **where**
  eval_pdt v (*Leaf l*) = *l*
| *eval_pdt v* (*Node x part*) = *eval_pdt v* (*lookup_part part* (*v x*))

lemma *eval_pdt_cong*: ∀ *x* ∈ *vars e*. *v x* = *v*′ *x* ⟹ *eval_pdt v e* = *eval_pdt v*′ *e*
  by (*induct e*) *auto*

lemma *vars_order_vars*: *vars_order vs e* ⟹ *vars e* ⊆ *set vs*
  by (*induct vs e rule*: *vars_order.induct*) *auto*

lemma *vars_order_distinct_paths*: *vars_order vs e* ⟹ *distinct vs* ⟹ *distinct_paths e*
  by (*induct vs e rule*: *vars_order.induct*) (*auto dest*!: *vars_order_vars*)

lemma *eval_pdt_fun_upd*: *vars_order vs e* ⟹ *x* ∉ *set vs* ⟹ *eval_pdt* (*v*(*x* := *d*)) *e* = *eval_pdt v e*
  by (*induct vs e rule*: *vars_order.induct*) *auto*

5 Proof System

unbundle *MFOTL_notation*

context begin

inductive *SAT* **and** *VIO* :: (′*n*, ′*d*) *trace* ⇒ (′*n*, ′*d*) *env* ⇒ *nat* ⇒ (′*n*, ′*d*) *formula* ⇒ *bool* **for** *σ* **where**
  STT: *SAT σ v i TT*
| *VFF*: *VIO σ v i FF*
| *SPred*: (*r*, *eval_trms v ts*) ∈ Γ *σ i* ⟹ *SAT σ v i* (*Pred r ts*)
| *VPred*: (*r*, *eval_trms v ts*) ∉ Γ *σ i* ⟹ *VIO σ v i* (*Pred r ts*)
| *SEq_Const*: *v x* = *c* ⟹ *SAT σ v i* (*Eq_Const x c*)
| *VEq_Const*: *v x* ≠ *c* ⟹ *VIO σ v i* (*Eq_Const x c*)
| *SNeg*: *VIO σ v i φ* ⟹ *SAT σ v i* (*Neg φ*)
| *VNeg*: *SAT σ v i φ* ⟹ *VIO σ v i* (*Neg φ*)
| *SOrL*: *SAT σ v i φ* ⟹ *SAT σ v i* (*Or φ ψ*)
| *SOrR*: *SAT σ v i ψ* ⟹ *SAT σ v i* (*Or φ ψ*)
| *VOr*: *VIO σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Or φ ψ*)
| *SAnd*: *SAT σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *SAT σ v i* (*And φ ψ*)
| *VAndL*: *VIO σ v i φ* ⟹ *VIO σ v i* (*And φ ψ*)
| *VAndR*: *VIO σ v i ψ* ⟹ *VIO σ v i* (*And φ ψ*)
| *SImpL*: *VIO σ v i φ* ⟹ *SAT σ v i* (*Imp φ ψ*)
| *SImpR*: *SAT σ v i ψ* ⟹ *SAT σ v i* (*Imp φ ψ*)
| *VImp*: *SAT σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Imp φ ψ*)
| *SIffSS*: *SAT σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *SAT σ v i* (*Iff φ ψ*)
| *SIffVV*: *VIO σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *SAT σ v i* (*Iff φ ψ*)
| *VIffSV*: *SAT σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Iff φ ψ*)
| *VIffVS*: *VIO σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *VIO σ v i* (*Iff φ ψ*)
| *SExists*: ∃ *z*. *SAT σ* (*v* (*x* := *z*)) *i φ* ⟹ *SAT σ v i* (*Exists x φ*)
| *VExists*: ∀ *z*. *VIO σ* (*v* (*x* := *z*)) *i φ* ⟹ *VIO σ v i* (*Exists x φ*)
| *SForall*: ∀ *z*. *SAT σ* (*v* (*x* := *z*)) *i φ* ⟹ *SAT σ v i* (*Forall x φ*)
| *VForall*: ∃ *z*. *VIO σ* (*v* (*x* := *z*)) *i φ* ⟹ *VIO σ v i* (*Forall x φ*)
| *SPrev*: *i* > *0* ⟹ *mem* (Δ *σ i*) *I* ⟹ *SAT σ v* (*i*−*1*) *φ* ⟹ *SAT σ v i* (**Y** *I φ*)
| *VPrev*: *i* > *0* ⟹ *VIO σ v* (*i*−*1*) *φ* ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevZ*: *i* = *0* ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevOutL*: *i* > *0* ⟹ (Δ *σ i*) < (*left I*) ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevOutR*: *i* > *0* ⟹ *enat* (Δ *σ i*) > (*right I*) ⟹ *VIO σ v i* (**Y** *I φ*)
| *SNext*: *mem* (Δ *σ* (*i*+*1*)) *I* ⟹ *SAT σ v* (*i*+*1*) *φ* ⟹ *SAT σ v i* (**X** *I φ*)
| *VNext*: *VIO σ v* (*i*+*1*) *φ* ⟹ *VIO σ v i* (**X** *I φ*)
| *VNextOutL*: (Δ *σ* (*i*+*1*)) < (*left I*) ⟹ *VIO σ v i* (**X** *I φ*)

| *VNextOutR*: *enat* (Δ σ (i+1)) > (*right I*) $\Longrightarrow$ *VIO* σ v i (**X** *I* φ)
| *SOnce*: $j \leq i \Longrightarrow$ *mem* (δ σ i j) *I* $\Longrightarrow$ *SAT* σ v j $\varphi \Longrightarrow$ *SAT* σ v i (**P** *I* φ)
| *VOnceOut*: τ σ i < τ σ 0 + *left I* $\Longrightarrow$ *VIO* σ v i (**P** *I* φ)
| *VOnce*: j = (*case right I of* $\infty \Rightarrow$ 0
| *enat b* $\Rightarrow$ *ETP_p* σ i *b*) $\Longrightarrow$
(τ σ i) $\geq$ (τ σ 0) + *left I* $\Longrightarrow$
($\bigwedge$*k*. $k \in \{j$ .. *LTP_p* σ i *I*$\} \Longrightarrow$ *VIO* σ v k $\varphi) \Longrightarrow$ *VIO* σ v i (**P** *I* φ)
| *SEventually*: $j \geq i \Longrightarrow$ *mem* (δ σ j i) *I* $\Longrightarrow$ *SAT* σ v j $\varphi \Longrightarrow$ *SAT* σ v i (**F** *I* φ)
| *VEventually*: ($\bigwedge$*k*. $k \in$ (*case right I of* $\infty \Rightarrow \{$*ETP_f* σ i *I* ..$\}$
| *enat b* $\Rightarrow \{$*ETP_f* σ i *I* .. *LTP_f* σ i *b*$\}) \Longrightarrow$ *VIO* σ v k $\varphi) \Longrightarrow$
VIO σ v i (**F** *I* φ)
| *SHistorically*: j = (*case right I of* $\infty \Rightarrow$ 0
| *enat b* $\Rightarrow$ *ETP_p* σ i *b*) $\Longrightarrow$
(τ σ i) $\geq$ (τ σ 0) + *left I* $\Longrightarrow$
($\bigwedge$*k*. $k \in \{j$ .. *LTP_p* σ i *I*$\} \Longrightarrow$ *SAT* σ v k $\varphi) \Longrightarrow$ *SAT* σ v i (**H** *I* φ)
| *SHistoricallyOut*: τ σ i < τ σ 0 + *left I* $\Longrightarrow$ *SAT* σ v i (**H** *I* φ)
| *VHistorically*: $j \leq i \Longrightarrow$ *mem* (δ σ i j) *I* $\Longrightarrow$ *VIO* σ v j $\varphi \Longrightarrow$ *VIO* σ v i (**H** *I* φ)
| *SAlways*: ($\bigwedge$*k*. $k \in$ (*case right I of* $\infty \Rightarrow \{$*ETP_f* σ i *I* ..$\}$
| *enat b* $\Rightarrow \{$*ETP_f* σ i *I* .. *LTP_f* σ i *b*$\}) \Longrightarrow$ *SAT* σ v k $\varphi) \Longrightarrow$
SAT σ v i (**G** *I* φ)
| *VAlways*: $j \geq i \Longrightarrow$ *mem* (δ σ j i) *I* $\Longrightarrow$ *VIO* σ v j $\varphi \Longrightarrow$ *VIO* σ v i (**G** *I* φ)
| *SSince*: $j \leq i \Longrightarrow$ *mem* (δ σ i j) *I* $\Longrightarrow$ *SAT* σ v j $\psi \Longrightarrow$ ($\bigwedge$*k*. $k \in \{j <..$ $i\} \Longrightarrow$
SAT σ v k $\varphi) \Longrightarrow$ *SAT* σ v i (φ **S** *I* ψ)
| *VSinceOut*: τ σ i < τ σ 0 + *left I* $\Longrightarrow$ *VIO* σ v i (φ **S** *I* ψ)
| *VSince*: (*case right I of* $\infty \Rightarrow$ *True*
| *enat b* $\Rightarrow$ *ETP* σ ((τ σ i) − b) $\leq j) \Longrightarrow$
$j \leq i \Longrightarrow$ (τ σ 0) + *left I* $\leq$ (τ σ i) $\Longrightarrow$ *VIO* σ v j $\varphi \Longrightarrow$
($\bigwedge$*k*. $k \in \{j$ .. (*LTP_p* σ i *I*)$\} \Longrightarrow$ *VIO* σ v k $\psi) \Longrightarrow$ *VIO* σ v i (φ **S** *I* ψ)
| *VSinceInf*: j = (*case right I of* $\infty \Rightarrow$ 0
| *enat b* $\Rightarrow$ *ETP_p* σ i *b*) $\Longrightarrow$
(τ σ i) $\geq$ (τ σ 0) + *left I* $\Longrightarrow$
($\bigwedge$*k*. $k \in \{j$ .. *LTP_p* σ i *I*$\} \Longrightarrow$ *VIO* σ v k $\psi) \Longrightarrow$ *VIO* σ v i (φ **S** *I* ψ)
| *SUntil*: $j \geq i \Longrightarrow$ *mem* (δ σ j i) *I* $\Longrightarrow$ *SAT* σ v j $\psi \Longrightarrow$ ($\bigwedge$*k*. $k \in \{i$ ..< $j\} \Longrightarrow$ *SAT* σ v k $\varphi) \Longrightarrow$
SAT σ v i (φ **U** *I* ψ)
| *VUntil*: (*case right I of* $\infty \Rightarrow$ *True*
| *enat b* $\Rightarrow$ $j <$ *LTP_f* σ i *b*) $\Longrightarrow$
$j \geq i \Longrightarrow$ *VIO* σ v j $\varphi \Longrightarrow$ ($\bigwedge$*k*. $k \in \{$*ETP_f* σ i *I* .. $j\} \Longrightarrow$ *VIO* σ v k $\psi) \Longrightarrow$
VIO σ v i (φ **U** *I* ψ)
| *VUntilInf*: ($\bigwedge$*k*. $k \in$ (*case right I of* $\infty \Rightarrow \{$*ETP_f* σ i *I* ..$\}$
| *enat b* $\Rightarrow \{$*ETP_f* σ i *I* .. *LTP_f* σ i *b*$\}) \Longrightarrow$ *VIO* σ v k $\psi) \Longrightarrow$
VIO σ v i (φ **U** *I* ψ)

5.1 Soundness and Completeness

lemma *not_sat_SinceD*:
assumes *unsat*: ¬ $\langle\sigma, v, i\rangle \models \varphi$ **S** *I* ψ **and**
witness: $\exists j \leq i$. *mem* (τ σ i − τ σ j) *I* $\wedge$ $\langle\sigma, v, j\rangle \models \psi$
shows $\exists j \leq i$. *ETP* σ (*case right I of* $\infty \Rightarrow$ 0 | *enat n* $\Rightarrow \tau$ σ i − n) $\leq j \wedge$ ¬ $\langle\sigma, v, j\rangle \models \varphi$
$\wedge$ ($\forall k \in \{j$ .. (*min i* (*LTP* σ (τ σ i − *left I*)))$\}$. ¬ $\langle\sigma, v, k\rangle \models \psi$)
proof −
define *A* **and** *j* **where** *A_def*: $A \equiv \{j.\ j \leq i \wedge$ *mem* (τ σ i − τ σ j) *I* $\wedge$ $\langle\sigma, v, j\rangle \models \psi\}$
and *j_def*: $j \equiv$ *Max A*
from *witness* **have** *j*: $j \leq i$ $\langle\sigma, v, j\rangle \models \psi$ *mem* (τ σ i − τ σ j) *I*
using *Max_in*[*of A*] **unfolding** *j_def*[*symmetric*] **unfolding** *A_def*
by *auto*
moreover
from *j*(*3*) **have** *ETP* σ (*case right I of enat n* $\Rightarrow \tau$ σ i − n | $\infty \Rightarrow$ 0) $\leq j$
unfolding *ETP_def* **by** (*intro Least_le*) (*auto split*: *enat.splits*)

```
  moreover
  { fix j
    assume j: τ σ j ≤ τ σ i
    then obtain k where k: τ σ i < τ σ k
      by (meson ex_le_τ gt_ex less_le_trans)
    have j ≤ ETP σ (Suc (τ σ i))
      unfolding ETP_def
    proof (intro LeastI2[of _ k λi. j ≤ i])
      fix l
      assume Suc (τ σ i) ≤ τ σ l
      with j show j ≤ l
        by (metis lessI less_τD nless_le order_less_le_trans)
    qed (auto simp: Suc_le_eq k(1))
  } note * = this
  { fix k
    assume k ∈ {j <.. (min i (LTP σ (τ σ i − left I)))}
    with j(3) have k: j < k k ≤ i k ≤ Max {j. left I + τ σ j ≤ τ σ i}
      by (auto simp: LTP_def le_diff_conv2 add.commute)
    with j(3) obtain l where left I + τ σ l ≤ τ σ i k ≤ l
      by (subst (asm) Max_ge_iff) (auto simp: le_diff_conv2 *
          intro!: finite_subset[of _ {0 .. ETP σ (τ σ i + 1)}])
    then have mem (τ σ i − τ σ k) I
      using k(1,2) j(3)
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 add.commute dest: τ_mono
        elim: order_trans[rotated] order_trans)
    with Max_ge[of A k] k have ¬ ⟨σ, v, k⟩ ⊨ ψ
      unfolding j_def[symmetric] unfolding A_def
      by auto
  }
  ultimately show ?thesis using unsat
    by (auto dest!: spec[of _ j])
qed

lemma not_sat_UntilD:
  assumes unsat: ¬ ⟨σ, v, i⟩ ⊨ φ U I ψ
    and witness: ∃j ≥ i. mem (δ σ j i) I ∧ ⟨σ, v, j⟩ ⊨ ψ
  shows ∃j ≥ i. (case right I of ∞ ⇒ True | enat n ⇒ j < LTP σ (τ σ i + n))
  ∧ ¬ (⟨σ, v, j⟩ ⊨ φ) ∧ (∀k ∈ {(max i (ETP σ (τ σ i + left I))) .. j}.
   ¬ ⟨σ, v, k⟩ ⊨ ψ)
proof −
  from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
  from witness obtain jmax where jmax: jmax ≥ i ⟨σ, v, jmax⟩ ⊨ ψ
    mem (δ σ jmax i) I by blast
  define A and j where A_def: A ≡ {j. j ≥ i ∧ j ≤ jmax
  ∧ mem (δ σ j i) I ∧ ⟨σ, v, j⟩ ⊨ ψ} and j_def: j ≡ Min A
  have j: j ≥ i ⟨σ, v, j⟩ ⊨ ψ mem (δ σ j i) I
    using A_def j_def jmax Min_in[of A]
    unfolding j_def[symmetric] unfolding A_def
    by fastforce+
  moreover have case right I of ∞ ⇒ True | enat n ⇒ j ≤ LTP σ (τ σ i + n)
    using i0 j(1,3)
    by (auto simp: i_LTP_tau trans_le_add1 split: enat.splits)
  moreover
  { fix k
    assume k_def: k ∈ {(max i (ETP σ (τ σ i + left I))) ..< j}
    then have ki: τ σ k ≥ τ σ i + left I using i_ETP_tau by auto
    with k_def have kj: k < j by auto
    then have τ σ k ≤ τ σ j by auto
```

```
    then have δ σ k i ≤ δ σ j i by auto
    with this j(3) have enat (δ σ k i) ≤ right I
      by (meson enat_ord_simps(1) order_subst2)
    with this ki j(3) have mem_k: mem (δ σ k i) I
      unfolding ETP_def by (auto simp: Least_le)

    with j_def have j ≤ jmax using Min_in[of A]
      using jmax A_def
      by (metis (mono_tags, lifting) Collect_empty_eq
          finite_nat_set_iff_bounded_le mem_Collect_eq order_refl)
    with this k_def have kjm: k ≤ jmax by auto

    with this mem_k ki Min_le[of A k] k_def have k ∉ A
      unfolding j_def[symmetric] unfolding A_def unfolding ETP_def
      using finite_nat_set_iff_bounded_le kj leD by blast
    with this mem_k k_def kjm have ¬ ⟨σ, v, k⟩ ⊨ ψ
      by (simp add: A_def) }
  ultimately show ?thesis using unsat
    by (auto split: enat.splits dest!: spec[of _ j])
qed

lemma soundness_raw: (SAT σ v i φ ⟶ ⟨σ, v, i⟩ ⊨ φ) ∧ (VIO σ v i φ ⟶ ¬ ⟨σ, v, i⟩ ⊨ φ)
proof (induct v i φ rule: SAT_VIO.induct)
  case (VOnceOut i I v φ)
  { fix j
    from τ_mono have j0: τ σ 0 ≤ τ σ j by auto
    then have τ σ i < τ σ j + left I using VOnceOut by linarith
    then have δ σ i j < left I
      using VOnceOut less_τD verit_comp_simplify1(3) by fastforce
    then have ¬ mem (δ σ i j) I by auto }
  then show ?case
    by auto
next
  case (VOnce j I i v φ)
  { fix k
    assume k_def: ⟨σ, v, k⟩ ⊨ φ ∧ mem (δ σ i k) I ∧ k ≤ i
    then have k_tau: τ σ k ≤ τ σ i − left I
      using diff_le_mono2 by fastforce
    then have k_ltp: k ≤ LTP σ (τ σ i − left I)
      using VOnce i_LTP_tau add_le_imp_le_diff
      by blast
    then have k ∉ {j .. LTP_p σ i I}
      using k_def VOnce k_tau
      by auto
    then have k < j using k_def k_ltp by auto }
  then show ?case
    using VOnce
    by (cases right I = ∞)
      (auto 0 0 simp: i_ETP_tau i_LTP_tau le_diff_conv2)
next
  case (VEventually I i v φ)
  { fix k n
    assume r: right I = enat n
    from this have tin0: τ σ i + n ≥ τ σ 0
      by (auto simp add: trans_le_add1)
    define j where j = LTP σ ((τ σ i) + n)
    then have j_i: i ≤ j
      by (auto simp add: i_LTP_tau trans_le_add1 j_def)
```

```
    assume k_def: ⟨σ, v, k⟩ ⊨ φ ∧ mem (δ σ k i) I ∧ i ≤ k
    then have τ σ k ≥ τ σ i + left I
      using le_diff_conv2 by auto
    then have k_etp: k ≥ ETP σ (τ σ i + left I)
      using i_ETP_tau by blast
    from this k_def VEventually have k ∉ {ETP_f σ i I .. j}
      by (auto simp: r j_def)
    then have j < k using r k_def k_etp by auto
    from k_def r have δ σ k i ≤ n by auto
    then have τ σ k ≤ τ σ i + n by auto
    then have k ≤ j using tin0 i_LTP_tau by (auto simp add: j_def) }
  note aux = this
  show ?case
  proof (cases right I)
    case (enat n)
    show ?thesis
      using VEventually[unfolded enat, simplified] aux
      by (simp add: i_ETP_tau enat)
        (metis τ_mono le_add_diff_inverse nat_add_left_cancel_le)
  next
    case infinity
    show ?thesis
      using VEventually
      by (auto simp: infinity i_ETP_tau le_diff_conv2)
  qed
next
  case (SHistorically j I i v φ)
  { fix k
    assume k_def: ¬ ⟨σ, v, k⟩ ⊨ φ ∧ mem (δ σ i k) I ∧ k ≤ i
    then have k_tau: τ σ k ≤ τ σ i − left I
      using diff_le_mono2 by fastforce
    then have k_ltp: k ≤ LTP σ (τ σ i − left I)
      using SHistorically i_LTP_tau add_le_imp_le_diff
      by blast
    then have k ∉ {j .. LTP_p σ i I}
      using k_def SHistorically k_tau
      by auto
    then have k < j using k_def k_ltp by auto }
  then show ?case
    using SHistorically
    by (cases right I = ∞)
      (auto 0 0 simp add: le_diff_conv2 i_ETP_tau i_LTP_tau)
next
  case (SHistoricallyOut i I v φ)
  { fix j
    from τ_mono have j0: τ σ 0 ≤ τ σ j by auto
    then have τ σ i < τ σ j + left I using SHistoricallyOut by linarith
    then have δ σ i j < left I
      using SHistoricallyOut less_τD not_le by fastforce
    then have ¬ mem (δ σ i j) I by auto }
  then show ?case by auto
next
  case (SAlways I i v φ)
  { fix k n
    assume r: right I = enat n
    from this SAlways have tin0: τ σ i + n ≥ τ σ 0
      by (auto simp add: trans_le_add1)
    define j where j = LTP σ ((τ σ i) + n)
```

```
    from SAlways have j_i: i ≤ j
      by (auto simp add: i_LTP_tau trans_le_add1 j_def)
    assume k_def: ¬ ⟨σ, v, k⟩ ⊨ φ ∧ mem (δ σ k i) I ∧ i ≤ k
    then have τ σ k ≥ τ σ i + left I
      using le_diff_conv2 by auto
    then have k_etp: k ≥ ETP σ (τ σ i + left I)
      using SAlways i_ETP_tau by blast
    from this k_def SAlways have k ∉ {ETP_f σ i I .. j}
      by (auto simp: r j_def)
    then have j < k using SAlways k_def k_etp by simp
    from k_def r have δ σ k i ≤ n by simp
    then have τ σ k ≤ τ σ i + n by simp
    then have k ≤ j
      using tin0 i_LTP_tau
      by (auto simp add: j_def) }
  note aux = this
  show ?case
  proof (cases right I)
    case (enat n)
    show ?thesis
      using SAlways[unfolded enat, simplified] aux
      by (simp add: i_ETP_tau le_diff_conv2 enat)
        (metis Groups.ab_semigroup_add_class.add.commute add_le_imp_le_diff)
  next
    case infinity
    show ?thesis
      using SAlways
      by (auto simp: infinity i_ETP_tau le_diff_conv2)
  qed
next
  case (VSinceOut i I v φ ψ)
  { fix j
    from τ_mono have j0: τ σ 0 ≤ τ σ j by auto
    then have τ σ i < τ σ j + left I using VSinceOut by linarith
    then have δ σ i j < left I using VSinceOut j0
    by (metis add.commute gr_zeroI leI less_τD less_diff_conv2 order_less_imp_not_less zero_less_diff)
    then have ¬ mem (δ σ i j) I by auto }
  then show ?case using VSinceOut by auto
next
  case (VSince I i j v φ ψ)
  { fix k
    assume k_def: ⟨σ, v, k⟩ ⊨ ψ ∧ mem (δ σ i k) I ∧ k ≤ i
    then have τ σ k ≤ τ σ i − left I using diff_le_mono2 by fastforce
    then have k_ltp: k ≤ LTP σ (τ σ i − left I)
      using VSince i_LTP_tau add_le_imp_le_diff
      by blast
    then have k < j using k_def VSince(7)[of k]
      by force
    then have j ∈ {k <.. i} ∧ ¬ ⟨σ, v, j⟩ ⊨ φ using VSince
      by auto }
  then show ?case using VSince
    by force
next
  case (VSinceInf j I i v ψ φ)
  { fix k
    assume k_def: ⟨σ, v, k⟩ ⊨ ψ ∧ mem (δ σ i k) I ∧ k ≤ i
    then have k_tau: τ σ k ≤ τ σ i − left I
      using diff_le_mono2 by fastforce
```

```
    then have k_ltp: k ≤ LTP σ (τ σ i − left I)
      using VSinceInf i_LTP_tau add_le_imp_le_diff
      by blast
    then have k ∉ {j .. LTP_p σ i I}
      using k_def VSinceInf k_tau
      by auto
    then have k < j using k_def k_ltp by auto }
  then show ?case
    using VSinceInf
    by (cases right I = ∞)
      (auto 0 0 simp: i_ETP_tau i_LTP_tau le_diff_conv2)
next
  case (VUntil I j i v φ ψ)
  { fix k
    assume k_def: ⟨σ, v, k⟩ ⊨ ψ ∧ mem (δ σ k i) I ∧ i ≤ k
    then have τ σ k ≥ τ σ i + left I
      using le_diff_conv2 by auto
    then have k_etp: k ≥ ETP σ (τ σ i + left I)
      using VUntil i_ETP_tau by blast
    from this k_def VUntil have k ∉ {ETP_f σ i I .. j} by auto
    then have j < k using k_etp k_def by auto
    then have j ∈ {i ..< k} ∧ VIO σ v j φ using VUntil k_def
      by auto }
  then show ?case
    using VUntil by force
next
  case (VUntilInf I i v ψ φ)
  { fix k n
    assume r: right I = enat n
    from this VUntilInf have tin0: τ σ i + n ≥ τ σ 0
      by (auto simp add: trans_le_add1)
    define j where j = LTP σ ((τ σ i) + n)
    from VUntilInf have j_i: i ≤ j
      by (auto simp add: i_LTP_tau trans_le_add1 j_def)
    assume k_def: ⟨σ, v, k⟩ ⊨ ψ ∧ mem (δ σ k i) I ∧ i ≤ k
    then have τ σ k ≥ τ σ i + left I
      using le_diff_conv2 by auto
    then have k_etp: k ≥ ETP σ (τ σ i + left I)
      using VUntilInf i_ETP_tau by blast
    from this k_def VUntilInf have k ∉ {ETP_f σ i I .. j}
      by (auto simp: r j_def)
    then have j < k using VUntilInf k_def k_etp by auto
    from k_def r have δ σ k i ≤ n by auto
    then have τ σ k ≤ τ σ i + n by auto
    then have k ≤ j
      using tin0 VUntilInf i_LTP_tau r k_def
      by (force simp add: j_def) }
  note aux = this
  show ?case
  proof (cases right I)
    case (enat n)
    show ?thesis
      using VUntilInf[unfolded enat, simplified] aux
      by (simp add: i_ETP_tau enat)
        (metis τ_mono le_add_diff_inverse nat_add_left_cancel_le)
  next
    case infinity
    show ?thesis
```

```
    using VUntilInf
    by (auto simp: infinity i_ETP_tau le_diff_conv2)
  qed
qed (auto simp: fun_upd_def split: nat.splits)

lemmas soundness = soundness_raw[THEN conjunct1, THEN mp] soundness_raw[THEN conjunct2,
THEN mp]

lemma completeness_raw: (⟨σ, v, i⟩ ⊨ φ ⟶ SAT σ v i φ) ∧ (¬ ⟨σ, v, i⟩ ⊨ φ ⟶ VIO σ v i φ)
proof (induct φ arbitrary: v i)
  case (Prev I φ)
  show ?case using Prev
    by (auto intro: SAT_VIO.SPrev SAT_VIO.VPrev SAT_VIO.VPrevOutL SAT_VIO.VPrevOutR
SAT_VIO.VPrevZ split: nat.splits)
next
  case (Once I φ)
  { assume ⟨σ, v, i⟩ ⊨ P I φ
    with Once have SAT σ v i (P I φ)
      by (auto intro: SAT_VIO.SOnce) }
  moreover
  { assume i_l: τ σ i < τ σ 0 + left I
    with Once have VIO σ v i (P I φ)
      by (auto intro: SAT_VIO.VOnceOut) }
  moreover
  { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ P I φ
      and i_ge: τ σ 0 + left I ≤ τ σ i
    with Once have VIO σ v i (P I φ)
      by (auto intro!: SAT_VIO.VOnce simp: i_LTP_tau i_ETP_tau
          split: enat.splits) }
  ultimately show ?case
    by force
next
  case (Historically I φ)
  from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
  { assume sat: ⟨σ, v, i⟩ ⊨ H I φ
      and i_ge: τ σ i ≥ τ σ 0 + left I
    with Historically have SAT σ v i (H I φ)
      using le_diff_conv
      by (auto intro!: SAT_VIO.SHistorically simp: i_LTP_tau i_ETP_tau
          split: enat.splits) }
  moreover
  { assume ¬ ⟨σ, v, i⟩ ⊨ H I φ
    with Historically have VIO σ v i (H I φ)
      by (auto intro: SAT_VIO.VHistorically) }
  moreover
  { assume i_l: τ σ i < τ σ 0 + left I
    with Historically have SAT σ v i (H I φ)
      by (auto intro: SAT_VIO.SHistoricallyOut) }
  ultimately show ?case
    by force
next
  case (Eventually I φ)
  from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
  { assume ⟨σ, v, i⟩ ⊨ F I φ
    with Eventually have SAT σ v i (F I φ)
      by (auto intro: SAT_VIO.SEventually) }
  moreover
  { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ F I φ
```

```
    with Eventually have VIO σ v i (F I φ)
      by (auto intro!: SAT_VIO.VEventually simp: add_increasing2 i0 i_LTP_tau i_ETP_tau
          split: enat.splits) }
  ultimately show ?case by auto
next
  case (Always I φ)
    from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
  { assume ¬ ⟨σ, v, i⟩ ⊨ G I φ
    with Always have VIO σ v i (G I φ)
      by (auto intro: SAT_VIO.VAlways) }
  moreover
  { assume sat: ⟨σ, v, i⟩ ⊨ G I φ
    with Always have SAT σ v i (G I φ)
     by (auto intro!: SAT_VIO.SAlways simp: add_increasing2 i0 i_LTP_tau i_ETP_tau le_diff_conv
split: enat.splits)}
  ultimately show ?case by auto
next
  case (Since φ I ψ)
  { assume ⟨σ, v, i⟩ ⊨ φ S I ψ
    with Since have SAT σ v i (φ S I ψ)
      by (auto intro: SAT_VIO.SSince) }
  moreover
  { assume i_l: τ σ i < τ σ 0 + left I
    with Since have VIO σ v i (φ S I ψ)
      by (auto intro: SAT_VIO.VSinceOut) }
  moreover
  { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ φ S I ψ
      and nw: ∀ j≤i. ¬ mem (δ σ i j) I ∨ ¬ ⟨σ, v, j⟩ ⊨ ψ
      and i_ge: τ σ 0 + left I ≤ τ σ i
    with Since have VIO σ v i (φ S I ψ)
      by (auto intro!: SAT_VIO.VSinceInf simp: i_LTP_tau i_ETP_tau
          split: enat.splits)}
  moreover
  { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ φ S I ψ
      and jw: ∃ j≤i. mem (δ σ i j) I ∧ ⟨σ, v, j⟩ ⊨ ψ
      and i_ge: τ σ 0 + left I ≤ τ σ i
    from unsat jw not_sat_SinceD[of σ v i φ I ψ]
    obtain j where j: j ≤ i
      case right I of ∞ ⇒ True | enat n ⇒ ETP σ (τ σ i − n) ≤ j
      ¬ ⟨σ, v, j⟩ ⊨ φ (∀ k ∈ {j .. (min i (LTP σ (τ σ i − left I)))}.
      ¬ ⟨σ, v, k⟩ ⊨ ψ) by (auto split: enat.splits)
    with Since have VIO σ v i (φ S I ψ)
      using i_ge unsat jw
      by (auto intro!: SAT_VIO.VSince) }
  ultimately show ?case
    by (force simp del: sat.simps)
next
  case (Until φ I ψ)
  from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
  { assume ⟨σ, v, i⟩ ⊨ φ U I ψ
    with Until have SAT σ v i (φ U I ψ)
      by (auto intro: SAT_VIO.SUntil) }
  moreover
  { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ φ U I ψ
      and witness: ∃ j ≥ i. mem (δ σ j i) I ∧ ⟨σ, v, j⟩ ⊨ ψ
    from this Until not_sat_UntilD[of σ v i φ I ψ] obtain j
      where j: j ≥ i (case right I of ∞ ⇒ True | enat n
      ⇒ j < LTP σ (τ σ i + n)) ¬ (⟨σ, v, j⟩ ⊨ φ)
```

```
      (∀ k ∈ {(max i (ETP σ (τ σ i + left I))) .. j}. ¬ ⟨σ, v, k⟩ ⊨ ψ)
    by auto
  with Until have VIO σ v i (φ U I ψ)
    using unsat witness
    by (auto intro!: SAT_VIO.VUntil) }
 moreover
 { assume unsat: ¬ ⟨σ, v, i⟩ ⊨ φ U I ψ
    and no_witness: ∀ j ≥ i. ¬ mem (δ σ j i) I ∨ ¬ ⟨σ, v, j⟩ ⊨ ψ
  with Until have VIO σ v i (φ U I ψ)
    by (auto intro!: SAT_VIO.VUntilInf simp: add_increasing2 i0 i_LTP_tau i_ETP_tau
        split: enat.splits)
 }
 ultimately show ?case by auto
qed (auto intro: SAT_VIO.intros)

lemmas completeness = completeness_raw[THEN conjunct1, THEN mp] completeness_raw[THEN conjunct2, THEN mp]

lemma SAT_or_VIO: SAT σ v i φ ∨ VIO σ v i φ
 using completeness[of σ v i φ] by auto

end

unbundle MFOTL_no_notation
```

6 Proof Objects

```
datatype (dead 'n, dead 'd) sproof = STT nat
 | SPred nat 'n ('n, 'd) Formula.trm list
 | SEq_Const nat 'n 'd
 | SNeg ('n, 'd) vproof
 | SOrL ('n, 'd) sproof
 | SOrR ('n, 'd) sproof
 | SAnd ('n, 'd) sproof ('n, 'd) sproof
 | SImpL ('n, 'd) vproof
 | SImpR ('n, 'd) sproof
 | SIffSS ('n, 'd) sproof ('n, 'd) sproof
 | SIffVV ('n, 'd) vproof ('n, 'd) vproof
 | SExists 'n 'd ('n, 'd) sproof
 | SForall 'n ('d, ('n, 'd) sproof) part
 | SPrev ('n, 'd) sproof
 | SNext ('n, 'd) sproof
 | SOnce nat ('n, 'd) sproof
 | SEventually nat ('n, 'd) sproof
 | SHistorically nat nat ('n, 'd) sproof list
 | SHistoricallyOut nat
 | SAlways nat nat ('n, 'd) sproof list
 | SSince ('n, 'd) sproof ('n, 'd) sproof list
 | SUntil ('n, 'd) sproof list ('n, 'd) sproof
and ('n, 'd) vproof = VFF nat
 | VPred nat 'n ('n, 'd) Formula.trm list
 | VEq_Const nat 'n 'd
 | VNeg ('n, 'd) sproof
 | VOr ('n, 'd) vproof ('n, 'd) vproof
 | VAndL ('n, 'd) vproof
 | VAndR ('n, 'd) vproof
 | VImp ('n, 'd) sproof ('n, 'd) vproof
```

```
  | VIffSV ('n, 'd) sproof ('n, 'd) vproof
  | VIffVS ('n, 'd) vproof ('n, 'd) sproof
  | VExists 'n ('d, ('n, 'd) vproof) part
  | VForall 'n 'd ('n, 'd) vproof
  | VPrev ('n, 'd) vproof
  | VPrevZ
  | VPrevOutL nat
  | VPrevOutR nat
  | VNext ('n, 'd) vproof
  | VNextOutL nat
  | VNextOutR nat
  | VOnceOut nat
  | VOnce nat nat ('n, 'd) vproof list
  | VEventually nat nat ('n, 'd) vproof list
  | VHistorically nat ('n, 'd) vproof
  | VAlways nat ('n, 'd) vproof
  | VSinceOut nat
  | VSince nat ('n, 'd) vproof ('n, 'd) vproof list
  | VSinceInf nat nat ('n, 'd) vproof list
  | VUntil nat ('n, 'd) vproof list ('n, 'd) vproof
  | VUntilInf nat nat ('n, 'd) vproof list

type_synonym ('n, 'd) proof = ('n, 'd) sproof + ('n, 'd) vproof

type_synonym ('n, 'd) expl = ('d, ('n, 'd) proof, 'n) pdt

fun s_at :: ('n, 'd) sproof ⇒ nat and
  v_at :: ('n, 'd) vproof ⇒ nat where
  s_at (STT i) = i
| s_at (SPred i _ _) = i
| s_at (SEq_Const i _ _) = i
| s_at (SNeg vp) = v_at vp
| s_at (SOrL sp1) = s_at sp1
| s_at (SOrR sp2) = s_at sp2
| s_at (SAnd sp1 _) = s_at sp1
| s_at (SImpL vp1) = v_at vp1
| s_at (SImpR sp2) = s_at sp2
| s_at (SIffSS sp1 _) = s_at sp1
| s_at (SIffVV vp1 _) = v_at vp1
| s_at (SExists _ _ sp) = s_at sp
| s_at (SForall _ part) = s_at (part_hd part)
| s_at (SPrev sp) = s_at sp + 1
| s_at (SNext sp) = s_at sp - 1
| s_at (SOnce i _) = i
| s_at (SEventually i _) = i
| s_at (SHistorically i _ _) = i
| s_at (SHistoricallyOut i) = i
| s_at (SAlways i _ _) = i
| s_at (SSince sp2 sp1s) = (case sp1s of [] ⇒ s_at sp2 | _ ⇒ s_at (last sp1s))
| s_at (SUntil sp1s sp2) = (case sp1s of [] ⇒ s_at sp2 | sp1 # _ ⇒ s_at sp1)
| v_at (VFF i) = i
| v_at (VPred i _ _) = i
| v_at (VEq_Const i _ _) = i
| v_at (VNeg sp) = s_at sp
| v_at (VOr vp1 _) = v_at vp1
| v_at (VAndL vp1) = v_at vp1
| v_at (VAndR vp2) = v_at vp2
| v_at (VImp sp1 _) = s_at sp1
```

| *v_at (VIffSV sp1 _) = s_at sp1*
| *v_at (VIffVS vp1 _) = v_at vp1*
| *v_at (VExists _ part) = v_at (part_hd part)*
| *v_at (VForall _ _ vp1) = v_at vp1*
| *v_at (VPrev vp) = v_at vp + 1*
| *v_at (VPrevZ) = 0*
| *v_at (VPrevOutL i) = i*
| *v_at (VPrevOutR i) = i*
| *v_at (VNext vp) = v_at vp − 1*
| *v_at (VNextOutL i) = i*
| *v_at (VNextOutR i) = i*
| *v_at (VOnceOut i) = i*
| *v_at (VOnce i _ _) = i*
| *v_at (VEventually i _ _) = i*
| *v_at (VHistorically i _) = i*
| *v_at (VAlways i _) = i*
| *v_at (VSinceOut i) = i*
| *v_at (VSince i _ _) = i*
| *v_at (VSinceInf i _ _) = i*
| *v_at (VUntil i _ _) = i*
| *v_at (VUntilInf i _ _) = i*

definition *p_at* :: (*'n*, *'d*) *proof* ⇒ *nat* **where** *p_at p* = *case_sum s_at v_at p*

7 Auxiliary Lemmas

lemma *Cons_eq_upt_conv*: *x # xs* = [*m ..< n*] ⟷ *m < n* ∧ *x* = *m* ∧ *xs* = [*Suc m ..< n*]
by (*induct n arbitrary*: *xs*) (*force simp*: *Cons_eq_append_conv*)+

lemma *map_setE*[*elim_format*]: *map f xs* = *ys* ⟹ *y* ∈ *set ys* ⟹ ∃ *x*∈*set xs*. *f x* = *y*
by (*induct xs arbitrary*: *ys*) *auto*

lemma *set_Cons_eq*: *set_Cons X XS* = (⋃ *xs*∈*XS*. (λ*x*. *x # xs*) ` *X*)
by (*auto simp*: *set_Cons_def*)

lemma *set_Cons_empty_iff*: *set_Cons X XS* = {} ⟷ (*X* = {} ∨ *XS* = {})
by (*auto simp*: *set_Cons_eq*)

lemma *infinite_set_ConsI*:
XS ≠ {} ⟹ *infinite X* ⟹ *infinite* (*set_Cons X XS*)
X ≠ {} ⟹ *infinite XS* ⟹ *infinite* (*set_Cons X XS*)
proof(*unfold set_Cons_eq*)
assume *infinite X* **and** *XS* ≠ {}
then obtain *xs* **where** *xs* ∈ *XS*
by *blast*
hence *inj* (λ*x*. *x # xs*)
by (*clarsimp simp*: *inj_on_def*)
hence *infinite* ((λ*x*. *x # xs*) ` *X*)
using ‹*infinite X*› *finite_imageD inj_on_def*
by *blast*
moreover have ((λ*x*. *x # xs*) ` *X*) ⊆ (⋃ *xs*∈*XS*. (λ*x*. *x # xs*) ` *X*)
using ‹*xs* ∈ *XS*› **by** *auto*
ultimately show *infinite* (⋃ *xs*∈*XS*. (λ*x*. *x # xs*) ` *X*)
by (*simp add*: *infinite_super*)
next
assume *infinite XS* **and** *X* ≠ {}
then show *infinite* (⋃ *xs*∈*XS*. (λ*x*. *x # xs*) ` *X*)

by (*elim contrapos_nn finite_surj*[*of* _ _ *tl*]) (*auto simp*: *image_iff*)
qed

primrec *fst_pos* :: *'a list* ⇒ *'a* ⇒ *nat option*
where *fst_pos* [] *x* = *None*
| *fst_pos* (*y*#*ys*) *x* = (*if x* = *y then Some 0 else*
(*case fst_pos ys x of None* ⇒ *None* | *Some n* ⇒ *Some* (*Suc n*)))

lemma *fst_pos_None_iff*: *fst_pos xs x* = *None* ⟷ *x* ∉ *set xs*
by (*induct xs arbitrary*: *x*; *force split*: *option.splits*)

lemma *nth_fst_pos*: *x* ∈ *set xs* ⟹ *xs* ! (*the* (*fst_pos xs x*)) = *x*
by (*induct xs arbitrary*: *x*; *fastforce simp*: *fst_pos_None_iff split*: *option.splits*)

primrec *positions* :: *'a list* ⇒ *'a* ⇒ *nat list*
where *positions* [] *x* = []
| *positions* (*y*#*ys*) *x* = (λ*ns*. *if x* = *y then 0* # *ns else ns*) (*map Suc* (*positions ys x*))

lemma *eq_positions_iff*: *length xs* = *length ys*
⟹ *positions xs x* = *positions ys y* ⟷ (∀ *n*< *length xs*. *xs* ! *n* = *x* ⟷ *ys* ! *n* = *y*)
by (*induct xs ys arbitrary*: *x y rule*: *list_induct2*) (*use less_Suc_eq_0_disj* **in** *auto*)

lemma *positions_eq_nil_iff*: *positions xs x* = [] ⟷ *x* ∉ *set xs*
by (*induct xs*) *simp_all*

lemma *positions_nth*: *n* ∈ *set* (*positions xs x*) ⟹ *xs* ! *n* = *x*
by (*induct xs arbitrary*: *n x*)
(*auto simp*: *positions_eq_nil_iff*[*symmetric*] *split*: *if_splits*)

lemma *set_positions_eq*: *set* (*positions xs x*) = {*n*. *xs* ! *n* = *x* ∧ *n* < *length xs*}
by (*induct xs arbitrary*: *x*)
(*use less_Suc_eq_0_disj* **in** ‹*auto simp*: *positions_eq_nil_iff*[*symmetric*] *image_iff split*: *if_splits*›)

lemma *positions_length*: *n* ∈ *set* (*positions xs x*) ⟹ *n* < *length xs*
by (*induct xs arbitrary*: *n x*)
(*auto simp*: *positions_eq_nil_iff*[*symmetric*] *split*: *if_splits*)

lemma *positions_nth_cong*:
m ∈ *set* (*positions xs x*) ⟹ *n* ∈ *set* (*positions xs x*) ⟹ *xs* ! *n* = *xs* ! *m*
using *positions_nth*[*of* _ *xs x*] **by** *simp*

lemma *fst_pos_in_positions*: *x* ∈ *set xs* ⟹ *the* (*fst_pos xs x*) ∈ *set* (*positions xs x*)
by (*induct xs arbitrary*: *x*, *simp*)
(*fastforce simp*: *hd_map fst_pos_None_iff split*: *option.splits*)

lemma *hd_positions_eq_fst_pos*: *x* ∈ *set xs* ⟹ *hd* (*positions xs x*) = *the* (*fst_pos xs x*)
by (*induct xs arbitrary*: *x*)
(*auto simp*: *hd_map fst_pos_None_iff positions_eq_nil_iff split*: *option.splits*)

lemma *sorted_positions*: *sorted* (*positions xs x*)
by (*induct xs arbitrary*: *x*) (*auto simp add*: *sorted_iff_nth_Suc nth_Cons' gr0_conv_Suc*)

lemma *Min_sorted_list*: *sorted xs* ⟹ *xs* ≠ [] ⟹ *Min* (*set xs*) = *hd xs*
by (*induct xs*)
(*auto simp*: *Min_insert2*)

lemma *Min_positions*: *x* ∈ *set xs* ⟹ *Min* (*set* (*positions xs x*)) = *the* (*fst_pos xs x*)
by (*auto simp*: *Min_sorted_list*[*OF sorted_positions*]

positions_eq_nil_iff hd_positions_eq_fst_pos)

lemma *subset_positions_map_fst*: *set* (*positions tXs tX*) ⊆ *set* (*positions* (*map fst tXs*) (*fst tX*))
by (*induct tXs arbitrary*: *tX*)
(*auto simp*: *subset_eq*)

lemma *subset_positions_map_snd*: *set* (*positions tXs tX*) ⊆ *set* (*positions* (*map snd tXs*) (*snd tX*))
by (*induct tXs arbitrary*: *tX*)
(*auto simp*: *subset_eq*)

lemma *Max_eqI*: *finite A* ⟹ *A* ≠ {} ⟹ (⋀*a*. *a* ∈ *A* ⟹ *a* ≤ *b*) ⟹ ∃ *a*∈*A*. *b* ≤ *a* ⟹ *Max A* = *b*
by (*rule antisym*[*OF Max.boundedI Max_ge_iff*[*THEN iffD2*]]; *clarsimp*)

lemma *Max_Suc*: *X* ≠ {} ⟹ *finite X* ⟹ *Max* (*Suc* ‘ *X*) = *Suc* (*Max X*)
using *Max_ge Max_in*
by (*intro Max_eqI*) *blast*+

lemma *Max_insert0*: *X* ≠ {} ⟹ *finite X* ⟹ *Max* (*insert* (*0*::*nat*) *X*) = *Max X*
using *Max_ge Max_in*
by (*intro Max_eqI*) *blast*+

lemma *positions_Cons_notin_tail*: *x* ∉ *set xs* ⟹ *positions* (*x* # *xs*) *x* = [*0*::*nat*]
by (*cases xs*) (*auto simp*: *positions_eq_nil_iff*)

lemma *Max_set_positions_Cons_hd*:
x ∉ *set xs* ⟹ *Max* (*set* (*positions* (*x* # *xs*) *x*)) = *0*
by (*subst positions_Cons_notin_tail*) *simp_all*

lemma *Max_set_positions_Cons_tl*:
y ∈ *set xs* ⟹ *Max* (*set* (*positions* (*x* # *xs*) *y*)) = *Suc* (*Max* (*set* (*positions xs y*)))
by (*auto simp*: *Max_Suc positions_eq_nil_iff*)

lemma *max_aux*: *finite X* ⟹ *Suc j* ∈ *X* ⟹ *Max* (*insert* (*Suc j*) (*X* − {*j*})) = *Max* (*insert j X*)
by (*smt* (*verit*) *max.orderI Max.insert_remove Max_ge Max_insert empty_iff insert_Diff_single*
insert_absorb insert_iff max_def not_less_eq_eq)

lemma *ball_swap*: (∀ *x* ∈ *A*. ∀ *y* ∈ *B*. *P x y*) = (∀ *y* ∈ *B*. ∀ *x* ∈ *A*. *P x y*)
by *auto*

lemma *ball_triv_nonempty*: *A* ≠ {} ⟹ (∀ *x* ∈ *A*. *P*) = *P*
by *auto*

8 Proof Checker

unbundle *MFOTL_notation*

context fixes σ :: (′*n*, ′*d* :: {*default*, *linorder*}) *trace*

begin

fun *s_check* :: (′*n*, ′*d*) *env* ⇒ (′*n*, ′*d*) *formula* ⇒ (′*n*, ′*d*) *sproof* ⇒ *bool*
and *v_check* :: (′*n*, ′*d*) *env* ⇒ (′*n*, ′*d*) *formula* ⇒ (′*n*, ′*d*) *vproof* ⇒ *bool* **where**
s_check v f p = (*case* (*f*, *p*) *of*
(⊤, *STT i*) ⇒ *True*
| (*r* † *ts*, *SPred i s ts*′) ⇒
(*r* = *s* ∧ *ts* = *ts*′ ∧ (*r*, *v*⟦*ts*⟧) ∈ Γ σ *i*)
| (*x* ≈ *c*, *SEq_Const i x*′ *c*′) ⇒

$$
\begin{array}{l}
\quad (c = c' \wedge x = x' \wedge v\ x = c) \\
\mid (\neg_F\ \varphi,\ SNeg\ vp) \Rightarrow v\_check\ v\ \varphi\ vp \\
\mid (\varphi \vee_F \psi,\ SOrL\ sp1) \Rightarrow s\_check\ v\ \varphi\ sp1 \\
\mid (\varphi \vee_F \psi,\ SOrR\ sp2) \Rightarrow s\_check\ v\ \psi\ sp2 \\
\mid (\varphi \wedge_F \psi,\ SAnd\ sp1\ sp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge s\_check\ v\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2 \\
\mid (\varphi \longrightarrow_F \psi,\ SImpL\ vp1) \Rightarrow v\_check\ v\ \varphi\ vp1 \\
\mid (\varphi \longrightarrow_F \psi,\ SImpR\ sp2) \Rightarrow s\_check\ v\ \psi\ sp2 \\
\mid (\varphi \longleftrightarrow_F \psi,\ SIffSS\ sp1\ sp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge s\_check\ v\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2 \\
\mid (\varphi \longleftrightarrow_F \psi,\ SIffVV\ vp1\ vp2) \Rightarrow v\_check\ v\ \varphi\ vp1 \wedge v\_check\ v\ \psi\ vp2 \wedge v\_at\ vp1 = v\_at\ vp2 \\
\mid (\exists_F x.\ \varphi,\ SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s\_check\ (v\ (x := val))\ \varphi\ sp) \\
\mid (\forall_F x.\ \varphi,\ SForall\ y\ sp\_part) \Rightarrow (let\ i = s\_at\ (part\_hd\ sp\_part) \\
\quad in\ x = y \wedge (\forall (sub,\ sp) \in SubsVals\ sp\_part.\ s\_at\ sp = i \wedge (\forall z \in sub.\ s\_check\ (v\ (x := z))\ \varphi\ sp))) \\
\mid (\mathbf{Y}\ I\ \varphi,\ SPrev\ sp) \Rightarrow \\
\quad (let\ j = s\_at\ sp;\ i = s\_at\ (SPrev\ sp)\ in \\
\quad i = j{+}1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s\_check\ v\ \varphi\ sp) \\
\mid (\mathbf{X}\ I\ \varphi,\ SNext\ sp) \Rightarrow \\
\quad (let\ j = s\_at\ sp;\ i = s\_at\ (SNext\ sp)\ in \\
\quad j = i{+}1 \wedge mem\ (\Delta\ \sigma\ j)\ I \wedge s\_check\ v\ \varphi\ sp) \\
\mid (\mathbf{P}\ I\ \varphi,\ SOnce\ i\ sp) \Rightarrow \\
\quad (let\ j = s\_at\ sp\ in \\
\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge s\_check\ v\ \varphi\ sp) \\
\mid (\mathbf{F}\ I\ \varphi,\ SEventually\ i\ sp) \Rightarrow \\
\quad (let\ j = s\_at\ sp\ in \\
\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge s\_check\ v\ \varphi\ sp) \\
\mid (\mathbf{H}\ I\ \varphi,\ SHistoricallyOut\ i) \Rightarrow \\
\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I \\
\mid (\mathbf{H}\ I\ \varphi,\ SHistorically\ i\ li\ sps) \Rightarrow \\
\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - b)) \\
\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \\
\quad \wedge map\ s\_at\ sps = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1] \\
\quad \wedge (\forall sp \in set\ sps.\ s\_check\ v\ \varphi\ sp)) \\
\mid (\mathbf{G}\ I\ \varphi,\ SAlways\ i\ hi\ sps) \Rightarrow \\
\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \\
\quad \wedge right\ I \neq \infty \\
\quad \wedge map\ s\_at\ sps = [(ETP\_f\ \sigma\ i\ I)\ ..<\ hi + 1] \\
\quad \wedge (\forall sp \in set\ sps.\ s\_check\ v\ \varphi\ sp)) \\
\mid (\varphi\ \mathbf{S}\ I\ \psi,\ SSince\ sp2\ sp1s) \Rightarrow \\
\quad (let\ i = s\_at\ (SSince\ sp2\ sp1s);\ j = s\_at\ sp2\ in \\
\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \\
\quad \wedge map\ s\_at\ sp1s = [j{+}1\ ..<\ i{+}1] \\
\quad \wedge s\_check\ v\ \psi\ sp2 \\
\quad \wedge (\forall sp1 \in set\ sp1s.\ s\_check\ v\ \varphi\ sp1)) \\
\mid (\varphi\ \mathbf{U}\ I\ \psi,\ SUntil\ sp1s\ sp2) \Rightarrow \\
\quad (let\ i = s\_at\ (SUntil\ sp1s\ sp2);\ j = s\_at\ sp2\ in \\
\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \\
\quad \wedge map\ s\_at\ sp1s = [i\ ..<\ j] \wedge s\_check\ v\ \psi\ sp2 \\
\quad \wedge (\forall sp1 \in set\ sp1s.\ s\_check\ v\ \varphi\ sp1)) \\
\mid (\_\ ,\ \_) \Rightarrow False) \\
\mid v\_check\ v\ f\ p = (case\ (f,\ p)\ of \\
\quad (\bot,\ VFF\ i) \Rightarrow True \\
\mid (r\ \dagger\ ts,\ VPred\ i\ pred\ ts') \Rightarrow \\
\quad (r = pred \wedge ts = ts' \wedge (r,\ v[\![ts]\!]) \notin \Gamma\ \sigma\ i) \\
\mid (x \approx c,\ VEq\_Const\ i\ x'\ c') \Rightarrow \\
\quad (c = c' \wedge x = x' \wedge v\ x \neq c) \\
\mid (\neg_F\ \varphi,\ VNeg\ sp) \Rightarrow s\_check\ v\ \varphi\ sp \\
\mid (\varphi \vee_F \psi,\ VOr\ vp1\ vp2) \Rightarrow v\_check\ v\ \varphi\ vp1 \wedge v\_check\ v\ \psi\ vp2 \wedge v\_at\ vp1 = v\_at\ vp2 \\
\mid (\varphi \wedge_F \psi,\ VAndL\ vp1) \Rightarrow v\_check\ v\ \varphi\ vp1 \\
\mid (\varphi \wedge_F \psi,\ VAndR\ vp2) \Rightarrow v\_check\ v\ \psi\ vp2
\end{array}
$$

$$
\begin{aligned}
&\mid (\varphi \longrightarrow_F \psi,\ VImp\ sp1\ vp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge v\_check\ v\ \psi\ vp2 \wedge s\_at\ sp1 = v\_at\ vp2\\
&\mid (\varphi \longleftrightarrow_F \psi,\ VIffSV\ sp1\ vp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge v\_check\ v\ \psi\ vp2 \wedge s\_at\ sp1 = v\_at\ vp2\\
&\mid (\varphi \longleftrightarrow_F \psi,\ VIffVS\ vp1\ sp2) \Rightarrow v\_check\ v\ \varphi\ vp1 \wedge s\_check\ v\ \psi\ sp2 \wedge v\_at\ vp1 = s\_at\ sp2\\
&\mid (\exists_F x.\ \varphi,\ VExists\ y\ vp\_part) \Rightarrow (let\ i = v\_at\ (part\_hd\ vp\_part)\\
&\quad in\ x = y \wedge (\forall (sub, vp) \in SubsVals\ vp\_part.\ v\_at\ vp = i \wedge (\forall z \in sub.\ v\_check\ (v\ (x := z))\ \varphi\ vp)))\\
&\mid (\forall_F x.\ \varphi,\ VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v\_check\ (v\ (x := val))\ \varphi\ vp)\\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrev\ vp) \Rightarrow\\
&\ (let\ j = v\_at\ vp;\ i = v\_at\ (VPrev\ vp)\ in\\
&\ i = j{+}1 \wedge v\_check\ v\ \varphi\ vp)\\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevZ) \Rightarrow True\\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevOutL\ i) \Rightarrow\\
&\ i > 0 \wedge \Delta\ \sigma\ i < left\ I\\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevOutR\ i) \Rightarrow\\
&\ i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I\\
&\mid (\mathbf{X}\ I\ \varphi,\ VNext\ vp) \Rightarrow\\
&\ (let\ j = v\_at\ vp;\ i = v\_at\ (VNext\ vp)\ in\\
&\ j = i{+}1 \wedge v\_check\ v\ \varphi\ vp)\\
&\mid (\mathbf{X}\ I\ \varphi,\ VNextOutL\ i) \Rightarrow\\
&\ \Delta\ \sigma\ (i{+}1) < left\ I\\
&\mid (\mathbf{X}\ I\ \varphi,\ VNextOutR\ i) \Rightarrow\\
&\ enat\ (\Delta\ \sigma\ (i{+}1)) > right\ I\\
&\mid (\mathbf{P}\ I\ \varphi,\ VOnceOut\ i) \Rightarrow\\
&\ \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I\\
&\mid (\mathbf{P}\ I\ \varphi,\ VOnce\ i\ li\ vps) \Rightarrow\\
&\ (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b)\\
&\ \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i\\
&\ \wedge map\ v\_at\ vps = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1]\\
&\ \wedge (\forall vp \in set\ vps.\ v\_check\ v\ \varphi\ vp))\\
&\mid (\mathbf{F}\ I\ \varphi,\ VEventually\ i\ hi\ vps) \Rightarrow\\
&\ (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \wedge right\ I \neq \infty\\
&\ \wedge map\ v\_at\ vps = [(ETP\_f\ \sigma\ i\ I)\ ..<\ hi + 1]\\
&\ \wedge (\forall vp \in set\ vps.\ v\_check\ v\ \varphi\ vp))\\
&\mid (\mathbf{H}\ I\ \varphi,\ VHistorically\ i\ vp) \Rightarrow\\
&\ (let\ j = v\_at\ vp\ in\\
&\ j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v\_check\ v\ \varphi\ vp)\\
&\mid (\mathbf{G}\ I\ \varphi,\ VAlways\ i\ vp) \Rightarrow\\
&\ (let\ j = v\_at\ vp\\
&\ in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v\_check\ v\ \varphi\ vp)\\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceOut\ i) \Rightarrow\\
&\ \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I\\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSince\ i\ vp1\ vp2s) \Rightarrow\\
&\ (let\ j = v\_at\ vp1\ in\\
&\ (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b \leq j) \wedge j \leq i\\
&\ \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i\\
&\ \wedge map\ v\_at\ vp2s = [j\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1] \wedge v\_check\ v\ \varphi\ vp1\\
&\ \wedge (\forall vp2 \in set\ vp2s.\ v\_check\ v\ \psi\ vp2))\\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceInf\ i\ li\ vp2s) \Rightarrow\\
&\ (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b)\\
&\ \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i\\
&\ \wedge map\ v\_at\ vp2s = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1]\\
&\ \wedge (\forall vp2 \in set\ vp2s.\ v\_check\ v\ \psi\ vp2))\\
&\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntil\ i\ vp2s\ vp1) \Rightarrow\\
&\ (let\ j = v\_at\ vp1\ in\\
&\ (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP\_f\ \sigma\ i\ b) \wedge i \leq j\\
&\ \wedge map\ v\_at\ vp2s = [ETP\_f\ \sigma\ i\ I\ ..<\ j + 1] \wedge v\_check\ v\ \varphi\ vp1\\
&\ \wedge (\forall vp2 \in set\ vp2s.\ v\_check\ v\ \psi\ vp2))\\
&\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntilInf\ i\ hi\ vp2s) \Rightarrow\\
&\ (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \wedge right\ I \neq \infty
\end{aligned}
$$

```
    ∧ map v_at vp2s = [ETP_f σ i I ..< hi + 1]
    ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
  | ( _ , _) ⇒ False)

declare s_check.simps[simp del] v_check.simps[simp del]
simps_of_case s_check_simps[simp]: s_check.simps[unfolded prod.case] (splits: formula.split sproof.split)
simps_of_case v_check_simps[simp]: v_check.simps[unfolded prod.case] (splits: formula.split vproof.split)
```

8.1 Checker Soundness

```
lemma check_soundness:
  s_check v φ sp ⟹ SAT σ v (s_at sp) φ
  v_check v φ vp ⟹ VIO σ v (v_at vp) φ
proof (induction sp and vp arbitrary: v φ and v φ)
  case STT
  then show ?case by (cases φ) (auto intro: SAT_VIO.STT)
next
  case SPred
  then show ?case by (cases φ) (auto intro: SAT_VIO.SPred)
next
  case SEq_Const
  then show ?case by (cases φ) (auto intro: SAT_VIO.SEq_Const)
next
  case SNeg
  then show ?case by (cases φ) (auto intro: SAT_VIO.SNeg)
next
  case SAnd
  then show ?case by (cases φ) (auto intro: SAT_VIO.SAnd)
next
  case SOrL
  then show ?case by (cases φ) (auto intro: SAT_VIO.SOrL)
next
  case SOrR
  then show ?case by (cases φ) (auto intro: SAT_VIO.SOrR)
next
  case SImpR
  then show ?case by (cases φ) (auto intro: SAT_VIO.SImpR)
next
  case SImpL
  then show ?case by (cases φ) (auto intro: SAT_VIO.SImpL)
next
  case SIffSS
  then show ?case by (cases φ) (auto intro: SAT_VIO.SIffSS)
next
  case SIffVV
  then show ?case by (cases φ) (auto intro: SAT_VIO.SIffVV)
next
  case (SExists x z sp)
  with SExists(1)[of v(x := z)] show ?case
    by (cases φ) (auto intro: SAT_VIO.SExists)
next
  case (SForall x part)
  then show ?case
  proof (cases φ)
    case (Forall y ψ)
    show ?thesis unfolding Forall
    proof (intro SAT_VIO.SForall allI)
      fix z
```

```
      let ?sp = lookup_part part z
      from lookup_part_SubsVals[of z part] obtain D where z ∈ D (D, ?sp) ∈ SubsVals part
        by blast
      with SForall(2−) Forall have s_check (v(y := z)) ψ ?sp s_at ?sp = s_at (SForall x part)
        by auto
      then show SAT σ (v(y := z)) (s_at (SForall x part)) ψ
        by (auto simp del: fun_upd_apply dest!: SForall(1)[rotated])
    qed
  qed auto
next
  case (SSince spsi sps)
  then show ?case
  proof (cases φ)
    case (Since φ I ψ)
    show ?thesis
      using SSince(3)
      unfolding Since
    proof (intro SAT_VIO.SSince[of s_at spsi], goal_cases le mem SATψ SATφ)
      case (SATφ k)
      then show ?case
        by (cases k ≤ s_at (hd sps))
          (auto 0 3 simp: Let_def elim: map_setE[of _ _ _ k] intro: SSince(2) dest!: sym[of s_at _ Suc
(s_at _)])
    qed (auto simp: Let_def intro: SSince(1))
  qed auto
next
  case (SOnce i sp)
  then show ?case
  proof (cases φ)
    case (Once I φ)
    show ?thesis
      using SOnce
      unfolding Once
      by (intro SAT_VIO.SOnce[of s_at sp]) (auto simp: Let_def)
  qed auto
next
  case (SEventually i sp)
  then show ?case
  proof (cases φ)
    case (Eventually I φ)
    show ?thesis
      using SEventually
      unfolding Eventually
      by (intro SAT_VIO.SEventually[of _ s_at sp]) (auto simp: Let_def)
  qed auto
next
  case SHistoricallyOut
  then show ?case by (cases φ) (auto intro: SAT_VIO.SHistoricallyOut)
next
  case (SHistorically i li sps)
  then show ?case
  proof (cases φ)
    case (Historically I φ)
    {fix k
      define j where j_def: j ≡ case right I of ∞ ⇒ 0 | enat n ⇒ ETP σ (τ  σ i − n)
      assume k_def: k ≥ j ∧ k ≤ i ∧ k ≤ LTP σ (τ σ i − left I)
      from SHistorically Historically j_def have map: set (map s_at sps) = set [j ..< Suc (LTP_p σ i
I)]
```

```
      by (auto simp: Let_def)
    then have kset: k ∈ set ([j ..< Suc (LTP_p σ i I)]) using j_def k_def by auto
    then obtain x where x: x ∈ set sps  s_at x = k using k_def map
      unfolding set_map set_eq_iff image_iff
      by metis
    then have SAT σ v k φ using SHistorically unfolding Historically
      by (auto simp: Let_def)
  } note * = this
  show ?thesis
    using SHistorically *
    unfolding Historically
    by (auto simp: Let_def intro!: SAT_VIO.SHistorically)
 qed (auto intro: SAT_VIO.intros)
next
 case (SAlways i hi sps)
 then show ?case
 proof (cases φ)
  case (Always I φ)
  obtain n where n_def: right I = enat n
    using SAlways
    by (auto simp: Always split: enat.splits)
  {fix k
    define j where j_def: j ≡ LTP σ (τ σ i + n)
    assume k_def: k ≤ j ∧ k ≥ i ∧ k ≥ ETP σ (τ σ i + left I)
    from SAlways Always j_def have map: set (map s_at sps) = set [(ETP_f σ i I) ..< Suc j]
      by (auto simp: Let_def n_def)
    then have kset: k ∈ set ([(ETP_f σ i I) ..< Suc j]) using k_def j_def by auto
    then obtain x where x: x ∈ set sps s_at x = k using k_def map
      unfolding set_map set_eq_iff image_iff
      by metis
    then have SAT σ v k φ using SAlways unfolding Always
      by (auto simp: Let_def n_def)
  } note * = this
  then show ?thesis
    using SAlways
    unfolding Always
    by (auto simp: Let_def n_def intro: SAT_VIO.SAlways split: if_splits enat.splits)
 qed(auto intro: SAT_VIO.intros)
next
 case (SUntil sps spsi)
 then show ?case
 proof (cases φ)
  case (Until φ I ψ)
  show ?thesis
    using SUntil(3)
    unfolding Until
  proof (intro SAT_VIO.SUntil[of _ s_at spsi], goal_cases le mem SATψ SATφ)
    case (SATφ k)
    then show ?case
      by (cases k ≤ s_at (hd sps))
        (auto 0 3 simp: Let_def elim: map_setE[of _ _ _ k] intro: SUntil(1))
  qed (auto simp: Let_def intro: SUntil(2))
 qed auto
next
 case (SNext sp)
 then show ?case by (cases φ) (auto simp add: Let_def SAT_VIO.SNext)
next
 case (SPrev sp)
```

```
  then show ?case by (cases φ) (auto simp add: Let_def SAT_VIO.SPrev)
next
  case VFF
  then show ?case by (cases φ) (auto intro: SAT_VIO.VFF)
next
  case VPred
  then show ?case by (cases φ) (auto intro: SAT_VIO.VPred)
next
  case VEq_Const
  then show ?case by (cases φ) (auto intro: SAT_VIO.VEq_Const)
next
  case VNeg
  then show ?case by (cases φ) (auto intro: SAT_VIO.VNeg)
next
  case VOr
  then show ?case by (cases φ) (auto intro: SAT_VIO.VOr)
next
  case VAndL
  then show ?case by (cases φ) (auto intro: SAT_VIO.VAndL)
next
  case VAndR
  then show ?case by (cases φ) (auto intro: SAT_VIO.VAndR)
next
  case VImp
  then show ?case by (cases φ) (auto intro: SAT_VIO.VImp)
next
  case VIffSV
  then show ?case by (cases φ) (auto intro: SAT_VIO.VIffSV)
next
  case VIffVS
  then show ?case by (cases φ) (auto intro: SAT_VIO.VIffVS)
next
  case (VExists x part)
  then show ?case
  proof (cases φ)
    case (Exists y ψ)
    show ?thesis unfolding Exists
    proof (intro SAT_VIO.VExists allI)
      fix z
      let ?vp = lookup_part part z
      from lookup_part_SubsVals[of z part] obtain D where z ∈ D (D, ?vp) ∈ SubsVals part
        by blast
      with VExists(2−) Exists have v_check (v(y := z)) ψ ?vp v_at ?vp = v_at (VExists x part)
        by auto
      then show VIO σ (v(y := z)) (v_at (VExists x part)) ψ
        by (auto simp del: fun_upd_apply dest!: VExists(1)[rotated])
    qed
  qed auto
next
  case (VForall x z vp)
  with VForall(1)[of v(x := z)] show ?case
    by (cases φ) (auto intro: SAT_VIO.VForall)
next
  case VOnceOut
  then show ?case by (cases φ) (auto intro: SAT_VIO.VOnceOut)
next
  case (VOnce i li vps)
  then show ?case
```

```
  proof (cases φ)
    case (Once I φ)
    {fix k
      define j where j_def: j ≡ case right I of ∞ ⇒ 0 | enat n ⇒ ETP σ (τ σ i − n)
      assume k_def: k ≥ j ∧ k ≤ i ∧ k ≤ LTP σ (τ σ i − left I)
      from VOnce Once j_def have map: set (map v_at vps) = set [j ..< Suc (LTP_p σ i I)]
        by (auto simp: Let_def)
      then have kset: k ∈ set ([j ..< Suc (LTP_p σ i I)]) using j_def k_def by auto
      then obtain x where x: x ∈ set vps  v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k φ using VOnce unfolding Once
        by (auto simp: Let_def)
    } note ∗ = this
    show ?thesis
      using VOnce ∗
      unfolding Once
      by (auto simp: Let_def intro!: SAT_VIO.VOnce)
  qed (auto intro: SAT_VIO.intros)
next
  case (VEventually i hi vps)
  then show ?case
  proof (cases φ)
    case (Eventually I φ)
    obtain n where n_def: right I = enat n
      using VEventually
      by (auto simp: Eventually split: enat.splits)
    {fix k
      define j where j_def: j ≡ LTP σ (τ σ i + n)
      assume k_def: k ≤ j ∧ k ≥ i ∧ k ≥ ETP σ (τ σ i + left I)
      from VEventually Eventually j_def have map: set (map v_at vps) = set [(ETP_f σ i I) ..< Suc j]
        by (auto simp: Let_def n_def)
      then have kset: k ∈ set ([(ETP_f σ i I) ..< Suc j]) using k_def j_def by auto
      then obtain x where x: x ∈ set vps v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k φ using VEventually unfolding Eventually
        by (auto simp: Let_def n_def)
    } note ∗ = this
    then show ?thesis
      using VEventually
      unfolding Eventually
      by (auto simp: Let_def n_def intro: SAT_VIO.VEventually split: if_splits enat.splits)
  qed(auto intro: SAT_VIO.intros)
next
  case (VHistorically i vp)
  then show ?case
  proof (cases φ)
    case (Historically I φ)
    show ?thesis
      using VHistorically
      unfolding Historically
      by (intro SAT_VIO.VHistorically[of v_at vp]) (auto simp: Let_def)
  qed auto
next
  case (VAlways i vp)
  then show ?case
  proof (cases φ)
```

```
    case (Always I φ)
    show ?thesis
      using VAlways
      unfolding Always
      by (intro SAT_VIO.VAlways[of _ v_at vp]) (auto simp: Let_def)
  qed auto
next
  case VNext
  then show ?case by (cases φ) (auto intro: SAT_VIO.VNext)
next
  case VNextOutR
  then show ?case by (cases φ) (auto intro: SAT_VIO.VNextOutR)
next
  case VNextOutL
  then show ?case by (cases φ) (auto intro: SAT_VIO.VNextOutL)
next
  case VPrev
  then show ?case by (cases φ) (auto intro: SAT_VIO.VPrev)
next
  case VPrevOutR
  then show ?case by (cases φ) (auto intro: SAT_VIO.VPrevOutR)
next
  case VPrevOutL
  then show ?case by (cases φ) (auto intro: SAT_VIO.VPrevOutL)
next
  case VPrevZ
  then show ?case by (cases φ) (auto intro: SAT_VIO.VPrevZ)
next
  case VSinceOut
  then show ?case by (cases φ) (auto intro: SAT_VIO.VSinceOut)
next
  case (VSince i vp vps)
  then show ?case
  proof (cases φ)
    case (Since φ I ψ)
    {fix k
      assume k_def: k ≥ v_at vp ∧ k ≤ i ∧ k ≤ LTP σ (τ σ i - left I)
      from VSince Since have map: set (map v_at vps) = set ([(v_at vp) ..< Suc (LTP_p σ i I)])
        by (auto simp: Let_def)
      then have kset: k ∈ set ([(v_at vp) ..< Suc (LTP_p σ i I)]) using k_def by auto
      then obtain x where x: x ∈ set vps v_at x = k using k_def map kset
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k ψ using VSince unfolding Since
        by (auto simp: Let_def)
    } note * = this
    show ?thesis
      using VSince *
      unfolding Since
      by (auto simp: Let_def split: enat.splits if_splits
          intro!: SAT_VIO.VSince[of _ i v_at vp])
  qed (auto intro: SAT_VIO.intros)
next
  case (VUntil i vps vp)
  then show ?case
  proof (cases φ)
    case (Until φ I ψ)
    {fix k
```

```
      assume k_def: k ≤ v_at vp ∧ k ≥ i ∧ k ≥ ETP σ (τ σ i + left I)
      from VUntil Until have map: set (map v_at vps) = set [(ETP_f σ i I) ..< Suc (v_at vp)]
        by (auto simp: Let_def)
      then have kset: k ∈ set ([(ETP_f σ i I) ..< Suc (v_at vp)]) using k_def by auto
      then obtain x where x: x ∈ set vps v_at x = k using k_def map kset
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k ψ using VUntil unfolding Until
        by (auto simp: Let_def)
    } note * = this
    then show ?thesis
      using VUntil
      unfolding Until
      by (auto simp: Let_def split: enat.splits if_splits
          intro!: SAT_VIO.VUntil)
  qed(auto intro: SAT_VIO.intros)
next
  case (VSinceInf i li vps)
  then show ?case
  proof (cases φ)
    case (Since φ I ψ)
    {fix k
      define j where j_def: j ≡ case right I of ∞ ⇒ 0 | enat n ⇒ ETP σ (τ σ i − n)
      assume k_def: k ≥ j ∧ k ≤ i ∧ k ≤ LTP σ (τ σ i − left I)
      from VSinceInf Since j_def have map: set (map v_at vps) = set [j ..< Suc (LTP_p σ i I)]
        by (auto simp: Let_def)
      then have kset: k ∈ set ([j ..< Suc (LTP_p σ i I)]) using j_def k_def by auto
      then obtain x where x: x ∈ set vps  v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k ψ using VSinceInf unfolding Since
        by (auto simp: Let_def)
    } note * = this
    show ?thesis
      using VSinceInf *
      unfolding Since
      by (auto simp: Let_def intro!: SAT_VIO.VSinceInf)
  qed (auto intro: SAT_VIO.intros)
next
  case (VUntilInf i hi vps)
  then show ?case
  proof (cases φ)
    case (Until φ I ψ)
    obtain n where n_def: right I = enat n
      using VUntilInf
      by (auto simp: Until split: enat.splits)
    {fix k
      define j where j_def: j ≡ LTP σ (τ σ i + n)
      assume k_def: k ≤ j ∧ k ≥ i ∧ k ≥ ETP σ (τ σ i + left I)
      from VUntilInf Until j_def have map: set (map v_at vps) = set [(ETP_f σ i I) ..< Suc j]
        by (auto simp: Let_def n_def)
      then have kset: k ∈ set ([(ETP_f σ i I) ..< Suc j]) using k_def j_def by auto
      then obtain x where x: x ∈ set vps v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO σ v k ψ using VUntilInf unfolding Until
        by (auto simp: Let_def n_def)
    } note * = this
```

```
    then show ?thesis
      using VUntilInf
      unfolding Until
      by (auto simp: Let_def n_def intro: SAT_VIO.VUntilInf split: if_splits enat.splits)
  qed(auto intro: SAT_VIO.intros)
qed

definition compatible X vs v ⟷ (∀x∈X. v x ∈ vs x)

definition compatible_vals X vs = {v. ∀x ∈ X. v x ∈ vs x}

lemma compatible_alt:
  compatible X vs v ⟷ v ∈ compatible_vals X vs
  by (auto simp: compatible_def compatible_vals_def)

lemma compatible_empty_iff: compatible {} vs v ⟷ True
  by (auto simp: compatible_def)

lemma compatible_vals_empty_eq: compatible_vals {} vs = UNIV
  by (auto simp: compatible_vals_def)

lemma compatible_union_iff:
  compatible (X ∪ Y) vs v ⟷ compatible X vs v ∧ compatible Y vs v
  by (auto simp: compatible_def)

lemma compatible_vals_union_eq:
  compatible_vals (X ∪ Y) vs = compatible_vals X vs ∩ compatible_vals Y vs
  by (auto simp: compatible_vals_def)

lemma compatible_antimono:
  compatible X vs v ⟹ Y ⊆ X ⟹ compatible Y vs v
  by (auto simp: compatible_def)

lemma compatible_vals_antimono:
  Y ⊆ X ⟹ compatible_vals X vs ⊆ compatible_vals Y vs
  by (auto simp: compatible_vals_def)

lemma compatible_extensible:
  (∀x. vs x ≠ {}) ⟹ compatible X vs v ⟹ X ⊆ Y ⟹ ∃v'. compatible Y vs v' ∧ (∀x∈X. v x = v' x)
  using some_in_eq[of vs _] by (auto simp: override_on_def compatible_def
      intro: exI[where x=override_on v (λx. SOME y. y ∈ vs x) (Y−X)])

lemmas compatible_vals_extensible = compatible_extensible[unfolded compatible_alt]

primrec mk_values :: (('n, 'd) trm × 'a set) list ⇒ 'a list set
  where mk_values [] = {[]}
  | mk_values (T # Ts) = (case T of
      (v x, X) ⇒
        let terms = map fst Ts in
        if v x ∈ set terms then
          let fst_pos = hd (positions terms (v x)) in (λxs. (xs ! fst_pos) # xs) ` (mk_values Ts)
        else set_Cons X (mk_values Ts)
    | (c a, X) ⇒ set_Cons X (mk_values Ts))

lemma mk_values_nempty:
  {} ∉ set (map snd tXs) ⟹ mk_values tXs ≠ {}
  by (induct tXs)
    (auto simp: set_Cons_def image_iff split: trm.splits if_splits)
```

lemma *mk_values_not_Nil*:
{} ∉ *set* (*map snd tXs*) ⟹ *tXs* ≠ [] ⟹ *vs* ∈ *mk_values tXs* ⟹ *vs* ≠ []
by (*induct tXs*)
(*auto simp*: *set_Cons_def image_iff split*: *trm.splits if_splits*)

lemma *mk_values_nth_cong*: **v** *x* ∈ *set* (*map fst tXs*) ⟹
n ∈ *set* (*positions* (*map fst tXs*) (**v** *x*)) ⟹
m ∈ *set* (*positions* (*map fst tXs*) (**v** *x*)) ⟹
vs ∈ *mk_values tXs* ⟹
vs ! *n* = *vs* ! *m*
proof (*induct tXs arbitrary*: *n m vs x*)
case (*Cons tX tXs*)
show *?case*
proof (*cases n*)
case *0*
then show *?thesis*
proof (*cases m*)
case (*Suc m′*)
with *0* **show** *?thesis*
using *Cons*(*2*−) *Cons.hyps*(*1*)[*of x m′ _ tl vs*] *positions_eq_nil_iff*[*of map fst tXs trm.Var x*]
by (*fastforce split*: *if_splits simp*: *in_set_conv_nth*
Let_def nth_Cons′ gr0_conv_Suc neq_Nil_conv)
qed *simp*
next
case *n*: (*Suc n′*)
then show *?thesis*
proof (*cases m*)
case *0*
with *n* **show** *?thesis*
using *Cons*(*2*−) *Cons.hyps*(*1*)[*of x _ n′ tl vs*] *positions_eq_nil_iff*[*of map fst tXs trm.Var x*]
by (*fastforce split*: *if_splits simp*: *in_set_conv_nth*
Let_def nth_Cons′ gr0_conv_Suc neq_Nil_conv)
next
case (*Suc m′*)
with *n* **show** *?thesis*
using *Cons*(*1*)[*of x n′ m′ tl vs*] *Cons*(*2*−)
by (*fastforce simp*: *set_Cons_def set_positions_eq split*: *trm.splits if_splits*)
qed
qed
qed *simp*

definition *mk_values_subset p tXs X*
⟷ (*let* (*fintXs*, *inftXs*) = *partition* (λ*tX*. *finite* (*snd tX*)) *tXs in*
if inftXs = [] *then* {*p*} × *mk_values tXs* ⊆ *X*
else let inf_dups = *filter* (λ*tX*. (*fst tX*) ∈ *set* (*map fst fintXs*)) *inftXs in*
if inf_dups = [] *then* (*if finite X then False else Code.abort STR ″subset on infinite subset″* (λ_. {*p*} × *mk_values tXs* ⊆ *X*))
else if list_all (λ*tX*. *Max* (*set* (*positions tXs tX*)) < *Max* (*set* (*positions* (*map fst tXs*) (*fst tX*)))) *inf_dups*
then {*p*} × *mk_values tXs* ⊆ *X*
else (*if finite X then False else Code.abort STR ″subset on infinite subset″* (λ_. {*p*} × *mk_values tXs* ⊆ *X*)))

lemma *mk_values_nemptyI*: ∀ *tX* ∈ *set tXs*. *snd tX* ≠ {} ⟹ *mk_values tXs* ≠ {}
by (*induct tXs*)
(*auto simp*: *Let_def set_Cons_eq split*: *prod.splits trm.splits*)

```
lemma infinite_mk_values1: ∀ tX ∈ set tXs. snd tX ≠ {} ⟹ tY ∈ set tXs ⟹
  ∀ Y. (fst tY, Y) ∈ set tXs ⟶ infinite Y ⟹ infinite (mk_values tXs)
proof (induct tXs arbitrary: tY)
  case (Cons tX tXs)
  show ?case
    unfolding Let_def image_iff mk_values.simps split_beta
      trm.split[of infinite] if_split[of infinite]
  proof (safe, goal_cases var_in var_out const)
    case (var_in x)
    hence ∀ tX∈set tXs. snd tX ≠ {}
      by (simp add: Cons.prems(1))
    moreover have ∀ Z. (trm.Var x, Z) ∈ set tXs ⟶ infinite Z
      using Cons.prems(2,3) var_in
      by (cases tY ∈ set tXs; clarsimp)
        (metis (no_types, lifting) Cons.hyps Cons.prems(1)
          finite_imageD inj_on_def list.inject list.set_intros(2))
    ultimately have infinite (mk_values tXs)
      using Cons.hyps var_in
      by auto
    moreover have inj (λxs. xs ! hd (positions (map fst tXs) (trm.Var x)) # xs)
      by (clarsimp simp: inj_on_def)
    ultimately show ?case
      using var_in(3) finite_imageD inj_on_subset
      by fastforce
  next
    case (var_out x)
    hence infinite (snd tX)
      using Cons
      by (metis infinite_set_ConsI(2) insert_iff list.simps(15) prod.collapse)
    moreover have mk_values tXs ≠ {}
      using Cons.prems
      by (auto intro!: mk_values_nemptyI)
    then show ?case
      using Cons var_out infinite_set_ConsI(1)[OF ‹mk_values tXs ≠ {}› ‹infinite (snd tX)›]
      by auto
  next
    case (const c)
    hence infinite (snd tX)
      using Cons
      by (metis infinite_set_ConsI(2) insert_iff list.simps(15) prod.collapse)
    moreover have mk_values tXs ≠ {}
      using Cons.prems
      by (auto intro!: mk_values_nemptyI)
    then show ?case
      using Cons const infinite_set_ConsI(1)[OF ‹mk_values tXs ≠ {}› ‹infinite (snd tX)›]
      by auto
  qed
qed simp

lemma infinite_mk_values2: ∀ tX∈set tXs. snd tX ≠ {} ⟹
  tY ∈ set tXs ⟹ infinite (snd tY) ⟹
  Max (set (positions tXs tY)) ≥ Max (set (positions (map fst tXs) (fst tY))) ⟹
  infinite (mk_values tXs)
proof (induct tXs arbitrary: tY)
  case (Cons tX tXs)
  hence obs1: ∀ tX∈set tXs. snd tX ≠ {}
    by (simp add: Cons.prems(1))
  note IH = Cons.hyps[OF obs1 _ ‹infinite (snd tY)›]
```

```
  have obs2: tY ∈ set tXs ⟹
    Max (set (positions (map fst tXs) (fst tY))) ≤ Max (set (positions tXs tY))
    using Cons.prems(4) unfolding list.map
    by (metis Max_set_positions_Cons_tl Suc_le_mono positions_eq_nil_iff set_empty2 subset_empty
subset_positions_map_fst)
  show ?case
    unfolding Let_def image_iff mk_values.simps split_beta
      trm.split[of infinite] if_split[of infinite]
  proof (safe, goal_cases var_in var_out const)
    case (var_in x)
    then show ?case
    proof (cases tY ∈ set tXs)
      case True
      hence infinite ((λXs. Xs ! hd (positions (map fst tXs) (trm.Var x)) # Xs) ‘ mk_values tXs)
        using IH[OF True obs2[OF True]] finite_imageD inj_on_def by blast
      then show False
        using var_in by blast
    next
      case False
      have Max (set (positions (map fst (tX # tXs)) (fst tY)))
      = Suc (Max (set (positions (map fst tXs) (fst tY))))
        using Cons.prems var_in
        by (simp only: list.map(2))
          (subst Max_set_positions_Cons_tl; force simp: image_iff)
      moreover have tY ∉ set tXs ⟹ Max (set (positions (tX # tXs) tY)) = (0::nat)
        using Cons.prems Max_set_positions_Cons_hd by fastforce
      ultimately show False
        using Cons.prems(4) False
        by linarith
    qed
  next
    case (var_out x)
    then show ?case
    proof (cases tY ∈ set tXs)
      case True
      hence infinite (mk_values tXs)
        using IH obs2 by blast
      hence infinite (set_Cons (snd tX) (mk_values tXs))
        by (metis Cons.prems(1) infinite_set_ConsI(2) list.set_intros(1))
      then show False
        using var_out by blast
    next
      case False
      hence snd tY = snd tX and infinite (snd tX)
        using var_out Cons.prems
        by auto
      hence infinite (set_Cons (snd tX) (mk_values tXs))
        by (simp add: infinite_set_ConsI(1) mk_values_nemptyI obs1)
      then show False
        using var_out by blast
    qed
  next
    case (const c)
    then show ?case
    proof (cases tY ∈ set tXs)
      case True
      hence infinite (mk_values tXs)
        using IH obs2 by blast
```

```
    hence infinite (set_Cons (snd tX) (mk_values tXs))
      by (metis Cons.prems(1) infinite_set_ConsI(2) list.set_intros(1))
    then show False
      using const by blast
  next
    case False
    hence infinite (set_Cons (snd tX) (mk_values tXs))
      using const Cons.prems
      by (simp add: infinite_set_ConsI(1) mk_values_nemptyI obs1)
    then show False
      using const by blast
  qed
 qed
qed simp

lemma mk_values_subset_iff: ∀ tX ∈ set tXs. snd tX ≠ {} ⟹
   mk_values_subset p tXs X ⟷ {p} × mk_values tXs ⊆ X
 unfolding mk_values_subset_def image_iff Let_def comp_def split_beta if_split_eq1
   partition_filter1 partition_filter2 o_def set_map set_filter filter_filter bex_simps
proof safe
 assume ∀ tX∈set tXs. snd tX ≠ {} and finite X
   and filter1: filter (λxy. infinite (snd xy) ∧ (∃ ab. (ab ∈ set tXs ∧ finite (snd ab)) ∧ fst xy = fst ab))
tXs = []
   and filter2: filter (λx. infinite (snd x)) tXs ≠ []
 then obtain tY where tY ∈ set tXs and infinite (snd tY)
   by (meson filter_False)
 moreover have ∀ Y. (fst tY, Y) ∈ set tXs ⟶ infinite Y
   using filter1 calculation
   by (auto simp: filter_empty_conv)
 ultimately have infinite (mk_values tXs)
   using infinite_mk_values1[OF ‹∀ tX∈set tXs. snd tX ≠ {}›]
   by auto
 hence infinite ({p} × mk_values tXs)
   using finite_cartesian_productD2 by auto
 thus {p} × mk_values tXs ⊆ X ⟹ False
   using ‹finite X›
   by (simp add: finite_subset)
next
 assume ∀ tX∈set tXs. snd tX ≠ {}
   and finite X
   and ex_dupl_inf: ¬ list_all (λtX. Max (set (positions tXs tX))
   < Max (set (positions (map fst tXs) (fst tX))))
      (filter (λxy. infinite (snd xy) ∧ (∃ ab. (ab ∈ set tXs ∧ finite (snd ab)) ∧ fst xy = fst ab)) tXs)
   and filter: filter (λx. infinite (snd x)) tXs ≠ []
 then obtain tY and Z where tY ∈ set tXs
   and infinite (snd tY)
   and (fst tY, Z) ∈ set tXs
   and finite Z
   and Max (set (positions tXs tY)) ≥ Max (set (positions (map fst tXs) (fst tY)))
   by (auto simp: list_all_iff)
 hence infinite (mk_values tXs)
   using infinite_mk_values2[OF ‹∀ tX∈set tXs. snd tX ≠ {}› ‹tY ∈ set tXs›]
   by auto
 hence infinite ({p} × mk_values tXs)
   using finite_cartesian_productD2 by auto
 thus {p} × mk_values tXs ⊆ X ⟹ False
   using ‹finite X›
   by (simp add: finite_subset)
```

qed *auto*

lemma *mk_values_sound*: *cs* ∈ *mk_values* (*vs*⦃*ts*⦄) ⟹
∃ *v*∈*compatible_vals* (*fv* (*r* † *ts*)) *vs*. *cs* = *v*⟦*ts*⟧
proof (*induct ts arbitrary*: *cs vs*)
case (*Cons t ts*)
show *?case*
proof(*cases t*)
case (*Var x*)
let *?Ts* = *vs*⦃*ts*⦄
have *vs*⦃(*t* # *ts*)⦄ = (**v** *x*, *vs x*) # *?Ts*
using *Var* **by** (*simp add*: *eval_trms_set_def*)
show *?thesis*
proof (*cases* **v** *x* ∈ *set ts*)
case *True*
then obtain *n* **where** *n_in*: *n* ∈ *set* (*positions ts* (**v** *x*))
and *nth_n*: *ts* ! *n* = **v** *x*
by (*meson fst_pos_in_positions nth_fst_pos*)
hence *n_in'*: *n* ∈ *set* (*positions* (*map fst ?Ts*) (**v** *x*))
by (*induct ts arbitrary*: *n*)
(*auto simp*: *eval_trms_set_def split*: *if_splits*)
have *key*: **v** *x* ∈ *set* (*map fst ?Ts*)
using *True* **by** (*induct ts*)
(*auto simp*: *eval_trms_set_def*)
then obtain *a as*
where *as_head*: *as* ! (*hd* (*positions* (*map fst ?Ts*) (**v** *x*))) = *a*
and *as_tail*: *as* ∈ *mk_values ?Ts*
and *as_shape*: *cs* = *a* # *as*
using *Cons*(*2*)
by (*clarsimp simp add*: *eval_trms_set_def Var image_iff*)
obtain *v* **where** *v_hyps*: *v* ∈ *compatible_vals* (*fv* (*r* † *ts*)) *vs*
as = *v*⟦*ts*⟧
using *Cons*(*1*)[*OF as_tail*] **by** *blast*
hence *as'_nth*: *as* ! *n* = *v x*
using *nth_n positions_length*[*OF n_in*]
by (*simp add*: *eval_trms_def*)
have *evals_neq_Nil*: *?Ts* ≠ []
using *key* **by** *auto*
moreover have *positions* (*map fst ?Ts*) (**v** *x*) ≠ []
using *positions_eq_nil_iff*[*of map fst ?Ts* **v** *x*] *key*
by *fastforce*
ultimately have *as_hyp*: *a* = *as* ! *n*
using *mk_values_nth_cong*[*OF key hd_in_set n_in' as_tail*] *as_head* **by** *blast*
thus *?thesis*
using *Var as_shape True v_hyps as'_nth*
by (*auto simp*: *compatible_vals_def eval_trms_def intro*!: *exI*[*of* _ *v*])
next
case *False*
hence ∗: **v** *x* ∉ *set* (*map fst ?Ts*)
proof (*induct ts arbitrary*: *x*)
case (*Cons a ts*)
then show *?case*
by (*cases a*) (*auto simp*: *eval_trms_set_def*)
qed (*simp add*: *eval_trms_set_def*)
from *Cons*(*2*) *False* **show** *?thesis*
unfolding *set_Cons_def eval_trms_set_def Let_def list.simps Var*
∗[*THEN eq_False*[*THEN iffD2*], *unfolded eval_trms_set_def*] *if_False*
mk_values.simps eval_trm_set.simps prod.case trm.case mem_Collect_eq

```
      proof (elim exE conjE, goal_cases)
        case (1 a as)
        with Cons(1)[of as vs] obtain v where v ∈ compatible_vals (fv (r † ts)) vs as = v⟦ts⟧
          by (auto simp: eval_trms_set_def)
        with 1 show ?case
          by (auto simp: eval_trms_set_def eval_trms_def compatible_vals_def in_fv_trm_conv
            intro!: exI[of _ v(x := a)] eval_trm_fv_cong)
      qed
    qed
  next
    case (Const c)
    then show ?thesis
      using Cons(1)[of _ vs] Cons(2)
      by (auto simp: eval_trms_set_def set_Cons_def
          eval_trms_def compatible_def)
  qed
qed (simp add: eval_trms_set_def eval_trms_def compatible_vals_def)

lemma fst_eval_trm_set[simp]:
  fst (vs⦃t⦄) = t
  by (cases t; clarsimp)

lemma mk_values_complete: cs = v⟦ts⟧ ⟹
  v ∈ compatible_vals (fv (r † ts)) vs ⟹
  cs ∈ mk_values (vs⦃ts⦄)
proof (induct ts arbitrary: v cs vs)
  case (Cons t ts)
  then obtain a as
    where a_def: a = v⟦t⟧
      and as_def: as = v⟦ts⟧
      and cs_cons: cs = a # as
    by (auto simp: eval_trms_def)
  have compat_v_vs: v ∈ compatible_vals (fv (r † ts)) vs
    using Cons.prems
    by (auto simp: compatible_vals_def)
  hence mk_values_ts: as ∈ mk_values (vs⦃ts⦄)
    using Cons.hyps[OF as_def]
    unfolding eval_trms_set_def by blast
  show ?case
  proof (cases t)
    case (Var x)
    then show ?thesis
    proof (cases v x ∈ set ts)
      case True
      then obtain n
        where n_head: n = hd (positions ts (v x))
          and n_in: n ∈ set (positions ts (v x))
          and nth_n: ts ! n = v x
        by (simp_all add: hd_positions_eq_fst_pos nth_fst_pos fst_pos_in_positions)
      hence n_in': n = hd (positions (map fst (vs⦃ts⦄)) (v x))
        by (clarsimp simp: eval_trms_set_def o_def)
      moreover have as ! n = a
        using a_def as_def nth_n Var n_in True positions_length
        by (fastforce simp: eval_trms_def)
      moreover have v x ∈ set (map fst (vs⦃ts⦄))
        using True by (induct ts)
          (auto simp: eval_trms_set_def)
      ultimately show ?thesis
```

```
      using mk_values_ts cs_cons
      by (clarsimp simp: eval_trms_set_def Var image_iff)
    next
      case False
      then show ?thesis
        using Var cs_cons mk_values_ts Cons.prems a_def
        by (clarsimp simp: eval_trms_set_def image_iff
            set_Cons_def compatible_vals_def split: trm.splits)
    qed
  next
    case (Const a)
    then show ?thesis
      using cs_cons mk_values_ts Cons.prems a_def
      by (clarsimp simp: eval_trms_set_def image_iff
          set_Cons_def compatible_vals_def split: trm.splits)
  qed
qed (simp add: compatible_vals_def
    eval_trms_set_def eval_trms_def)

definition mk_values_subset_Compl r vs ts i = ({r} × mk_values (vs⦇ts⦈) ⊆ − Γ σ i)

fun check_values where
  check_values _ _ _ None = None
| check_values vs (c c # ts) (u # us) f = (if c = u then check_values vs ts us f else None)
| check_values vs (v x # ts) (u # us) (Some v) = (if u ∈ vs x ∧ (v x = Some u ∨ v x = None) then
check_values vs ts us (Some (v(x ↦ u))) else None)
| check_values vs [] [] f = f
| check_values _ _ _ _ = None

lemma mk_values_alt:
  mk_values (vs⦇ts⦈) =
   {cs. ∃ v∈compatible_vals (⋃(fv_trm ‘ set ts)) vs. cs = v⟦ts⟧}
  by (auto dest!: mk_values_sound intro: mk_values_complete)

lemma check_values_neq_NoneI:
  assumes v ∈ compatible_vals (⋃ (fv_trm ‘ set ts) − dom f) vs ⋀x y. f x = Some y ⟹ y ∈ vs x
  shows check_values vs ts ((λx. case f x of None ⇒ v x | Some y ⇒ y)⟦ts⟧) (Some f) ≠ None
  using assms
proof (induct ts arbitrary: f)
  case (Cons t ts)
  then show ?case
  proof (cases t)
    case (Var x)
    show ?thesis
    proof (cases f x)
      case None
      with Cons(2) Var have v_in[simp]: v x ∈ vs x
        by (auto simp: compatible_vals_def)
      from Cons(2) have v ∈ compatible_vals (⋃ (fv_trm ‘ set ts) − dom (f(x ↦ v x))) vs
        by (auto simp: compatible_vals_def)
      then have check_values vs ts
        ((λz. case (f(x ↦ v x)) z of None ⇒ v z | Some y ⇒ y)⟦ts⟧)
        (Some (f(x ↦ v x))) ≠ None
        using Cons(3) None Var
        by (intro Cons(1)) (auto simp: compatible_vals_def split: if_splits)
      then show ?thesis
        by (elim eq_neq_eq_imp_neq[OF _ _ refl, rotated])
          (auto simp add: eval_trms_def compatible_vals_def Var None split: if_splits option.splits
```

```
        intro!: arg_cong2[of _ _ _ _ check_values vs ts] eval_trm_fv_cong)
    next
      case (Some y)
      with Cons(1)[of f] Cons(2−) Var fun_upd_triv[of f x] show ?thesis
        by (auto simp: domI eval_trms_def compatible_vals_def split: option.splits)
    qed
  next
    case (Const c)
    with Cons show ?thesis
      by (auto simp: eval_trms_def compatible_vals_def split: option.splits)
  qed
qed (simp add: eval_trms_def)

lemma check_values_eq_NoneI:
  ∀ v∈compatible_vals (⋃ (fv_trm ‘ set ts) − dom f) vs. us ≠ (λx. case f x of None ⇒ v x | Some y ⇒
y)⟦ts⟧ ⟹
  check_values vs ts us (Some f) = None
proof (induct vs ts us Some f arbitrary: f rule: check_values.induct)
  case (3 vs x ts u us v)
  show ?case
    unfolding check_values.simps if_split_eq1 simp_thms
  proof (intro impI 3(1), safe, goal_cases)
    case (1 v')
    with 3(2) show ?case
      by (auto simp: insert_dom domI eval_trms_def intro!: eval_trm_fv_cong split: if_splits dest!:
bspec[of _ _ v'])
  next
    case (2 v')
    with 3(2) show ?case
     by (auto simp: insert_dom domI compatible_vals_def eval_trms_def intro!: eval_trm_fv_cong split:
if_splits option.splits dest!: spec[of _ v'(x := u)])
  qed
qed (auto simp: compatible_vals_def eval_trms_def)

lemma mk_values_subset_Compl_code[code]:
  mk_values_subset_Compl r vs ts i = (∀ (q, us) ∈ Γ σ i. q ≠ r ∨ check_values vs ts us (Some Map.empty)
= None)
  unfolding mk_values_subset_Compl_def eval_trms_set_def[symmetric] mk_values_alt
proof (safe, goal_cases)
  case (1 _ us y)
  then show ?case
    by (auto simp: subset_eq check_values_eq_NoneI[where f=Map.empty, simplified] dest!: spec[of _
us])
qed (auto simp: subset_eq  dest!: check_values_neq_NoneI[where f=Map.empty, simplified])
```

8.2 Executable Variant of the Checker

```
fun s_check_exec :: ('n, 'd) envset ⇒ ('n, 'd) formula ⇒ ('n, 'd) sproof ⇒ bool
and v_check_exec :: ('n, 'd) envset ⇒ ('n, 'd) formula ⇒ ('n, 'd) vproof ⇒ bool where
  s_check_exec vs f p = (case (f, p) of
    (⊤, STT i) ⇒ True
  | (r † ts, SPred i s ts') ⇒
    (r = s ∧ ts = ts' ∧ mk_values_subset r (vs⦇ts⦈) (Γ σ i))
  | (x ≈ c, SEq_Const i x' c') ⇒
    (c = c' ∧ x = x' ∧ vs x = {c})
  | (¬F φ, SNeg vp) ⇒ v_check_exec vs φ vp
  | (φ ∨F ψ, SOrL sp1) ⇒ s_check_exec vs φ sp1
  | (φ ∨F ψ, SOrR sp2) ⇒ s_check_exec vs ψ sp2
```

```
  | (φ ∧_F ψ, SAnd sp1 sp2) ⇒ s_check_exec vs φ sp1 ∧ s_check_exec vs ψ sp2 ∧ s_at sp1 = s_at sp2
  | (φ ⟶_F ψ, SImpL vp1) ⇒ v_check_exec vs φ vp1
  | (φ ⟶_F ψ, SImpR sp2) ⇒ s_check_exec vs ψ sp2
  | (φ ⟷_F ψ, SIffSS sp1 sp2) ⇒ s_check_exec vs φ sp1 ∧ s_check_exec vs ψ sp2 ∧ s_at sp1 = s_at
sp2
  | (φ ⟷_F ψ, SIffVV vp1 vp2) ⇒ v_check_exec vs φ vp1 ∧ v_check_exec vs ψ vp2 ∧ v_at vp1 = v_at
vp2
  | (∃_F x.  φ, SExists y val sp) ⇒ (x = y ∧ s_check_exec (vs (x := {val})) φ sp)
  | (∀_F x.  φ, SForall y sp_part) ⇒ (let i = s_at (part_hd sp_part)
      in x = y ∧ (∀ (sub, sp) ∈ SubsVals sp_part. s_at sp = i ∧ s_check_exec (vs (x := sub)) φ sp))
  | (Y I φ, SPrev sp) ⇒
    (let j = s_at sp; i = s_at (SPrev sp) in
    i = j+1 ∧ mem (Δ σ i) I ∧ s_check_exec vs φ sp)
  | (X I φ, SNext sp) ⇒
    (let j = s_at sp; i = s_at (SNext sp) in
    j = i+1 ∧ mem (Δ σ j) I ∧ s_check_exec vs φ sp)
  | (P I φ, SOnce i sp) ⇒
    (let j = s_at sp in
    j ≤ i ∧ mem (τ σ i − τ σ j) I ∧ s_check_exec vs φ sp)
  | (F I φ, SEventually i sp) ⇒
    (let j = s_at sp in
    j ≥ i ∧ mem (τ σ j − τ σ i) I ∧ s_check_exec vs φ sp)
  | (H I φ, SHistoricallyOut i) ⇒
    τ σ i < τ σ 0 + left I
  | (H I φ, SHistorically i li sps) ⇒
    (li = (case right I of ∞ ⇒ 0 | enat b ⇒ ETP σ (τ σ i − b))
    ∧ τ σ 0 + left I ≤ τ σ i
    ∧ map s_at sps = [li ..< (LTP_p σ i I) + 1]
    ∧ (∀ sp ∈ set sps. s_check_exec vs φ sp))
  | (G I φ, SAlways i hi sps) ⇒
    (hi = (case right I of enat b ⇒ LTP_f σ i b)
    ∧ right I ≠ ∞
    ∧ map s_at sps = [(ETP_f σ i I) ..< hi + 1]
    ∧ (∀ sp ∈ set sps. s_check_exec vs φ sp))
  | (φ S I ψ, SSince sp2 sp1s) ⇒
    (let i = s_at (SSince sp2 sp1s); j = s_at sp2 in
    j ≤ i ∧ mem (τ σ i − τ σ j) I
    ∧ map s_at sp1s = [j+1 ..< i+1]
    ∧ s_check_exec vs ψ sp2
    ∧ (∀ sp1 ∈ set sp1s. s_check_exec vs φ sp1))
  | (φ U I ψ, SUntil sp1s sp2) ⇒
    (let i = s_at (SUntil sp1s sp2); j = s_at sp2 in
    j ≥ i ∧ mem (τ σ j − τ σ i) I
    ∧ map s_at sp1s = [i ..< j] ∧ s_check_exec vs ψ sp2
    ∧ (∀ sp1 ∈ set sp1s. s_check_exec vs φ sp1))
  | ( _ , _) ⇒ False)
| v_check_exec vs f p = (case (f, p) of
    (⊥, VFF i) ⇒ True
  | (r † ts, VPred i pred ts') ⇒
    (r = pred ∧ ts = ts' ∧ mk_values_subset_Compl r vs ts i)
  | (x ≈ c, VEq_Const i x' c') ⇒
    (c = c' ∧ x = x' ∧ c ∉ vs x)
  | (¬_F φ, VNeg sp) ⇒ s_check_exec vs φ sp
  | (φ ∨_F ψ, VOr vp1 vp2) ⇒ v_check_exec vs φ vp1 ∧ v_check_exec vs ψ vp2 ∧ v_at vp1 = v_at vp2
  | (φ ∧_F ψ, VAndL vp1) ⇒ v_check_exec vs φ vp1
  | (φ ∧_F ψ, VAndR vp2) ⇒ v_check_exec vs ψ vp2
  | (φ ⟶_F ψ, VImp sp1 vp2) ⇒ s_check_exec vs φ sp1 ∧ v_check_exec vs ψ vp2 ∧ s_at sp1 = v_at
vp2
```

$|\ (\varphi \longleftrightarrow_F \psi,\ VIffSV\ sp1\ vp2) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1 \wedge v\_check\_exec\ vs\ \psi\ vp2 \wedge s\_at\ sp1 = v\_at\ vp2$

$|\ (\varphi \longleftrightarrow_F \psi,\ VIffVS\ vp1\ sp2) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1 \wedge s\_check\_exec\ vs\ \psi\ sp2 \wedge v\_at\ vp1 = s\_at\ sp2$

$|\ (\exists_F x.\ \varphi,\ VExists\ y\ vp\_part) \Rightarrow (let\ i = v\_at\ (part\_hd\ vp\_part)$
$\quad in\ x = y \wedge (\forall\,(sub,\ vp) \in SubsVals\ vp\_part.\ v\_at\ vp = i \wedge v\_check\_exec\ (vs\ (x := sub))\ \varphi\ vp))$

$|\ (\forall_F x.\ \varphi,\ VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v\_check\_exec\ (vs\ (x := \{val\}))\ \varphi\ vp)$

$|\ (\mathbf{Y}\ I\ \varphi,\ VPrev\ vp) \Rightarrow$
$(let\ j = v\_at\ vp;\ i = v\_at\ (VPrev\ vp)\ in$
$i = j{+}1 \wedge v\_check\_exec\ vs\ \varphi\ vp)$

$|\ (\mathbf{Y}\ I\ \varphi,\ VPrevZ) \Rightarrow True$

$|\ (\mathbf{Y}\ I\ \varphi,\ VPrevOutL\ i) \Rightarrow$
$i > 0 \wedge \Delta\ \sigma\ i < left\ I$

$|\ (\mathbf{Y}\ I\ \varphi,\ VPrevOutR\ i) \Rightarrow$
$i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I$

$|\ (\mathbf{X}\ I\ \varphi,\ VNext\ vp) \Rightarrow$
$(let\ j = v\_at\ vp;\ i = v\_at\ (VNext\ vp)\ in$
$j = i{+}1 \wedge v\_check\_exec\ vs\ \varphi\ vp)$

$|\ (\mathbf{X}\ I\ \varphi,\ VNextOutL\ i) \Rightarrow$
$\Delta\ \sigma\ (i{+}1) < left\ I$

$|\ (\mathbf{X}\ I\ \varphi,\ VNextOutR\ i) \Rightarrow$
$enat\ (\Delta\ \sigma\ (i{+}1)) > right\ I$

$|\ (\mathbf{P}\ I\ \varphi,\ VOnceOut\ i) \Rightarrow$
$\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$

$|\ (\mathbf{P}\ I\ \varphi,\ VOnce\ i\ li\ vps) \Rightarrow$
$(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b)$
$\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
$\wedge\ map\ v\_at\ vps = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1]$
$\wedge\ (\forall\ vp \in set\ vps.\ v\_check\_exec\ vs\ \varphi\ vp))$

$|\ (\mathbf{F}\ I\ \varphi,\ VEventually\ i\ hi\ vps) \Rightarrow$
$(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
$\wedge\ map\ v\_at\ vps = [(ETP\_f\ \sigma\ i\ I)\ ..<\ hi + 1]$
$\wedge\ (\forall\ vp \in set\ vps.\ v\_check\_exec\ vs\ \varphi\ vp))$

$|\ (\mathbf{H}\ I\ \varphi,\ VHistorically\ i\ vp) \Rightarrow$
$(let\ j = v\_at\ vp\ in$
$j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v\_check\_exec\ vs\ \varphi\ vp)$

$|\ (\mathbf{G}\ I\ \varphi,\ VAlways\ i\ vp) \Rightarrow$
$(let\ j = v\_at\ vp$
$in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v\_check\_exec\ vs\ \varphi\ vp)$

$|\ (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceOut\ i) \Rightarrow$
$\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$

$|\ (\varphi\ \mathbf{S}\ I\ \psi,\ VSince\ i\ vp1\ vp2s) \Rightarrow$
$(let\ j = v\_at\ vp1\ in$
$(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b \leq j) \wedge j \leq i$
$\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
$\wedge\ map\ v\_at\ vp2s = [j\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1] \wedge v\_check\_exec\ vs\ \varphi\ vp1$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$

$|\ (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceInf\ i\ li\ vp2s) \Rightarrow$
$(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b)$
$\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
$\wedge\ map\ v\_at\ vp2s = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1]$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$

$|\ (\varphi\ \mathbf{U}\ I\ \psi,\ VUntil\ i\ vp2s\ vp1) \Rightarrow$
$(let\ j = v\_at\ vp1\ in$
$(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP\_f\ \sigma\ i\ b) \wedge i \leq j$
$\wedge\ map\ v\_at\ vp2s = [ETP\_f\ \sigma\ i\ I\ ..<\ j + 1] \wedge v\_check\_exec\ vs\ \varphi\ vp1$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$

$|\ (\varphi\ \mathbf{U}\ I\ \psi,\ VUntilInf\ i\ hi\ vp2s) \Rightarrow$

```
    (hi = (case right I of enat b ⇒ LTP_f σ i b) ∧ right I ≠ ∞
    ∧ map v_at vp2s = [ETP_f σ i I ..< hi + 1]
    ∧ (∀ vp2 ∈ set vp2s. v_check_exec vs ψ vp2))
  | ( _ , _) ⇒ False)

declare s_check_exec.simps[simp del] v_check_exec.simps[simp del]
simps_of_case s_check_exec_simps[simp, code]: s_check_exec.simps[unfolded prod.case] (splits: for-
mula.split sproof.split)
simps_of_case v_check_exec_simps[simp, code]: v_check_exec.simps[unfolded prod.case] (splits: for-
mula.split vproof.split)


lemma check_fv_cong:
  assumes ∀ x ∈ fv φ. v x = v' x
  shows s_check v φ sp ⟷ s_check v' φ sp v_check v φ vp ⟷ v_check v' φ vp
  using assms
proof (induct φ arbitrary: v v' sp vp)
  case TT
  {
    case 1
    then show ?case
      by (cases sp) auto
  next
    case 2
    then show ?case
      by (cases vp) auto
  }
next
  case FF
  {
    case 1
    then show ?case
      by (cases sp) auto
  next
    case 2
    then show ?case
      by (cases vp) auto
  }
next
  case (Pred p ts)
  {
    case 1
    with Pred show ?case using eval_trms_fv_cong[of ts v v']
      by (cases sp) auto
  next
    case 2
    with Pred show ?case using eval_trms_fv_cong[of ts v v']
      by (cases vp) auto
  }
  case (Eq_Const x c)
  {
    case 1
    then show ?case
      by (cases sp) auto
  next
    case 2
    then show ?case
      by (cases vp) auto
```

```
  }
next
  case (Neg φ)
  {
    case 1
    with Neg[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Neg[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Or φ1 φ2)
  {
    case 1
    with Or[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Or[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (And φ1 φ2)
  {
    case 1
    with And[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with And[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Imp φ1 φ2)
  {
    case 1
    with Imp[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Imp[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Iff φ1 φ2)
  {
    case 1
    with Iff[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Iff[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Exists x φ)
```

```
  {
    case 1
    with Exists[of v(x := z) v'(x := z) for z] show ?case
      by (cases sp) (auto simp: fun_upd_def)
  next
    case 2
    with Exists[of v(x := z) v'(x := z) for z] show ?case
      by (cases vp) (auto simp: fun_upd_def)
  }
next
  case (Forall x φ)
  {
    case 1
    with Forall[of v(x := z) v'(x := z) for z] show ?case
      by (cases sp) (auto simp: fun_upd_def)
  next
    case 2
    with Forall[of v(x := z) v'(x := z) for z] show ?case
      by (cases vp) (auto simp: fun_upd_def)
  }
next
  case (Prev I φ)
  {
    case 1
    with Prev[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Prev[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Next I φ)
  {
    case 1
    with Next[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Next[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Once I φ)
  {
    case 1
    with Once[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Once[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Historically I φ)
  {
    case 1
    with Historically[of v v'] show ?case
```

```
    by (cases sp) auto
  next
    case 2
    with Historically[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Eventually I φ)
  {
    case 1
    with Eventually[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Eventually[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Always I φ)
  {
    case 1
    with Always[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Always[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Since φ1 I φ2)
  {
    case 1
    with Since[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Since[of v v'] show ?case
      by (cases vp) auto
  }
next
  case (Until φ1 I φ2)
  {
    case 1
    with Until[of v v'] show ?case
      by (cases sp) auto
  next
    case 2
    with Until[of v v'] show ?case
      by (cases vp) auto
  }
qed

lemma s_check_fun_upd_notin[simp]:
  x ∉ fv φ ⟹ s_check (v(x := t)) φ sp = s_check v φ sp
  by (rule check_fv_cong) auto
lemma v_check_fun_upd_notin[simp]:
  x ∉ fv φ ⟹ v_check (v(x := t)) φ sp = v_check v φ sp
  by (rule check_fv_cong) auto
```

lemma *SubsVals_nonempty*: $(X,\ t) \in SubsVals\ part \Longrightarrow X \neq \{\}$
by *transfer* (*auto simp*: *partition_on_def image_iff*)

lemma *compatible_vals_nonemptyI*: $\forall x.\ vs\ x \neq \{\} \Longrightarrow compatible\_vals\ A\ vs \neq \{\}$
by (*auto simp*: *compatible_vals_def intro*!: *bchoice*)

lemma *compatible_vals_fun_upd*: *compatible_vals A* (*vs*($x := X$)) =
(*if* $x \in A$ *then* $\{v \in compatible\_vals\ (A - \{x\})\ vs.\ v\ x \in X\}$ *else compatible_vals A vs*)
unfolding *compatible_vals_def*
by *auto*

lemma *fun_upd_in_compatible_vals*: $v \in compatible\_vals\ (A - \{x\})\ vs \Longrightarrow v(x := t) \in compatible\_vals\ (A - \{x\})\ vs$
unfolding *compatible_vals_def*
by *auto*

lemma *fun_upd_in_compatible_vals_in*: $v \in compatible\_vals\ (A - \{x\})\ vs \Longrightarrow t \in vs\ x \Longrightarrow v(x := t) \in compatible\_vals\ A\ vs$
unfolding *compatible_vals_def*
by *auto*

lemma *fun_upd_in_compatible_vals_notin*: $x \notin A \Longrightarrow v \in compatible\_vals\ A\ vs \Longrightarrow v(x := t) \in compatible\_vals\ A\ vs$
unfolding *compatible_vals_def*
by *auto*

lemma *check_exec_check*:
assumes $\forall x.\ vs\ x \neq \{\}$
shows *s_check_exec vs* φ *sp* $\longleftrightarrow$ $(\forall v \in compatible\_vals\ (fv\ \varphi)\ vs.\ s\_check\ v\ \varphi\ sp)$
and *v_check_exec vs* φ *vp* $\longleftrightarrow$ $(\forall v \in compatible\_vals\ (fv\ \varphi)\ vs.\ v\_check\ v\ \varphi\ vp)$
using *assms*
proof (*induct* φ *arbitrary*: *vs sp vp*)
case *TT*
{
case *1*
then show *?case* **using** *compatible_vals_nonemptyI*
by (*cases sp*)
auto
next
case *2*
then show *?case* **using** *compatible_vals_nonemptyI*
by *auto*
}
next
case *FF*
{
case *1*
then show *?case* **using** *compatible_vals_nonemptyI*
by (*cases sp*)
auto
next
case *2*
then show *?case* **using** *compatible_vals_nonemptyI*
by (*cases vp*)
auto
}
next

```
  case (Pred p ts)
  {
    case 1
    have obs: ∀ tX∈set (vs⦇ts⦈). snd tX ≠ {}
      using ‹∀ x. vs x ≠ {}›
    proof (induct ts)
      case (Cons a ts)
      then show ?case
        by (cases a) (auto simp: eval_trms_set_def)
    qed (auto simp: eval_trms_set_def)
    show ?case
      using 1 compatible_vals_nonemptyI[OF 1]
        mk_values_complete[OF refl, of _ p ts vs] mk_values_sound[of _ vs ts p]
      by (cases sp)
        (auto 6 0 simp: mk_values_subset_iff[OF obs] simp del: fv.simps)
  next
    case 2
    then show ?case using compatible_vals_nonemptyI[OF 2]
        mk_values_complete[OF refl, of _ p ts vs] mk_values_sound[of _ vs ts p]
      by (cases vp)
        (auto 6 0 simp: mk_values_subset_Compl_def eval_trms_set_def simp del: fv.simps)
  }
next
  case (Eq_Const x c)
  {
    case 1
    then show ?case
      by (cases sp) (auto simp: compatible_vals_def)
  next
    case 2
    then show ?case
      by (cases vp) (auto simp: compatible_vals_def)
  }
next
  case (Neg φ)
  {
    case 1
    then show ?case
      using Neg.hyps(2) compatible_vals_nonemptyI[OF 1]
      by (cases sp) auto
  next
    case 2
    then show ?case
      using Neg.hyps(1) compatible_vals_nonemptyI[OF 2]
      by (cases vp) auto
  }
next
  case (Or φ1 φ2)
  {
    case 1
    with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
    proof (cases sp)
      case (SOrL sp')
      from check_fv_cong(1)[of φ1 _ _ sp'] show ?thesis
        unfolding SOrL s_check_exec_simps s_check_simps fv.simps Or(1)[OF 1, of sp']
      by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    next
      case (SOrR sp')
```

```
    from check_fv_cong(1)[of φ2 _ _ sp'] show ?thesis
      unfolding SOrR s_check_exec_simps s_check_simps fv.simps Or(3)[OF 1, of sp']
    by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
   qed (auto simp: compatible_vals_union_eq)
  next
   case 2
   with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
   proof (cases vp)
    case (VOr vp1 vp2)
    from check_fv_cong(2)[of φ1 _ _ vp1] check_fv_cong(2)[of φ2 _ _ vp2] show ?thesis
     unfolding VOr v_check_exec_simps v_check_simps fv.simps ball_conj_distrib
       Or(2)[OF 2, of vp1]  Or(4)[OF 2, of vp2]
       ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
    proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
     case φ1
     then show ?case
    by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    next
     case φ2
     then show ?case
    by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    qed
   qed (auto simp: compatible_vals_union_eq)
  }
 next
  case (And φ1 φ2)
  {
   case 1
   with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
   proof (cases sp)
    case (SAnd sp1 sp2)
    from check_fv_cong(1)[of φ1 _ _ sp1] check_fv_cong(1)[of φ2 _ _ sp2] show ?thesis
     unfolding SAnd s_check_exec_simps s_check_simps fv.simps ball_conj_distrib
       And(1)[OF 1, of sp1] And(3)[OF 1, of sp2]
       ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2]]
    proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
     case φ1
     then show ?case
    by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    next
     case φ2
     then show ?case
    by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    qed
   qed (auto simp: compatible_vals_union_eq)
  next
   case 2
   with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
   proof (cases vp)
    case (VAndL vp')
    from check_fv_cong(2)[of φ1 _ _ vp'] show ?thesis
     unfolding VAndL v_check_exec_simps v_check_simps fv.simps And(2)[OF 2, of vp']
    by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
   next
    case (VAndR vp')
    from check_fv_cong(2)[of φ2 _ _ vp'] show ?thesis
     unfolding VAndR v_check_exec_simps v_check_simps fv.simps And(4)[OF 2, of vp']
    by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
```

```
    qed (auto simp: compatible_vals_union_eq)
  }
next
  case (Imp φ1 φ2)
  {
    case 1
    with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
    proof (cases sp)
      case (SImpL vp')
      from check_fv_cong(2)[of φ1 _ _ vp'] show ?thesis
        unfolding SImpL s_check_exec_simps s_check_simps fv.simps Imp(2)[OF 1, of vp']
      by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    next
      case (SImpR sp')
      from check_fv_cong(1)[of φ2 _ _ sp'] show ?thesis
        unfolding SImpR s_check_exec_simps s_check_simps fv.simps Imp(3)[OF 1, of sp']
      by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    qed (auto simp: compatible_vals_union_eq)
  next
    case 2
    with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
    proof (cases vp)
      case (VImp sp1 vp2)
      from check_fv_cong(1)[of φ1 _ _ sp1] check_fv_cong(2)[of φ2 _ _ vp2] show ?thesis
        unfolding VImp v_check_exec_simps v_check_simps fv.simps ball_conj_distrib
          Imp(1)[OF 2, of sp1] Imp(4)[OF 2, of vp2]
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
        case φ1
        then show ?case
      by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      next
        case φ2
        then show ?case
      by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      qed
    qed (auto simp: compatible_vals_union_eq)
  }
next
  case (Iff φ1 φ2)
  {
    case 1
    with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
    proof (cases sp)
      case (SIffSS sp1 sp2)
      from check_fv_cong(1)[of φ1 _ _ sp1] check_fv_cong(1)[of φ2 _ _ sp2] show ?thesis
        unfolding SIffSS s_check_exec_simps s_check_simps fv.simps ball_conj_distrib
          Iff(1)[OF 1, of sp1] Iff(3)[OF 1, of sp2]
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
        case φ1
        then show ?case
      by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      next
        case φ2
        then show ?case
      by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      qed
```

```
    next
      case (SIffVV vp1 vp2)
      from check_fv_cong(2)[of φ1 _ _ vp1] check_fv_cong(2)[of φ2 _ _ vp2] show ?thesis
        unfolding SIffVV s_check_exec_simps s_check_simps fv.simps ball_conj_distrib
          Iff(2)[OF 1, of vp1] Iff(4)[OF 1, of vp2]
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
        case φ1
        then show ?case
       by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      next
        case φ2
        then show ?case
       by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      qed
    qed (auto simp: compatible_vals_union_eq)
  next
    case 2
    with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
    proof (cases vp)
      case (VIffSV sp1 vp2)
      from check_fv_cong(1)[of φ1 _ _ sp1] check_fv_cong(2)[of φ2 _ _ vp2] show ?thesis
        unfolding VIffSV v_check_exec_simps v_check_simps fv.simps ball_conj_distrib
          Iff(1)[OF 2, of sp1] Iff(4)[OF 2, of vp2]
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
        case φ1
        then show ?case
       by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      next
        case φ2
        then show ?case
       by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      qed
    next
      case (VIffVS vp1 sp2)
      from check_fv_cong(2)[of φ1 _ _ vp1] check_fv_cong(1)[of φ2 _ _ sp2] show ?thesis
        unfolding VIffVS v_check_exec_simps v_check_simps fv.simps ball_conj_distrib
          Iff(2)[OF 2, of vp1] Iff(3)[OF 2, of sp2]
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] refl, goal_cases φ1 φ2)
        case φ1
        then show ?case
       by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      next
        case φ2
        then show ?case
       by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      qed
    qed (auto simp: compatible_vals_union_eq)
  }
next
  case (Exists x φ)
  {
    case 1
    then have (vs(x := Z)) y ≠ {} if Z ≠ {} for Z y
      using that by auto
    with 1 have IH:
```

```
    s_check_exec (vs(x := {z})) φ sp = (∀ v∈compatible_vals (fv φ) (vs(x := {z})). s_check v φ sp)
    for z sp
    by (intro Exists;
        auto simp: compatible_vals_fun_upd fun_upd_same
        simp del: fun_upd_apply intro: fun_upd_in_compatible_vals)
  from 1 show ?case
    using compatible_vals_nonemptyI[OF 1, of fv φ − {x}]
     by (cases sp) (auto simp: SubsVals_nonempty IH fun_upd_in_compatible_vals_notin compatible_vals_fun_upd)
 next
  case 2
  then have (vs(x := Z)) y ≠ {} if Z ≠ {} for Z y
    using that by auto
  with 2 have IH:
    Z ≠ {} ⟹ v_check_exec (vs(x := Z)) φ vp = (∀ v∈compatible_vals (fv φ) (vs(x := Z)). v_check v φ vp)
    for Z vp
    by (intro Exists;
        auto simp: compatible_vals_fun_upd fun_upd_same
        simp del: fun_upd_apply intro: fun_upd_in_compatible_vals)
  show ?case
    using compatible_vals_nonemptyI[OF 2, of fv φ − {x}]
    by (cases vp)
      (auto simp: SubsVals_nonempty IH[OF SubsVals_nonempty]
        fun_upd_in_compatible_vals fun_upd_in_compatible_vals_notin compatible_vals_fun_upd
        ball_conj_distrib 2[simplified] split: prod.splits if_splits |
        drule bspec, assumption)+
 }
next
 case (Forall x φ)
 {
  case 1
  then have (vs(x := Z)) y ≠ {} if Z ≠ {} for Z y
    using that by auto
  with 1 have IH:
    Z ≠ {} ⟹ s_check_exec (vs(x := Z)) φ sp = (∀ v∈compatible_vals (fv φ) (vs(x := Z)). s_check v φ sp)
    for Z sp
    by (intro Forall;
        auto simp: compatible_vals_fun_upd fun_upd_same
        simp del: fun_upd_apply intro: fun_upd_in_compatible_vals)
  show ?case
    using compatible_vals_nonemptyI[OF 1, of fv φ − {x}]
    by (cases sp)
      (auto simp: SubsVals_nonempty IH[OF SubsVals_nonempty]
        fun_upd_in_compatible_vals fun_upd_in_compatible_vals_notin compatible_vals_fun_upd
        ball_conj_distrib 1[simplified] split: prod.splits if_splits |
        drule bspec, assumption)+
 next
  case 2
  then have (vs(x := Z)) y ≠ {} if Z ≠ {} for Z y
    using that by auto
  with 2 have IH:
    v_check_exec (vs(x := {z})) φ vp = (∀ v∈compatible_vals (fv φ) (vs(x := {z})). v_check v φ vp)
    for z vp
    by (intro Forall;
        auto simp: compatible_vals_fun_upd fun_upd_same
        simp del: fun_upd_apply intro: fun_upd_in_compatible_vals)
```

```
    from 2 show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ − {x}]
       by (cases vp) (auto simp: SubsVals_nonempty IH fun_upd_in_compatible_vals_notin compatible_vals_fun_upd)
  }
next
  case (Prev I φ)
  {
    case 1
    with Prev[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) auto
  next
    case 2
    with Prev[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) auto
  }
next
  case (Next I φ)
  {
    case 1
    with Next[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) (auto simp: Let_def)
  next
    case 2
    with Next[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) auto
  }
next
  case (Once I φ)
  {
    case 1
    with Once[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) (auto simp: Let_def)
  next
    case 2
    with Once[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) auto
  }
next
  case (Historically I φ)
  {
    case 1
    with Historically[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) auto
  next
    case 2
    with Historically[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) (auto simp: Let_def)
  }
next
```

```
  case (Eventually I φ)
  {
    case 1
    with Eventually[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) (auto simp: Let_def)
  next
    case 2
    with Eventually[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) auto
  }
next
  case (Always I φ)
  {
    case 1
    with Always[of vs] show ?case
      using compatible_vals_nonemptyI[OF 1, of fv φ]
      by (cases sp) auto
  next
    case 2
    with Always[of vs] show ?case
      using compatible_vals_nonemptyI[OF 2, of fv φ]
      by (cases vp) (auto simp: Let_def)
  }
next
  case (Since φ1 I φ2)
  {
    case 1
    with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
    proof (cases sp)
      case (SSince sp' sps)
      from check_fv_cong(1)[of φ2 _ _ sp'] show ?thesis
        unfolding SSince s_check_exec_simps s_check_simps fv.simps ball_conj_distrib ball_swap[of _
set sps]
          Since(1)[OF 1] Since(3)[OF 1, of sp'] Let_def
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set sps, OF refl] refl, goal_cases φ2 φ1)
        case φ2
        then show ?case
       by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
      next
        case (φ1 sp)
        then show ?case using check_fv_cong(1)[of φ1 _ _ sp]
       by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      qed
    qed (auto simp: compatible_vals_union_eq)
  next
    case 2
    with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
    proof (cases vp)
      case (VSince i vp' vps)
      from check_fv_cong(2)[of φ1 _ _ vp'] show ?thesis
       unfolding VSince v_check_exec_simps v_check_simps fv.simps ball_conj_distrib ball_swap[of _
set vps]
          Since(2)[OF 2, of vp'] Since(4)[OF 2] Let_def
          ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set vps, OF refl] refl, goal_cases φ1 φ2)
```

```
      case φ1
      then show ?case
     by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    next
      case (φ2 vp)
      then show ?case using check_fv_cong(2)[of φ2 _ _ vp]
     by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    qed
  next
    case (VSinceInf i j vps)
    show ?thesis
     unfolding VSinceInf v_check_exec_simps v_check_simps fv.simps ball_conj_distrib ball_swap[of
_ set vps]
       Since(4)[OF 2] Let_def
       ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
    proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set vps, OF refl] refl, goal_cases φ2)
      case (φ2 vp)
      then show ?case using check_fv_cong(2)[of φ2 _ _ vp]
     by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    qed
  qed (auto simp: compatible_vals_union_eq)
 }
next
 case (Until φ1 I φ2)
 {
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2] show ?case
  proof (cases sp)
    case (SUntil sps sp')
    from check_fv_cong(1)[of φ2 _ _ sp'] show ?thesis
      unfolding SUntil s_check_exec_simps s_check_simps fv.simps ball_conj_distrib ball_swap[of _
set sps]
       Until(1)[OF 1] Until(3)[OF 1, of sp'] Let_def
       ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of fv φ1 ∪ fv φ2]]
    proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set sps, OF refl] refl, goal_cases φ2 φ1)
      case φ2
      then show ?case
     by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    next
      case (φ1 sp)
      then show ?case using check_fv_cong(1)[of φ1 _ _ sp]
     by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    qed
  qed (auto simp: compatible_vals_union_eq)
 next
  case 2
  with compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2] show ?case
  proof (cases vp)
    case (VUntil i vps vp')
    from check_fv_cong(2)[of φ1 _ _ vp'] show ?thesis
      unfolding VUntil v_check_exec_simps v_check_simps fv.simps ball_conj_distrib ball_swap[of _
set vps]
       Until(2)[OF 2, of vp'] Until(4)[OF 2] Let_def
       ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
    proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set vps, OF refl] refl, goal_cases φ1 φ2)
      case φ1
      then show ?case
     by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
```

```
      next
        case (φ2 vp)
        then show ?case using check_fv_cong(2)[of φ2 _ _ vp]
       by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      qed
    next
      case (VUntilInf i j vps)
      show ?thesis
       unfolding VUntilInf v_check_exec_simps v_check_simps fv.simps ball_conj_distrib ball_swap[of
_ set vps]
         Until(4)[OF 2] Let_def
         ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of fv φ1 ∪ fv φ2]]
      proof (intro arg_cong2[of _ _ _ _ (∧)] ball_cong[of set vps, OF refl] refl, goal_cases φ2)
        case (φ2 vp)
        then show ?case using check_fv_cong(2)[of φ2 _ _ vp]
       by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
      qed
    qed (auto simp: compatible_vals_union_eq)
  }
qed

lemma s_check_code[code]: s_check v φ sp = s_check_exec (λx. {v x}) φ sp
  by (subst check_exec_check)
    (auto simp: compatible_vals_def elim: check_fv_cong[THEN iffD2, rotated])

lemma v_check_code[code]: v_check v φ vp = v_check_exec (λx. {v x}) φ vp
  by (subst check_exec_check)
    (auto simp: compatible_vals_def elim: check_fv_cong[THEN iffD2, rotated])
```

8.3 Latest Relevant Time-Point

```
fun LRTP :: ('n, 'd) formula ⇒ nat ⇒ nat option where
  LRTP ⊤ i = Some i
| LRTP ⊥ i = Some i
| LRTP (_ † _) i = Some i
| LRTP (_ ≈ _) i = Some i
| LRTP (¬F φ) i = LRTP φ i
| LRTP (φ ∨F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (φ ∧F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (φ ⟶F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (φ ⟷F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (∃F_. φ) i = LRTP φ i
| LRTP (∀F_. φ) i = LRTP φ i
| LRTP (Y I φ) i = LRTP φ (i−1)
| LRTP (X I φ) i = LRTP φ (i+1)
| LRTP (P I φ) i = LRTP φ (LTP_p_safe σ i I)
| LRTP (H I φ) i = LRTP φ (LTP_p_safe σ i I)
| LRTP (F I φ) i = (case right I of ∞ ⇒ None | enat b ⇒ LRTP φ (LTP_f σ i b))
| LRTP (G I φ) i = (case right I of ∞ ⇒ None | enat b ⇒ LRTP φ (LTP_f σ i b))
| LRTP (φ S I ψ) i = max_opt (LRTP φ i) (LRTP ψ (LTP_p_safe σ i I))
| LRTP (φ U I ψ) i = (case right I of ∞ ⇒ None | enat b ⇒ max_opt (LRTP φ ((LTP_f σ i b)−1))
(LRTP ψ (LTP_f σ i b)))

lemma fb_LRTP:
  assumes future_bounded φ
  shows ¬ Option.is_none (LRTP φ i)
  using assms
  by (induction φ i rule: LRTP.induct)
```

(*auto simp add*: *max_opt_def Option.is_none_def*)

lemma *not_none_fb_LRTP*:
assumes *future_bounded* φ
shows *LRTP* φ *i* $\neq$ *None*
using *assms fb_LRTP* **by** (*auto simp add*: *Option.is_none_def*)

lemma *is_some_fb_LRTP*:
assumes *future_bounded* φ
shows $\exists$ *j*. *LRTP* φ *i* = *Some j*
using *assms fb_LRTP* **by** (*auto simp add*: *Option.is_none_def*)

lemma *enat_trans*[*simp*]: *enat i* $\leq$ *enat j* $\wedge$ *enat j* $\leq$ *enat k* $\Longrightarrow$ *enat i* $\leq$ *enat k*
by *auto*

8.4 Active Domain

definition *AD* :: ('*n*, '*d*) *formula* $\Rightarrow$ *nat* $\Rightarrow$ '*d set*
where *AD* φ *i* = *consts* φ $\cup$ ($\bigcup$ *k* $\leq$ *the* (*LRTP* φ *i*). $\bigcup$ (*set* ‘ *snd* ‘ Γ σ *k*))

lemma *val_in_AD_iff*:
x $\in$ *fv* φ $\Longrightarrow$ *v x* $\in$ *AD* φ *i* $\longleftrightarrow$ *v x* $\in$ *consts* φ $\vee$
($\exists$ *r ts k*. *k* $\leq$ *the* (*LRTP* φ *i*) $\wedge$ (*r*, *v*⟦*ts*⟧) $\in$ Γ σ *k* $\wedge$ *x* $\in$ $\bigcup$ (*set* (*map fv_trm ts*)))
unfolding *AD_def Un_iff UN_iff Bex_def atMost_iff set_map*
ex_comm[*of P* :: _ $\Rightarrow$ *nat* $\Rightarrow$ _ **for** *P*] *ex_simps image_iff*
proof (*safe intro*!: *arg_cong*[*of* _ _ λ*x*. _ $\vee$ *x*] *ex_cong*, *unfold snd_conv*, *goal_cases LR RL*)
case (*LR i* _ *r ds*)
then show *?case*
by (*intro exI*[*of* _ *r*] *conjI*
exI[*of* _ *map* (λ*d*. *if v x* = *d then* (**v** *x*) *else* **c** *d*) *ds*])
(*auto simp*: *eval_trms_def o_def map_idI*)
next
case (*RL i r ts t*)
then show *?case*
by (*intro exI*[*of* _ *v*⟦*ts*⟧] *conjI*)
(*auto simp*: *eval_trms_def image_iff in_fv_trm_conv intro*!: *bexI*[*of* _ *t*])
qed

lemma *val_notin_AD_iff*:
x $\in$ *fv* φ $\Longrightarrow$ *v x* $\notin$ *AD* φ *i* $\longleftrightarrow$ *v x* $\notin$ *consts* φ $\wedge$
($\forall$ *r ts k*. *k* $\leq$ *the* (*LRTP* φ *i*) $\wedge$ *x* $\in$ $\bigcup$ (*set* (*map fv_trm ts*)) $\longrightarrow$ (*r*, *v*⟦*ts*⟧) $\notin$ Γ σ *k*)
using *val_in_AD_iff* **by** *blast*

lemma *finite_values*: *finite* ($\bigcup$ (*set* ‘ *snd* ‘ Γ σ *k*))
by (*transfer*, *auto simp add*: *sfinite_def*)

lemma *finite_tps*: *future_bounded* φ $\Longrightarrow$ *finite* ($\bigcup$ *k* < *the* (*LRTP* φ *i*). {*k*})
using *fb_LRTP*[*of* φ] *finite_enat_bounded*
by *simp*

lemma *finite_AD* [*simp*]: *future_bounded* φ $\Longrightarrow$ *finite* (*AD* φ *i*)
using *finite_tps finite_values*
by (*simp add*: *AD_def enat_def*)

lemma *finite_AD_UNIV*:
assumes *future_bounded* φ **and** *AD* φ *i* = (*UNIV*:: '*d set*)
shows *finite* (*UNIV*::'*d set*)
proof −

have *finite* (*AD* φ *i*)
using *finite_AD*[*of* φ *i*, *OF assms*(*1*)] **by** *simp*
then show *?thesis*
using *assms*(*2*) **by** *simp*
qed

8.5 Congruence Modulo Active Domain

lemma *AD_simps*[*simp*]:
AD ($\neg_F$ φ) *i* = *AD* φ *i*
future_bounded (φ $\vee_F$ ψ) $\Longrightarrow$ *AD* (φ $\vee_F$ ψ) *i* = *AD* φ *i* $\cup$ *AD* ψ *i*
future_bounded (φ $\wedge_F$ ψ) $\Longrightarrow$ *AD* (φ $\wedge_F$ ψ) *i* = *AD* φ *i* $\cup$ *AD* ψ *i*
future_bounded (φ $\longrightarrow_F$ ψ) $\Longrightarrow$ *AD* (φ $\longrightarrow_F$ ψ) *i* = *AD* φ *i* $\cup$ *AD* ψ *i*
future_bounded (φ $\longleftrightarrow_F$ ψ) $\Longrightarrow$ *AD* (φ $\longleftrightarrow_F$ ψ) *i* = *AD* φ *i* $\cup$ *AD* ψ *i*
AD ($\exists_F x.$ φ) *i* = *AD* φ *i*
AD ($\forall_F x.$ φ) *i* = *AD* φ *i*
AD (**Y** *I* φ) *i* = *AD* φ (*i* − *1*)
AD (**X** *I* φ) *i* = *AD* φ (*i* + *1*)
future_bounded (**F** *I* φ) $\Longrightarrow$ *AD* (**F** *I* φ) *i* = *AD* φ (*LTP_f* σ *i* (*the_enat* (*right I*)))
future_bounded (**G** *I* φ) $\Longrightarrow$ *AD* (**G** *I* φ) *i* = *AD* φ (*LTP_f* σ *i* (*the_enat* (*right I*)))
AD (**P** *I* φ) *i* = *AD* φ (*LTP_p_safe* σ *i I*)
AD (**H** *I* φ) *i* = *AD* φ (*LTP_p_safe* σ *i I*)
future_bounded (φ **S** *I* ψ) $\Longrightarrow$ *AD* (φ **S** *I* ψ) *i* = *AD* φ *i* $\cup$ *AD* ψ (*LTP_p_safe* σ *i I*)
future_bounded (φ **U** *I* ψ) $\Longrightarrow$ *AD* (φ **U** *I* ψ) *i* = *AD* φ (*LTP_f* σ *i* (*the_enat* (*right I*)) − *1*) $\cup$ *AD* ψ (*LTP_f* σ *i* (*the_enat* (*right I*)))
by (*auto 0 3 simp*: *AD_def max_opt_def not_none_fb_LRTP le_max_iff_disj Bex_def split*: *option.splits*)

lemma *LTP_p_mono*: *i* ≤ *j* $\Longrightarrow$ *LTP_p_safe* σ *i I* ≤ *LTP_p_safe* σ *j I*
unfolding *LTP_p_safe_def*
by (*smt* (*verit*, *ccfv_threshold*) τ*_mono bot_nat_0.extremum diff_le_mono order.trans i_LTP_tau le_cases3 min.bounded_iff*)

lemma *LTP_f_mono*:
assumes *i* ≤ *j*
shows*LTP_f* σ *i b* ≤ *LTP_f* σ *j b*
unfolding *LTP_def*
proof (*rule Max_mono*)
show *finite* {*i*. τ σ *i* ≤ τ σ *j* + *b*}
unfolding *finite_nat_set_iff_bounded_le*
by (*metis i_le_LTPi_add le_Suc_ex mem_Collect_eq*)
qed (*auto simp*: *assms intro*!: *exI*[*of* _ *i*] *elim*: *order_trans*)

lemma *LRTP_mono*: *future_bounded* φ $\Longrightarrow$ *i* ≤ *j* $\Longrightarrow$ *the* (*LRTP* φ *i*) ≤ *the* (*LRTP* φ *j*)
proof (*induct* φ *arbitrary*: *i j*)
case (*Or* φ*1* φ*2*)
from *Or*(*1*,*2*)[*of i j*] *Or*(*3*−) **show** *?case*
by (*auto simp*: *max_opt_def not_none_fb_LRTP split*: *option.splits*)
next
case (*And* φ*1* φ*2*)
from *And*(*1*,*2*)[*of i j*] *And*(*3*−) **show** *?case*
by (*auto simp*: *max_opt_def not_none_fb_LRTP split*: *option.splits*)
next
case (*Imp* φ*1* φ*2*)
from *Imp*(*1*,*2*)[*of i j*] *Imp*(*3*−) **show** *?case*
by (*auto simp*: *max_opt_def not_none_fb_LRTP split*: *option.splits*)
next

```
  case (Iff φ1 φ2)
  from Iff(1,2)[of i j] Iff(3−) show ?case
    by (auto simp: max_opt_def not_none_fb_LRTP split: option.splits)
next
  case (Since φ1 I φ2)
  from Since(1)[OF _ Since(4)] Since(2)[of LTP_p_safe σ i I LTP_p_safe σ j I] Since(3−)
  show ?case
    by (auto simp: max_opt_def not_none_fb_LRTP LTP_p_mono split: option.splits)
next
  case (Until φ1 I φ2)
  from Until(1)[of LTP_f σ i (the_enat (right I)) − 1 LTP_f σ j (the_enat (right I)) − 1]
    Until(2)[of LTP_f σ i (the_enat (right I)) LTP_f σ j (the_enat (right I))] Until(3−)
  show ?case
    by (auto simp: max_opt_def not_none_fb_LRTP LTP_f_mono diff_le_mono split: option.splits)
qed (auto simp: LTP_p_mono LTP_f_mono)

lemma AD_mono: future_bounded φ ⟹ i ≤ j ⟹ AD φ i ⊆ AD φ j
  by (auto 0 3 simp: AD_def Bex_def intro: LRTP_mono elim!: order_trans)

lemma LTP_p_safe_le[simp]: LTP_p_safe σ i I ≤ i
  by (auto simp: LTP_p_safe_def)

lemma check_AD_cong:
  assumes future_bounded φ
    and (∀ x ∈ fv φ. v x = v' x ∨ (v x ∉ AD φ i ∧ v' x ∉ AD φ i))
  shows (s_at sp = i ⟹ s_check v φ sp ⟷ s_check v' φ sp)
    (v_at vp = i ⟹ v_check v φ vp ⟷ v_check v' φ vp)
  using assms
proof (induction v φ sp and v φ vp arbitrary: i v' and i v' rule: s_check_v_check.induct)
  case (1 v f sp)
  note IH = 1(1−23)[OF refl] and hyps = 1(24−26)
  show ?case
  proof (cases sp)
    case (SPred j r ts)
    then show ?thesis
    proof (cases f)
      case (Pred q us)
      with SPred hyps show ?thesis
        using eval_trms_fv_cong[of ts v v']
        by (force simp: val_notin_AD_iff dest!: spec[of _ i] spec[of _ r] spec[of _ ts])
    qed auto
  next
    case (SEq_Const j r ts)
    with hyps show ?thesis
      by (cases f) (auto simp: val_notin_AD_iff)
  next
    case (SNeg vp')
    then show ?thesis
      using IH(1)[of _ _ _ v'] hyps
      by (cases f) auto
  next
    case (SOrL sp')
    then show ?thesis
      using IH(2)[of _ _ _ _ v'] hyps
      by (cases f) auto
  next
    case (SOrR sp')
    then show ?thesis
```

```
    using IH(3)[of _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (SAnd sp1 sp2)
  then show ?thesis
    using IH(4,5)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (SImpL vp')
  then show ?thesis
    using IH(6)[of _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (SImpR sp')
  then show ?thesis
    using IH(7)[of _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (SIffSS sp1 sp2)
  then show ?thesis
    using IH(8,9)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (SIffVV vp1 vp2)
  then show ?thesis
    using IH(10,11)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (SExists x z sp')
  then show ?thesis
    using IH(12)[of x _ x z sp' i v'(x := z)] hyps
    by (cases f) (auto simp add: fun_upd_def)
next
  case (SForall x part)
  then show ?thesis
    using IH(13)[of x _ x part _ _ D _ z _ v'(x := z) for D z, OF _ _ _ _ refl _ refl] hyps
    by (cases f) (auto simp add: fun_upd_def)
next
  case (SPrev sp')
  then show ?thesis
    using IH(14)[of _ _ _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (SNext sp')
  then show ?thesis
    using IH(15)[of _ _ _ _ _ _ _ v'] hyps
    by (cases f) (auto simp add: Let_def)
next
  case (SOnce j sp')
  then show ?thesis
  proof (cases f)
    case (Once I φ)
    { fix k
      assume k: k ≤ i τ σ i − left I ≥ τ σ k
      then have τ σ i − left I ≥ τ σ 0
        by (meson τ_mono le0 order_trans)
      with k have k ≤ LTP_p_safe σ i I
        unfolding LTP_p_safe_def by (auto simp: i_LTP_tau)
```

```
    with Once hyps(2,3) have ∀x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
      by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
  }
  with Once SOnce show ?thesis
    using IH(16)[OF Once SOnce refl refl, of v'] hyps(1,2)
    by (auto simp: Let_def le_diff_conv2)
 qed auto
next
 case (SHistorically j k sps)
 then show ?thesis
 proof (cases f)
  case (Historically I φ)
  { fix sp :: ('n, 'd) sproof
    define l and u where l = s_at sp and u = LTP_p σ i I
    assume ∗: sp ∈ set sps τ σ 0 + left I ≤ τ σ i
    then have u_def: u = LTP_p_safe σ i I
      by (auto simp: LTP_p_safe_def u_def)
    from ∗(1) obtain j where j: sp = sps ! j j < length sps
      unfolding in_set_conv_nth by auto
    moreover
    assume eq: map s_at sps = [k ..< Suc u]
    then have len: length sps = Suc u − k
      by (auto dest!: arg_cong[where f=length])
    moreover
    have s_at (sps ! j) = k + j
      using arg_cong[where f=λxs. nth xs j, OF eq] j len ∗(2)
      by (auto simp: nth_append)
    ultimately have l ≤ u
      unfolding l_def by auto
    with Historically hyps(2,3) have ∀x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
      by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
  }
  with Historically SHistorically show ?thesis
    using IH(17)[OF Historically SHistorically _ refl, of _ v'] hyps(1,2)
    by auto
 qed auto
next
 case (SEventually j sp')
 then show ?thesis
 proof (cases f)
  case (Eventually I φ)
  { fix k
    assume τ σ k ≤ the_enat (right I) + τ σ i
    then have k ≤ LTP_f σ i (the_enat (right I))
      by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
    with Eventually hyps(2,3) have ∀x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
      by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
  }
  with Eventually SEventually show ?thesis
    using IH(18)[OF Eventually SEventually refl refl, of v'] hyps(1,2)
    by (auto simp: Let_def)
 qed auto
next
 case (SAlways j k sps)
 then show ?thesis
 proof (cases f)
  case (Always I φ)
  { fix sp :: ('n, 'd) sproof
```

```
      define l and u where l = s_at sp and u = LTP_f σ i (the_enat (right I))
      assume ∗: sp ∈ set sps
      then obtain j where j: sp = sps ! j j < length sps
        unfolding in_set_conv_nth by auto
      assume eq: map s_at sps = [ETP_f σ i I ..< Suc u]
      then have length sps = Suc u − ETP_f σ i I
        by (auto dest!: arg_cong[where f=length])
      with j eq have l ≤ LTP_f σ i (the_enat (right I))
        by (auto simp: l_def u_def dest!: arg_cong[where f=λxs. nth xs j]
            simp del: upt.simps split: if_splits)
      with Always hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Always SAlways show ?thesis
      using IH(19)[OF Always SAlways _ refl, of _ v'] hyps(1,2)
      by auto
  qed auto
next
  case (SSince sp' sps)
  then show ?thesis
  proof (cases f)
    case (Since φ I ψ)
    { fix sp :: ('n, 'd) sproof
      define l where l = s_at sp
      assume ∗: sp ∈ set sps
      from ∗(1) obtain j where j: sp = sps ! j j < length sps
        unfolding in_set_conv_nth by auto
      moreover
      assume eq: map s_at sps = [Suc (s_at sp')  ..< Suc i]
      then have len: length sps = i − s_at sp'
        by (auto dest!: arg_cong[where f=length])
      moreover
      have s_at (sps ! j) = Suc (s_at sp') + j
        using arg_cong[where f=λxs. nth xs j, OF eq] j len
        by (auto simp: nth_append)
      ultimately have l ≤ i
        unfolding l_def by auto
      with Since hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
        by (auto simp: dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    moreover
    { fix k
      assume k: k ≤ i τ σ i − left I ≥ τ σ k
      then have τ σ i − left I ≥ τ σ 0
        by (meson τ_mono le0 order_trans)
      with k have k ≤ LTP_p_safe σ i I
        unfolding LTP_p_safe_def by (auto simp: i_LTP_tau)
      with Since hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ k ∧ v' x ∉ AD ψ k
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    ultimately show ?thesis
      using Since SSince IH(20)[OF Since SSince refl refl refl, of v'] IH(21)[OF Since SSince refl refl
_ refl, of _ v'] hyps(1,2)
      by (auto simp: Let_def le_diff_conv2 simp del: upt.simps)
  qed auto
next
  case (SUntil sps sp')
  then show ?thesis
```

```
    proof (cases f)
      case (Until φ I ψ)
      { fix sp :: ('n, 'd) sproof
        define l where l = s_at sp
        assume ∗: sp ∈ set sps
        from ∗(1) obtain j where j: sp = sps ! j j < length sps
          unfolding in_set_conv_nth by auto
        moreover
        assume δ σ (s_at sp') i ≤ the_enat (right I)
        then have s_at sp' ≤ LTP_f σ i (the_enat (right I))
          by (metis add.commute i_le_LTPi_add le_add_diff_inverse le_diff_conv)
        moreover
        assume eq: map s_at sps = [i ..< s_at sp']
        then have len: length sps = s_at sp' − i
          by (auto dest!: arg_cong[where f=length])
        moreover
        have s_at (sps ! j) = i + j
          using arg_cong[where f=λxs. nth xs j, OF eq] j len
          by (auto simp: nth_append)
        ultimately have l ≤ LTP_f σ i (the_enat (right I)) − 1
          unfolding l_def by auto
        with Until hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
          by (auto simp: dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
      }
      moreover
      { fix k
        assume τ σ k ≤ the_enat (right I) + τ σ i
        then have k ≤ LTP_f σ i (the_enat (right I))
          by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
        with Until hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ k ∧ v' x ∉ AD ψ k
          by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
      }
      ultimately show ?thesis
        using Until SUntil IH(22)[OF Until SUntil refl refl refl, of v'] IH(23)[OF Until SUntil refl refl _
refl, of _ v'] hyps(1,2)
        by (auto simp: Let_def le_diff_conv2 simp del: upt.simps)
    qed auto
  qed (cases f; simp_all)+
next
  case (2 v f vp)
  note IH = 2(1−25)[OF refl] and hyps = 2(26−28)
  show ?case
  proof (cases vp)
    case (VPred j r ts)
    then show ?thesis
    proof (cases f)
      case (Pred q us)
      with VPred hyps show ?thesis
        using eval_trms_fv_cong[of ts v v']
        by (force simp: val_notin_AD_iff dest!: spec[of _ i] spec[of _ r] spec[of _ ts])
    qed auto
  next
    case (VEq_Const j r ts)
    with hyps show ?thesis
      by (cases f) (auto simp: val_notin_AD_iff)
  next
    case (VNeg sp')
    then show ?thesis
```

```
    using IH(1)[of _ _ _ v'] hyps
    by (cases f) auto
next
  case (VOr vp1 vp2)
  then show ?thesis
    using IH(2,3)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (VAndL vp')
  then show ?thesis
    using IH(4)[of _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (VAndR vp')
  then show ?thesis
    using IH(5)[of _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (VImp sp1 vp2)
  then show ?thesis
    using IH(6,7)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (VIffSV sp1 vp2)
  then show ?thesis
    using IH(8,9)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (VIffVS vp1 sp2)
  then show ?thesis
    using IH(10,11)[of _ _ _ _ _ v'] hyps
    by (cases f) (auto 7 0)+
next
  case (VExists x part)
  then show ?thesis
    using IH(12)[of x _ x part _ _ D _ z _ v'(x := z) for D z, OF _ _ _ _ refl _ refl] hyps
    by (cases f) (auto simp add: fun_upd_def)
next
  case (VForall x z vp')
  then show ?thesis
    using IH(13)[of x _ x z vp' i v'(x := z)] hyps
    by (cases f) (auto simp add: fun_upd_def)
next
  case (VPrev vp')
  then show ?thesis
    using IH(14)[of _ _ _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (VNext vp')
  then show ?thesis
    using IH(15)[of _ _ _ _ _ _ v'] hyps
    by (cases f) auto
next
  case (VOnce j k vps)
  then show ?thesis
  proof (cases f)
    case (Once I φ)
    { fix vp :: ('n, 'd) vproof
```

```
      define l and u where l = v_at vp and u = LTP_p σ i I
      assume ∗: vp ∈ set vps τ σ 0 + left I ≤ τ σ i
      then have u_def: u = LTP_p_safe σ i I
        by (auto simp: LTP_p_safe_def u_def)
      from ∗(1) obtain j where j: vp = vps ! j j < length vps
        unfolding in_set_conv_nth by auto
      moreover
      assume eq: map v_at vps = [k ..< Suc u]
      then have len: length vps = Suc u − k
        by (auto dest!: arg_cong[where f=length])
      moreover
      have v_at (vps ! j) = k + j
        using arg_cong[where f=λxs. nth xs j, OF eq] j len ∗(2)
        by (auto simp: nth_append)
      ultimately have l ≤ u
        unfolding l_def by auto
      with Once hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
        by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Once VOnce show ?thesis
      using IH(16)[OF Once VOnce _ refl, of _ v'] hyps(1,2)
      by auto
  qed auto
next
  case (VHistorically j vp')
  then show ?thesis
  proof (cases f)
    case (Historically I φ)
    { fix k
      assume k: k ≤ i τ σ i − left I ≥ τ σ k
      then have τ σ i − left I ≥ τ σ 0
        by (meson τ_mono le0 order_trans)
      with k have k ≤ LTP_p_safe σ i I
        unfolding LTP_p_safe_def by (auto simp: i_LTP_tau)
      with Historically hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Historically VHistorically show ?thesis
      using IH(17)[OF Historically VHistorically refl refl, of v'] hyps(1,2)
      by (auto simp: Let_def le_diff_conv2)
  qed auto
next
  case (VEventually j k vps)
  then show ?thesis
  proof (cases f)
    case (Eventually I φ)
    { fix vp :: ('n, 'd) vproof
      define l and u where l = v_at vp and u = LTP_f σ i (the_enat (right I))
      assume ∗: vp ∈ set vps
      then obtain j where j: vp = vps ! j j < length vps
        unfolding in_set_conv_nth by auto
      assume eq: map v_at vps = [ETP_f σ i I ..< Suc u]
      then have length vps = Suc u − ETP_f σ i I
        by (auto dest!: arg_cong[where f=length])
      with j eq have l ≤ LTP_f σ i (the_enat (right I))
        by (auto simp: l_def u_def dest!: arg_cong[where f=λxs. nth xs j]
            simp del: upt.simps split: if_splits)
      with Eventually hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ l ∧ v' x ∉ AD φ l
```

```
      by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Eventually VEventually show ?thesis
      using IH(18)[OF Eventually VEventually _ refl, of _ v'] hyps(1,2)
      by auto
  qed auto
next
  case (VAlways j vp')
  then show ?thesis
  proof (cases f)
    case (Always I φ)
    { fix k
      assume τ σ k ≤ the_enat (right I) + τ σ i
      then have k ≤ LTP_f σ i (the_enat (right I))
        by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
      with Always hyps(2,3) have ∀x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Always VAlways show ?thesis
      using IH(19)[OF Always VAlways refl refl, of v'] hyps(1,2)
      by (auto simp: Let_def)
  qed auto
next
  case (VSince j vp' vps)
  then show ?thesis
  proof (cases f)
    case (Since φ I ψ)
    { fix sp :: ('n, 'd) vproof
      define l and u where l = v_at sp and u = LTP_p σ i I
      assume ∗: sp ∈ set vps τ σ 0 + left I ≤ τ σ i
      then have u_def: u = LTP_p_safe σ i I
        by (auto simp: LTP_p_safe_def u_def)
      from ∗(1) obtain j where j: sp = vps ! j j < length vps
        unfolding in_set_conv_nth by auto
      moreover
      assume eq: map v_at vps = [v_at vp' ..< Suc u]
      then have len: length vps = Suc u − v_at vp'
        by (auto dest!: arg_cong[where f=length])
      moreover
      have v_at (vps ! j) = v_at vp' + j
        using arg_cong[where f=λxs. nth xs j, OF eq] j len
        by (auto simp: nth_append)
      ultimately have l ≤ u
        unfolding l_def by auto
      with Since hyps(2,3) have ∀x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ l ∧ v' x ∉ AD ψ l
        by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    moreover
    { fix k
      assume k: k ≤ i
      with Since hyps(2,3) have ∀x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    ultimately show ?thesis
      using Since VSince IH(20)[OF Since VSince refl refl, of v'] IH(21)[OF Since VSince refl _ refl,
of _ v'] hyps(1,2)
      by (auto simp: Let_def le_diff_conv2 simp del: upt.simps)
  qed auto
```

```
next
  case (VSinceInf j k vps)
  then show ?thesis
  proof (cases f)
    case (Since φ I ψ)
    { fix vp :: ('n, 'd) vproof
      define l and u where l = v_at vp and u = LTP_p σ i I
      assume ∗: vp ∈ set vps τ σ 0 + left I ≤ τ σ i
      then have u_def: u = LTP_p_safe σ i I
        by (auto simp: LTP_p_safe_def u_def)
      from ∗(1) obtain j where j: vp = vps ! j j < length vps
        unfolding in_set_conv_nth by auto
      moreover
      assume eq: map v_at vps = [k ..< Suc u]
      then have len: length vps = Suc u − k
        by (auto dest!: arg_cong[where f=length])
      moreover
      have v_at (vps ! j) = k + j
        using arg_cong[where f=λxs. nth xs j, OF eq] j len ∗(2)
        by (auto simp: nth_append)
      ultimately have l ≤ u
        unfolding l_def by auto
      with Since hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ l ∧ v' x ∉ AD ψ l
        by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    with Since VSinceInf show ?thesis
      using IH(22)[OF Since VSinceInf _ refl, of _ v'] hyps(1,2)
      by auto
  qed auto
next
  case (VUntil j vps vp')
  then show ?thesis
  proof (cases f)
    case (Until φ I ψ)
    { fix sp :: ('n, 'd) vproof
      define l and u where l = v_at sp and u = v_at vp'
      assume ∗: sp ∈ set vps v_at vp' ≤ LTP_f σ i (the_enat (right I))
      from ∗(1) obtain j where j: sp = vps ! j j < length vps
        unfolding in_set_conv_nth by auto
      moreover
      assume eq: map v_at vps = [ETP_f σ i I ..< Suc u]
      then have length vps = Suc u − ETP_f σ i I
        by (auto dest!: arg_cong[where f=length])
      with j eq ∗(2) have l ≤ LTP_f σ i (the_enat (right I))
        by (auto simp: l_def u_def dest!: arg_cong[where f=λxs. nth xs j]
            simp del: upt.simps split: if_splits)
      with Until hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ l ∧ v' x ∉ AD ψ l
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    moreover
    { fix k
      assume k < LTP_f σ i (the_enat (right I))
      then have k ≤ LTP_f σ i (the_enat (right I)) − 1
        by linarith
      with Until hyps(2,3) have ∀ x∈fv φ. v x = v' x ∨ v x ∉ AD φ k ∧ v' x ∉ AD φ k
        by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
    }
    ultimately show ?thesis
```

```
      using Until VUntil IH(23)[OF Until VUntil refl refl, of v'] IH(24)[OF Until VUntil refl _ refl, of
_ v'] hyps(1,2)
      by (auto simp: Let_def le_diff_conv2 simp del: upt.simps)
    qed auto
  next
    case (VUntilInf j k vps)
    then show ?thesis
    proof (cases f)
      case (Until φ I ψ)
      { fix vp :: ('n, 'd) vproof
        define l and u where l = v_at vp and u = LTP_f σ i (the_enat (right I))
        assume *: vp ∈ set vps
        then obtain j where j: vp = vps ! j j < length vps
          unfolding in_set_conv_nth by auto
        assume eq: map v_at vps = [ETP_f σ i I ..< Suc u]
        then have length vps = Suc u − ETP_f σ i I
          by (auto dest!: arg_cong[where f=length])
        with j eq have l ≤ LTP_f σ i (the_enat (right I))
          by (auto simp: l_def u_def dest!: arg_cong[where f=λxs. nth xs j]
              simp del: upt.simps split: if_splits)
        with Until hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ l ∧ v' x ∉ AD ψ l
          by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated −1])
      }
      with Until VUntilInf show ?thesis
        using IH(25)[OF Until VUntilInf _ refl, of _ v'] hyps(1,2)
        by auto
    qed auto
  qed (cases f; simp_all)+
qed
```

8.6 Checker Completeness

```
lemma part_hd_tabulate: distinct xs ⟹ part_hd (tabulate xs f z) = (case xs of [] ⇒ z | (x # _) ⇒ (if
set xs = UNIV then f x else z))
  by (transfer, auto split: list.splits)

lemma s_at_tabulate:
  assumes ∀ z. s_at (mypick z) = i
    and mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z ∉ AD φ i))
  shows ∀ (sub, vp) ∈ SubsVals mypart. s_at vp = i
  using assms by (transfer, auto)

lemma v_at_tabulate:
  assumes ∀ z. v_at (mypick z) = i
    and mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z ∉ AD φ i))
  shows ∀ (sub, vp) ∈ SubsVals mypart. v_at vp = i
  using assms by (transfer, auto)

lemma s_check_tabulate:
  assumes future_bounded φ
    and ∀ z. s_at (mypick z) = i
    and ∀ z. s_check (v(x:=z)) φ (mypick z)
    and mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z ∉ AD φ i))
  shows ∀ (sub, vp) ∈ SubsVals mypart. ∀ z ∈ sub. s_check (v(x := z)) φ vp
  using assms
proof (transfer fixing: σ φ mypick i v x, goal_cases 1)
  case (1 mypart)
  { fix z
```

```
    assume s_at_assm: ∀ z. s_at (mypick z) = i
      and s_check_assm: ∀ z. s_check (v(x := z)) φ (mypick z)
      and fb_assm: future_bounded φ
      and z_notin_AD: z ∉ (AD φ i)
    have s_at_mypick: s_at (mypick (SOME z. z ∉ local.AD φ i)) = i
      using s_at_assm by simp
    have s_check_mypick: Checker.s_check σ (v(x := SOME z. z ∉ AD φ i)) φ (mypick (SOME z. z ∉
AD φ i))
      using s_check_assm by simp
    have s_check (v(x := z)) φ (mypick (SOME z. z ∉ AD φ i))
      using z_notin_AD
      by (subst check_AD_cong(1)[of φ v(x := z) v(x := (SOME z. z ∉ Checker.AD σ φ i)) i mypick
(SOME z. z ∉ AD φ i), OF fb_assm _ s_at_mypick])
        (auto simp add: someI[of λz. z ∉ AD φ i z] s_check_mypick fb_assm split: if_splits)
  }
  with 1 show ?case
    by auto
qed

lemma v_check_tabulate:
  assumes future_bounded φ
    and ∀ z. v_at (mypick z) = i
    and ∀ z. v_check (v(x:=z)) φ (mypick z)
    and mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z ∉ AD φ i))
  shows ∀ (sub, vp) ∈ SubsVals mypart. ∀ z ∈ sub. v_check (v(x := z)) φ vp
  using assms
proof (transfer fixing: σ φ mypick i v x, goal_cases 1)
  case (1 mypart)
  { fix z
    assume v_at_assm: ∀ z. v_at (mypick z) = i
      and v_check_assm: ∀ z. v_check (v(x := z)) φ (mypick z)
      and fb_assm: future_bounded φ
      and z_notin_AD: z ∉ (AD φ i)
    have v_at_mypick: v_at (mypick (SOME z. z ∉ local.AD φ i)) = i
      using v_at_assm by simp
    have v_check_mypick: Checker.v_check σ (v(x := SOME z. z ∉ AD φ i)) φ (mypick (SOME z. z ∉
AD φ i))
      using v_check_assm by simp
    have v_check (v(x := z)) φ (mypick (SOME z. z ∉ AD φ i))
      using z_notin_AD
      by (subst check_AD_cong(2)[of φ v(x := z) v(x := (SOME z. z ∉ Checker.AD σ φ i)) i mypick
(SOME z. z ∉ AD φ i), OF fb_assm _ v_at_mypick])
        (auto simp add: someI[of λz. z ∉ AD φ i z] v_check_mypick fb_assm split: if_splits)
  }
  with 1 show ?case
    by auto
qed

lemma s_at_part_hd_tabulate:
  assumes future_bounded φ
    and ∀ z. s_at (f z) = i
    and mypart = tabulate (sorted_list_of_set (AD φ i)) f (f (SOME z. z ∉ AD φ i))
  shows s_at (part_hd mypart) = i
  using assms by (simp add: part_hd_tabulate split: list.splits)

lemma v_at_part_hd_tabulate:
  assumes future_bounded φ
    and ∀ z. v_at (f z) = i
```

```
    and mypart = tabulate (sorted_list_of_set (AD φ i)) f (f (SOME z. z ∉ AD φ i))
  shows v_at (part_hd mypart) = i
  using assms by (simp add: part_hd_tabulate split: list.splits)

lemma check_completeness_aux:
  (SAT σ v i φ ⟶ future_bounded φ ⟶ (∃ sp. s_at sp = i ∧ s_check v φ sp)) ∧
   (VIO σ v i φ ⟶ future_bounded φ ⟶ (∃ vp. v_at vp = i ∧ v_check v φ vp))
proof (induct v i φ rule: SAT_VIO.induct)
  case (STT v i)
  then show ?case
    by (auto intro!: exI[of _ STT i])
next
  case (VFF v i)
  then show ?case
    by (auto intro!: exI[of _ VFF i])
next
  case (SPred r v ts i)
  then show ?case
    by (auto intro!: exI[of _ SPred i r ts])
next
  case (VPred r v ts i)
  then show ?case
    by (auto intro!: exI[of _ VPred i r ts])
next
  case (SEq_Const v x c i)
  then show ?case
    by (auto intro!: exI[of _ SEq_Const i x c])
next
  case (VEq_Const v x c i)
  then show ?case
    by (auto intro!: exI[of _ VEq_Const i x c])
next
  case (SNeg v i φ)
  then show ?case
    by (auto intro: exI[of _ SNeg _])
next
  case (VNeg v i φ)
  then show ?case
    by (auto intro: exI[of _ VNeg _])
next
  case (SOrL v i φ ψ)
  then show ?case
    by (auto intro: exI[of _ SOrL _])
next
  case (SOrR v i ψ φ)
  then show ?case
    by (auto intro: exI[of _ SOrR _])
next
  case (VOr v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ VOr _ _])
next
  case (SAnd v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ SAnd _ _])
next
  case (VAndL v i φ ψ)
  then show ?case
```

```
    by (auto intro: exI[of _ VAndL _])
next
  case (VAndR v i ψ φ)
  then show ?case
    by (auto intro: exI[of _ VAndR _])
next
  case (SImpL v i φ ψ)
  then show ?case
    by (auto intro: exI[of _ SImpL _])
next
  case (SImpR v i ψ φ)
  then show ?case
    by (auto intro: exI[of _ SImpR _])
next
  case (VImp v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ VImp _ _])
next
  case (SIffSS v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ SIffSS _ _])
next
  case (SIffVV v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ SIffVV _ _])
next
  case (VIffSV v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ VIffSV _ _])
next
  case (VIffVS v i φ ψ)
  then show ?case
    by (auto 0 3 intro: exI[of _ VIffVS _ _])
next
  case (SExists v x i φ)
  then show ?case
    by (auto 0 3 simp: fun_upd_def intro: exI[of _ SExists x _ _])
next
  case (VExists v x i φ)
  show ?case
  proof
    assume future_bounded (∃F x. φ)
    then have fb: future_bounded φ
      by simp
    obtain mypick where mypick_def: v_at (mypick z) = i ∧ v_check (v(x:=z)) φ (mypick z) for z
      using VExists fb by metis
    define mypart where mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z
∉ (AD φ i)))
    have mypick_at: ∀ z. v_at (mypick z) = i
      by (simp add: mypick_def)
    have mypick_v_check: ∀ z. v_check (v(x:=z)) φ (mypick z)
      by (simp add: mypick_def)
    have mypick_v_check2: ∀ z. v_check (v(x := (SOME z. z ∉ AD φ i))) φ (mypick (SOME z. z ∉ AD
φ i))
      by (simp add: mypick_def)
    have v_at_myp: v_at (VExists x mypart) = i
      using v_at_part_hd_tabulate[OF fb, of mypick i]
      by (simp add: mypart_def mypick_def)
```

```
    have v_check_myp: v_check v (∃_F x.  φ) (VExists x mypart)
      using v_at_tabulate[of mypick i _ φ, OF mypick_at]
        v_check_tabulate[OF fb mypick_at mypick_v_check]
      by (auto simp add: mypart_def v_at_part_hd_tabulate[OF fb mypick_at])
    show ∃ vp. v_at vp = i ∧ v_check v (∃_F x.  φ) vp
      using v_at_myp v_check_myp by blast
  qed
next
  case (SForall v x i φ)
  show ?case
  proof
    assume future_bounded (∀_F x.  φ)
    then have fb: future_bounded φ
      by simp
    obtain mypick where mypick_def: s_at (mypick z) = i ∧ s_check (v(x:=z)) φ (mypick z) for z
      using SForall fb by metis
    define mypart where mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z
∉ (AD φ i)))
    have mypick_at: ∀ z. s_at (mypick z) = i
      by (simp add: mypick_def)
    have mypick_s_check: ∀ z. s_check (v(x:=z)) φ (mypick z)
      by (simp add: mypick_def)
    have mypick_s_check2: ∀ z. s_check (v(x := (SOME z. z ∉ AD φ i))) φ (mypick (SOME z. z ∉ AD
φ i))
      by (simp add: mypick_def)
    have s_at_myp: s_at (SForall x mypart) = i
      using s_at_part_hd_tabulate[OF fb, of mypick i]
      by (simp add: mypart_def mypick_def)
    have s_check_myp: s_check v (∀_F x.  φ) (SForall x mypart)
      using s_at_tabulate[of mypick i _ φ, OF mypick_at]
        s_check_tabulate[OF fb mypick_at mypick_s_check]
      by (auto simp add: mypart_def s_at_part_hd_tabulate[OF fb mypick_at])
    show ∃ sp. s_at sp = i ∧ s_check v (∀_F x.  φ) sp
      using s_at_myp s_check_myp by blast
  qed
next
  case (VForall v x i φ)
  then show ?case
    by (auto 0 3 simp: fun_upd_def intro: exI[of _ VForall x _ _])
next
  case (SPrev i I v φ)
  then show ?case
    by (force intro: exI[of _ SPrev _])
next
  case (VPrev i v φ I)
  then show ?case
    by (force intro: exI[of _ VPrev _])
next
  case (VPrevZ i v I φ)
  then show ?case
    by (auto intro!: exI[of _ VPrevZ])
next
  case (VPrevOutL i I v φ)
  then show ?case
    by (auto intro!: exI[of _ VPrevOutL i])
next
  case (VPrevOutR i I v φ)
  then show ?case
```

```
    by (auto intro!: exI[of _ VPrevOutR i])
next
  case (SNext i I v φ)
  then show ?case
    by (force simp: Let_def intro: exI[of _ SNext _])
next
  case (VNext v i φ I)
  then show ?case
    by (force simp: Let_def intro: exI[of _ VNext _])
next
  case (VNextOutL i I v φ)
  then show ?case
    by (auto intro!: exI[of _ VNextOutL i])
next
  case (VNextOutR i I v φ)
  then show ?case
    by (auto intro!: exI[of _ VNextOutR i])
next
  case (SOnce j i I v φ)
  then show ?case
    by (auto simp: Let_def intro: exI[of _ SOnce i _])
next
  case (VOnceOut i I v φ)
  then show ?case
    by (auto intro!: exI[of _ VOnceOut i])
next
  case (VOnce j I i v φ)
  show ?case
  proof
    assume future_bounded (P I φ)
    then have fb: future_bounded φ
      by simp
    obtain mypick where mypick_def: ∀ k ∈ {j .. LTP_p σ i I}. v_at (mypick k) = k ∧ v_check v φ
(mypick k)
      using VOnce fb by metis
    then obtain vps where vps_def: map (v_at) vps = [j ..< Suc (LTP_p σ i I)] ∧ (∀ vp ∈ set vps.
v_check v φ vp)
      by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([j ..< Suc
(LTP_p σ i I)])])
    then have v_at (VOnce i j vps) = i ∧ v_check v (P I φ) (VOnce i j vps)
      using VOnce by auto
    then show ∃ vp. v_at vp = i ∧ v_check v (P I φ) vp
      by blast
  qed
next
  case (SEventually j i I v φ)
  then show ?case
    by (auto simp: Let_def intro: exI[of _ SEventually i _])
next
  case (VEventually I i v φ)
  show ?case
  proof
    assume fb_eventually: future_bounded (F I φ)
    then have fb: future_bounded φ
      by simp
    obtain b where b_def: right I = enat b
      using fb_eventually by (atomize_elim, cases right I) auto
    define j where j_def: j = LTP σ (τ σ i + b)
```

obtain *mypick* **where** *mypick_def*: $\forall\, k \in \{ETP\_f\ \sigma\ i\ I\ ..\ j\}.\ v\_at\ (mypick\ k) = k \wedge v\_check\ v\ \varphi\ (mypick\ k)$
using *VEventually fb_eventually* **unfolding** *b_def j_def enat.simps*
by *atomize_elim* (*rule bchoice*, *simp*)
then obtain *vps* **where** *vps_def*: $map\ (v\_at)\ vps = [ETP\_f\ \sigma\ i\ I\ ..<\ Suc\ j] \wedge (\forall\, vp \in set\ vps.\ v\_check\ v\ \varphi\ vp)$
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*ETP_f* σ *i I ..< Suc j*])])
then have $v\_at\ (VEventually\ i\ j\ vps) = i \wedge v\_check\ v\ (\mathbf{F}\ I\ \varphi)\ (VEventually\ i\ j\ vps)$
using *VEventually b_def j_def* **by** *simp*
then show $\exists\, vp.\ v\_at\ vp = i \wedge v\_check\ v\ (\mathbf{F}\ I\ \varphi)\ vp$
by *blast*
qed
next
case (*SHistorically j I i v* φ)
show *?case*
proof
assume *fb_historically*: *future_bounded* (**H** *I* φ)
then have *fb*: *future_bounded* φ
by *simp*
obtain *mypick* **where** *mypick_def*: $\forall\, k \in \{j\ ..\ LTP\_p\ \sigma\ i\ I\}.\ s\_at\ (mypick\ k) = k \wedge s\_check\ v\ \varphi\ (mypick\ k)$
using *SHistorically fb* **by** *metis*
then obtain *sps* **where** *sps_def*: $map\ (s\_at)\ sps = [j\ ..<\ Suc\ (LTP\_p\ \sigma\ i\ I)] \wedge (\forall\, sp \in set\ sps.\ s\_check\ v\ \varphi\ sp)$
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*j ..< Suc* (*LTP_p* σ *i I*)])])
then have $s\_at\ (SHistorically\ i\ j\ sps) = i \wedge s\_check\ v\ (\mathbf{H}\ I\ \varphi)\ (SHistorically\ i\ j\ sps)$
using *SHistorically* **by** *auto*
then show $\exists\, sp.\ s\_at\ sp = i \wedge s\_check\ v\ (\mathbf{H}\ I\ \varphi)\ sp$
by *blast*
qed
next
case (*SHistoricallyOut i I v* φ)
then show *?case*
by (*auto intro*!: *exI*[*of* _ *SHistoricallyOut i*])
next
case (*VHistorically j i I v* φ)
then show *?case*
by (*auto simp*: *Let_def intro*: *exI*[*of* _ *VHistorically i* _])
next
case (*SAlways I i v* φ)
show *?case*
proof
assume *fb_always*: *future_bounded* (**G** *I* φ)
then have *fb*: *future_bounded* φ
by *simp*
obtain *b* **where** *b_def*: *right I* = *enat b*
using *fb_always* **by** (*atomize_elim*, *cases right I*) *auto*
define *j* **where** *j_def*: $j = LTP\ \sigma\ (\tau\ \sigma\ i + b)$
obtain *mypick* **where** *mypick_def*: $\forall\, k \in \{ETP\_f\ \sigma\ i\ I\ ..\ j\}.\ s\_at\ (mypick\ k) = k \wedge s\_check\ v\ \varphi\ (mypick\ k)$
using *SAlways fb_always* **unfolding** *b_def j_def enat.simps*
by *atomize_elim* (*rule bchoice*, *simp*)
then obtain *sps* **where** *sps_def*: $map\ (s\_at)\ sps = [ETP\_f\ \sigma\ i\ I\ ..<\ Suc\ j] \wedge (\forall\, sp \in set\ sps.\ s\_check\ v\ \varphi\ sp)$
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*ETP_f* σ *i I ..< Suc j*])])

```
    then have s_at (SAlways i j sps) = i ∧ s_check v (G I φ) (SAlways i j sps)
      using SAlways b_def j_def by simp
    then show ∃ sp. s_at sp = i ∧ s_check v (G I φ) sp
      by blast
  qed
next
  case (VAlways j i I v φ)
  then show ?case
    by (auto simp: Let_def intro: exI[of _ VAlways i _])
next
  case (SSince j i I v ψ φ)
  show ?case
  proof
    assume fb_since: future_bounded (φ S I ψ)
    then have fb: future_bounded φ future_bounded ψ
      by simp_all
    obtain sp2 where sp2_def: s_at sp2 = j ∧ s_check v ψ sp2
      using SSince fb_since by auto
    {
      assume Suc j > i
      then have s_at (SSince sp2 []) = i ∧ s_check v (φ S I ψ) (SSince sp2 [])
        using sp2_def SSince by auto
      then have ∃ sp. s_at sp = i ∧ s_check v (φ S I ψ) sp
        by blast
    }
    moreover
    {
      assume sucj_leq_i: Suc j ≤ i
      obtain mypick where mypick_def: ∀ k ∈ {Suc j ..< Suc i}. s_at (mypick k) = k ∧ s_check v φ
(mypick k)
        using SSince fb_since by atomize_elim (rule bchoice, simp)
      then obtain sp1s where sp1s_def: map (s_at) sp1s = [Suc j ..< Suc i] ∧ (∀ sp ∈ set sp1s. s_check
v φ sp)
        by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([Suc j
..< Suc i])])
      then have sp1s ≠ []
        using sucj_leq_i by auto
      then have s_at (SSince sp2 sp1s) = i ∧ s_check v (φ S I ψ) (SSince sp2 sp1s)
        using SSince sucj_leq_i fb sp2_def sp1s_def
        by (clarsimp simp add:
          Cons_eq_upt_conv append_eq_Cons_conv map_eq_append_conv
          split: list.splits) auto
      then have ∃ sp. s_at sp = i ∧ s_check v (φ S I ψ) sp
        by blast
    }
    ultimately show ∃ sp. s_at sp = i ∧ s_check v (φ S I ψ) sp
      using not_less by blast
  qed
next
  case (VSinceOut i I v φ ψ)
  then show ?case
    by (auto intro!: exI[of _ VSinceOut i])
next
  case (VSince I i j v φ ψ)
  show ?case
  proof
    assume fb_since: future_bounded (φ S I ψ)
    then have fb: future_bounded φ future_bounded ψ
```

by *simp_all*
obtain *vp1* **where** *vp1_def*: *v_at vp1 = j ∧ v_check v φ vp1*
using *fb_since VSince* **by** *auto*
obtain *mypick* **where** *mypick_def*: ∀ *k* ∈ {*j* .. *LTP_p σ i I*}. *v_at* (*mypick k*) = *k* ∧ *v_check v ψ* (*mypick k*)
using *VSince fb_since* **by** *atomize_elim* (*rule bchoice*, *simp*)
then obtain *vp2s* **where** *vp2s_def*: *map* (*v_at*) *vp2s* = [*j* ..< *Suc* (*LTP_p σ i I*)] ∧ (∀ *vp* ∈ *set vp2s*. *v_check v ψ vp*)
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*j* ..< *Suc* (*LTP_p σ i I*)])])
then have *v_at* (*VSince i vp1 vp2s*) = *i* ∧ *v_check v* (*φ* **S** *I ψ*) (*VSince i vp1 vp2s*)
using *vp1_def VSince* **by** *auto*
then show ∃ *vp*. *v_at vp* = *i* ∧ *v_check v* (*φ* **S** *I ψ*) *vp*
by *blast*
qed
next
case (*VSinceInf j I i v ψ φ*)
show *?case*
proof
assume *fb_since*: *future_bounded* (*φ* **S** *I ψ*)
then have *fb*: *future_bounded φ future_bounded ψ*
by *simp_all*
obtain *mypick* **where** *mypick_def*: ∀ *k* ∈ {*j* .. *LTP_p σ i I*}. *v_at* (*mypick k*) = *k* ∧ *v_check v ψ* (*mypick k*)
using *VSinceInf fb_since* **by** *atomize_elim* (*rule bchoice*, *simp*)
then obtain *vp2s* **where** *vp2s_def*: *map* (*v_at*) *vp2s* = [*j* ..< *Suc* (*LTP_p σ i I*)] ∧ (∀ *vp* ∈ *set vp2s*. *v_check v ψ vp*)
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*j* ..< *Suc* (*LTP_p σ i I*)])])
then have *v_at* (*VSinceInf i j vp2s*) = *i* ∧ *v_check v* (*φ* **S** *I ψ*) (*VSinceInf i j vp2s*)
using *VSinceInf* **by** *auto*
then show ∃ *vp*. *v_at vp* = *i* ∧ *v_check v* (*φ* **S** *I ψ*) *vp*
by *blast*
qed
next
case (*SUntil j i I v ψ φ*)
show *?case*
proof
assume *fb_until*: *future_bounded* (*φ* **U** *I ψ*)
then have *fb*: *future_bounded φ future_bounded ψ*
by *simp_all*
obtain *sp2* **where** *sp2_def*: *s_at sp2* = *j* ∧ *s_check v ψ sp2*
using *fb SUntil* **by** *blast*
{
assume *i* ≥ *j*
then have *s_at* (*SUntil* [] *sp2*) = *i* ∧ *s_check v* (*φ* **U** *I ψ*) (*SUntil* [] *sp2*)
using *sp2_def SUntil* **by** *auto*
then have ∃ *sp*. *s_at sp* = *i* ∧ *s_check v* (*φ* **U** *I ψ*) *sp*
by *blast*
}
moreover
{
assume *i_l_j*: *i* < *j*
obtain *mypick* **where** *mypick_def*: ∀ *k* ∈ {*i* ..< *j*}. *s_at* (*mypick k*) = *k* ∧ *s_check v φ* (*mypick k*)
using *SUntil fb_until* **by** *atomize_elim* (*rule bchoice*, *simp*)
then obtain *sp1s* **where** *sp1s_def*: *map* (*s_at*) *sp1s* = [*i* ..< *j*] ∧ (∀ *sp* ∈ *set sp1s*. *s_check v φ sp*)
by *atomize_elim* (*auto intro*!: *trans*[*OF list.map_cong list.map_id*] *exI*[*of* _ *map mypick* ([*i* ..< *j*])])

```
    then have s_at (SUntil sp1s sp2) = i ∧ s_check v (φ U I ψ) (SUntil sp1s sp2)
      using SUntil fb_until sp2_def sp1s_def i_l_j
      by (clarsimp simp add: append_eq_Cons_conv map_eq_append_conv split: list.splits)
        (auto simp: Cons_eq_upt_conv dest!: upt_eq_Nil_conv[THEN iffD1, OF sym])
    then have ∃ sp. s_at sp = i ∧ s_check v (φ U I ψ) sp
      by blast
   }
   ultimately show ∃ sp. s_at sp = i ∧ s_check v (φ U I ψ) sp
    using not_less by blast
 qed
next
 case (VUntil I j i v φ ψ)
 show ?case
 proof
   assume fb_until: future_bounded (φ U I ψ)
   then have fb: future_bounded φ future_bounded ψ
     by simp_all
   obtain vp1 where vp1_def: v_at vp1 = j ∧ v_check v φ vp1
     using VUntil fb_until by auto
   obtain mypick where mypick_def: ∀ k ∈ {ETP_f σ i I .. j}. v_at (mypick k) = k ∧ v_check v ψ
(mypick k)
     using VUntil fb_until by atomize_elim (rule bchoice, simp)
   then obtain vp2s where vp2s_def: map (v_at) vp2s = [ETP_f σ i I ..< Suc j] ∧ (∀ vp ∈ set vp2s.
v_check v ψ vp)
     by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([ETP_f
σ i I ..< Suc j])])
   then have v_at (VUntil i vp2s vp1) = i ∧ v_check v (φ U I ψ) (VUntil i vp2s vp1)
     using VUntil fb_until vp1_def by simp
   then show ∃ vp. v_at vp = i ∧ v_check v (φ U I ψ) vp
     by blast
 qed
next
 case (VUntilInf I i v ψ φ)
 show ?case
 proof
   assume fb_until: future_bounded (φ U I ψ)
   then have fb: future_bounded φ future_bounded ψ
     by simp_all
   obtain b where b_def: right I = enat b
     using fb_until by (atomize_elim, cases right I) auto
   define j where j_def: j = LTP σ (τ σ i + b)
   obtain mypick where mypick_def: ∀ k ∈ {ETP_f σ i I .. j}. v_at (mypick k) = k ∧ v_check v ψ
(mypick k)
     using VUntilInf fb_until unfolding b_def j_def by atomize_elim (rule bchoice, simp)
   then obtain vp2s where vp2s_def: map (v_at) vp2s = [ETP_f σ i I ..< Suc j] ∧ (∀ vp ∈ set vp2s.
v_check v ψ vp)
     by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([ETP_f
σ i I ..< Suc j])])
   then have v_at (VUntilInf i j vp2s) = i ∧ v_check v (φ U I ψ) (VUntilInf i j vp2s)
     using VUntilInf b_def j_def by simp
   then show ∃ vp. v_at vp = i ∧ v_check v (φ U I ψ) vp
     by blast
 qed
qed

lemmas check_completeness =
 conjunct1[OF check_completeness_aux, rule_format]
 conjunct2[OF check_completeness_aux, rule_format]
```

definition *p_check v φ p* = (*case p of Inl sp ⇒ s_check v φ sp | Inr vp ⇒ v_check v φ vp*)
definition *p_check_exec vs φ p* = (*case p of Inl sp ⇒ s_check_exec vs φ sp | Inr vp ⇒ v_check_exec vs φ vp*)

definition *valid* :: ('*n*, '*d*) *envset* ⇒ *nat* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *proof* ⇒ *bool* **where**
valid vs i φ p =
(*case p of*
Inl p ⇒ *s_check_exec vs φ p* ∧ *s_at p* = *i*
| *Inr p* ⇒ *v_check_exec vs φ p* ∧ *v_at p* = *i*)

end

8.7 Lifting the Checker to PDTs

fun *check_one* **where**
check_one σ v φ (*Leaf p*) = *p_check σ v φ p*
| *check_one σ v φ* (*Node x part*) = *check_one σ v φ* (*lookup_part part* (*v x*))

fun *check_all_aux* **where**
check_all_aux σ vs φ (*Leaf p*) = *p_check_exec σ vs φ p*
| *check_all_aux σ vs φ* (*Node x part*) = (∀ (*D*, *e*) ∈ *set* (*subsvals part*). *check_all_aux σ* (*vs*(*x* := *D*)) *φ e*)

fun *collect_paths_aux* **where**
collect_paths_aux DS σ vs φ (*Leaf p*) = (*if p_check_exec σ vs φ p then* {} *else rev* ‘ *DS*)
| *collect_paths_aux DS σ vs φ* (*Node x part*) = (⋃ (*D*, *e*) ∈ *set* (*subsvals part*). *collect_paths_aux* (*Cons D* ‘ *DS*) *σ* (*vs*(*x* := *D*)) *φ e*)

lemma *check_one_cong*: ∀ *x*∈*fv φ* ∪ *vars e*. *v x* = *v*′ *x* ⟹ *check_one σ v φ e* = *check_one σ v*′ *φ e*
proof (*induct e arbitrary*: *v v*′)
case (*Leaf x*)
then show *?case*
by (*auto simp*: *p_check_def check_fv_cong split*: *sum.splits*)
next
case (*Node x part*)
from *Node*(*2*) **have** ∗: *v x* = *v*′ *x*
by *simp*
from *Node*(*2*) **show** *?case*
unfolding *check_one.simps* ∗
by (*intro Node*(*1*)) *auto*
qed

lemma *check_all_aux_check_one*: ∀ *x*. *vs x* ≠ {} ⟹ *distinct_paths e* ⟹ (∀ *x* ∈ *vars e*. *vs x* = *UNIV*) ⟹
check_all_aux σ vs φ e ⟷ (∀ *v* ∈ *compatible_vals* (*fv φ*) *vs*. *check_one σ v φ e*)
proof (*induct e arbitrary*: *vs*)
case (*Node x part*)
show *?case*
unfolding *check_all_aux.simps check_one.simps split_beta*
proof (*safe*, *unfold fst_conv snd_conv*, *goal_cases LR RL*)
case (*LR v*)
from *Node*(*2*−) *fst_lookup*[*of v x part*] *LR*(*1*)[*rule_format*, *OF lookup_subsvals*[*of* _ *v x*]] *LR*(*2*)
show *?case*
by (*subst* (*asm*) *Node*(*1*))
(*auto 0 3 simp*: *compatible_vals_fun_upd dest*!: *bspec*[*of* _ _ *v*]
elim!: *compatible_vals_antimono*[*THEN set_mp*, *rotated*])
next

```
    case (RL D e)
    from RL(2) obtain d where d ∈ D
      by transfer (force simp: partition_on_def image_iff)
    with RL show ?case
      using Node(2−) lookup_subsvals[of part d] lookup_part_Vals[of part d]
        lookup_part_from_subvals[of D e part d]
    proof (intro Node(1)[THEN iffD2, OF _ _ _ _ ballI], goal_cases _ _ _ _ compatible)
      case (compatible v)
      from compatible(2−) compatible(1)[THEN bspec, of v(x := d)] compatible(1)[THEN bspec, of v]
      show ?case
        using lookup_part_from_subvals[of D e part v x]
          fun_upd_in_compatible_vals_in[of v fv φ x vs v x]
          check_one_cong[THEN iffD1, rotated −1, of σ v(x := d) φ e v, simplified]
        by (auto simp: compatible_vals_fun_upd fun_upd_apply[of _ _ _ x]
          fun_upd_in_compatible_vals_notin split: if_splits
          simp del: fun_upd_apply)
    qed auto
  qed
qed (auto simp: p_check_exec_def p_check_def check_exec_check split: sum.splits)

definition check_all :: ('n, 'd :: {default, linorder}) trace ⇒ ('n, 'd) formula ⇒ ('n, 'd) expl ⇒ bool
where
  check_all σ φ e = (distinct_paths e ∧ check_all_aux σ (λ_. UNIV) φ e)

lemma check_one_alt: check_one σ v φ e = p_check σ v φ (eval_pdt v e)
  by (induct e) auto

lemma check_all_alt: check_all σ φ e = (distinct_paths e ∧ (∀ v. p_check σ v φ (eval_pdt v e)))
  unfolding check_all_def
  by (rule conj_cong[OF refl], subst check_all_aux_check_one)
    (auto simp: compatible_vals_def check_one_alt)

fun pdt_at where
  pdt_at i (Leaf l) = (p_at l = i)
| pdt_at i (Node x part) = (∀ pdt ∈ Vals part. pdt_at i pdt)

lemma pdt_at_p_at_eval_pdt: pdt_at i e ⟹ p_at (eval_pdt v e) = i
  by (induct e) auto

lemma check_all_completeness_aux:
  fixes φ :: ('n, 'd :: {default, linorder}) formula
  shows set vs ⊆ fv φ ⟹ future_bounded φ ⟹ distinct vs ⟹
  ∃ e. pdt_at i e ∧ vars_order vs e ∧ (∀ v. (∀ x. x ∉ set vs ⟶ v x = w x) ⟶ p_check σ v φ (eval_pdt
v e))
proof (induct vs arbitrary: w)
  case Nil
  then show ?case
  proof (cases sat σ w i φ)
    case True
    then have SAT σ w i φ by (rule completeness)
    with Nil obtain sp where s_at sp = i s_check σ w φ sp by (blast dest: check_completeness)
    then show ?thesis
      by (intro exI[of _ Leaf (Inl sp)]) (auto simp: vars_order.intros p_check_def p_at_def)
  next
    case False
    then have VIO σ w i φ by (rule completeness)
    with Nil obtain vp where v_at vp = i v_check σ w φ vp by (blast dest: check_completeness)
    then show ?thesis
```

```
    by (intro exI[of _ Leaf (Inr vp)]) (auto simp: vars_order.intros p_check_def p_at_def)
  qed
next
  case (Cons x vs)
  define eq :: ('n ⇒ 'd) ⇒ ('n ⇒ 'd) ⇒ bool where eq = rel_fun (eq_onp (λx. x ∉ set vs)) (=)
  from Cons have ∀ w. ∃ e. pdt_at i e ∧ vars_order vs e ∧
    (∀ v. (∀ x. x ∉ set vs ⟶ v x = w x) ⟶ p_check σ v φ (eval_pdt v e)) by simp
  then obtain pick :: 'd ⇒ ('n, 'd) expl where pick: pdt_at i (pick a) vars_order vs (pick a) and
    eq_pick: ⋀v. eq v (w(x := a)) ⟹ p_check σ v φ (eval_pdt v (pick a)) for a
    unfolding eq_def rel_fun_def eq_onp_def choice_iff
  proof (atomize_elim, elim exE, goal_cases pick_val)
    case (pick_val f)
    then show ?case
      by (auto intro!: exI[of _ λa. f (w(x := a))])
  qed
  let ?a = SOME z. z ∉ AD σ φ i
  let ?AD = sorted_list_of_set (AD σ φ i)
  show ?case
  proof (intro exI[of _ Node x (tabulate ?AD pick (pick ?a))] conjI allI impI,
    goal_cases pdt_at vars_order p_check)
    case (p_check w')
    have w' x ∉ AD σ φ i ⟹ ?a ∉ AD σ φ i
      by (metis some_eq_imp)
    moreover have eq (w'(x := ?a)) (w(x := ?a))
      using p_check by (auto simp: eq_def rel_fun_def eq_onp_def)
    moreover have eq w' (w(x := w' x))
      using p_check by (auto simp: eq_def rel_fun_def eq_onp_def)
    ultimately show ?case
      using pick Cons(2−) eq_pick[of w' w' x] eq_pick[of w'(x := ?a) ?a]
        pdt_at_p_at_eval_pdt[of i pick ?a w'] eval_pdt_fun_upd[of vs pick ?a x w' ?a]
      by (auto simp: p_check_def p_at_def
        elim!: check_AD_cong[THEN iffD1, rotated −1, of _ _ _ _ _ _ i]
        split: if_splits sum.splits sum.splits)
  qed (use Cons(2−) pick in ‹simp_all add: vars_order.intros›)
qed

lemma check_all_completeness:
  fixes φ :: ('n, 'd :: {default, linorder}) formula
  assumes future_bounded φ
  shows ∃ e. pdt_at i e ∧ check_all σ φ e
proof −
  obtain vs where vs[simp]: distinct vs set vs = fv φ
    by (meson finite_distinct_list finite_fv)
  have s: s_check σ v φ sp
    if vars_order vs e
    and ∀ v. (∀ sp. eval_pdt v e = Inl sp ⟶ (∃ x. x ∉ fv φ ∧ v x ≠ undefined) ∨ s_check σ v φ sp) ∧
          (∀ vp. eval_pdt v e = Inr vp ⟶ (∃ x. x ∉ fv φ ∧ v x ≠ undefined) ∨ v_check σ v φ vp)
    and eval_pdt v e = Inl sp for e v sp
    using that eval_pdt_cong[of e v λx. if x ∈ fv φ then v x else undefined]
      check_fv_cong[of φ v λx. if x ∈ fv φ then v x else undefined]
    by (auto dest!: spec[of _ sp] vars_order_vars simp: subset_eq)
  have v: v_check σ v φ vp
    if vars_order vs e
    and ∀ v. (∀ sp. eval_pdt v e = Inl sp ⟶ (∃ x. x ∉ fv φ ∧ v x ≠ undefined) ∨ s_check σ v φ sp) ∧
          (∀ vp. eval_pdt v e = Inr vp ⟶ (∃ x. x ∉ fv φ ∧ v x ≠ undefined) ∨ v_check σ v φ vp)
    and eval_pdt v e = Inr vp for e v vp
    using that eval_pdt_cong[of e v λx. if x ∈ fv φ then v x else undefined]
      check_fv_cong[of φ v λx. if x ∈ fv φ then v x else undefined]
```

by (*auto dest*!: *spec*[*of* _ *vp*] *vars_order_vars simp*: *subset_eq*)
show *?thesis*
using *check_all_completeness_aux*[*of vs* φ *i* λ_. *undefined* σ] *assms*
unfolding *check_all_alt p_check_def*
by (*auto elim*!: *exI* [**where** *P* = λx. _ *x* $\wedge$ _ *x* , *OF conjI*] *simp*: *vars_order_distinct_paths split*: *sum.splits intro*: *s v*)
qed

lemma *check_all_soundness_aux*: *check_all* σ φ *e* $\Longrightarrow$ *p* = *eval_pdt v e* $\Longrightarrow$ *isl p* $\longleftrightarrow$ *sat* σ *v* (*p_at p*) φ
unfolding *check_all_alt*
by (*auto simp*: *isl_def p_check_def p_at_def dest*!: *spec*[*of* _ *v*]
dest: *check_soundness soundness split*: *sum.splits*)

lemma *check_all_soundness*: *check_all* σ φ *e* $\Longrightarrow$ *pdt_at i e* $\Longrightarrow$ *isl* (*eval_pdt v e*) $\longleftrightarrow$ *sat* σ *v i* φ
by (*drule check_all_soundness_aux*[*OF* _ *refl*, *of* _ _ _ *v*]) (*auto simp*: *pdt_at_p_at_eval_pdt*)

unbundle *MFOTL_no_notation* — disable notation

9 Type of Events

9.1 Code Adaptation for 8-bit strings

typedef *string8* = *UNIV* :: *char list set* **..**

setup_lifting *type_definition_string8*

lift_definition *empty_string* :: *string8* **is** [] .
lift_definition *string8_literal* :: *String.literal* $\Rightarrow$ *string8* **is** *String.explode* .
lift_definition *literal_string8*:: *string8* $\Rightarrow$ *String.literal* **is** *String.Abs_literal* .
declare [[*coercion string8_literal*]]

instantiation *string8* :: {*equal*, *linorder*}
begin

lift_definition *equal_string8* :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* **is** *HOL.equal* .
lift_definition *less_eq_string8* :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* **is** *ord_class.lexordp_eq* .
lift_definition *less_string8* :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* **is** *ord_class.lexordp* .

instance by *intro_classes*
(*transfer*; *auto simp*: *equal_eq lexordp_conv_lexordp_eq lexordp_eq_linear*
intro: *lexordp_eq_refl lexordp_eq_trans lexordp_eq_antisym*)+

end

lifting_forget *string8.lifting*

declare [[*code drop*: *literal_string8 string8_literal HOL.equal* :: *string8* $\Rightarrow$ _
($\leq$) :: *string8* $\Rightarrow$ _ (<) :: *string8* $\Rightarrow$ _
Code_Evaluation.term_of :: *string8* $\Rightarrow$ _]]

code_printing
type_constructor *string8* $\rightharpoonup$ (*OCaml*) *string*
| **constant** *HOL.equal* :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* $\rightharpoonup$ (*OCaml*) *Stdlib.*(=)
| **constant** ($\leq$) :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* $\rightharpoonup$ (*OCaml*) *Stdlib.*(<=)
| **constant** (<) :: *string8* $\Rightarrow$ *string8* $\Rightarrow$ *bool* $\rightharpoonup$ (*OCaml*) *Stdlib.*(<)
| **constant** *empty_string* :: *string8* $\rightharpoonup$ (*OCaml*)

| **constant** *string8_literal* :: *String.literal* ⇒ *string8* ⇀ (*OCaml*) *id*
| **constant** *literal_string8* :: *string8* ⇒ *String.literal* ⇀ (*OCaml*) *id*

ML ‹*structure String8 = struct fun to_term x = @{term Abs_string8} $ HOLogic.mk_string x; end;*›

code_printing
type_constructor *string8* ⇀ (*Eval*) *string*
| **constant** *string8_literal* :: *String.literal* ⇒ *string8* ⇀ (*Eval*) _
| **constant** *HOL.equal* :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* =
| **constant** (≤) :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* <=
| **constant** (<) :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* <
| **constant** *empty_string* :: *string8* ⇀ (*Eval*)
| **constant** *Code_Evaluation.term_of* :: *string8* ⇒ *term* ⇀ (*Eval*) *String8.to'_term*

ML ‹*structure String8 =struct fun to_term x = @{term Abs_string8} $ HOLogic.mk_string x; end;*›

code_printing
type_constructor *string8* ⇀ (*Eval*) *string*
| **constant** *string8_literal* :: *String.literal* ⇒ *string8* ⇀ (*Eval*) _
| **constant** *HOL.equal* :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* =
| **constant** (≤) :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* <=
| **constant** (<) :: *string8* ⇒ *string8* ⇒ *bool* ⇀ (*Eval*) **infixl** *6* <
| **constant** *Code_Evaluation.term_of* :: *string8* ⇒ *term* ⇀ (*Eval*) *String8.to'_term*

9.2 Event Parameters

definition *div_to_zero* :: *integer* ⇒ *integer* ⇒ *integer* **where**
div_to_zero x y = (*let z* = *fst* (*Code_Numeral.divmod_abs x y*) *in*
if (*x* < *0*) ≠ (*y* < *0*) *then* − *z else z*)

definition *mod_to_zero* :: *integer* ⇒ *integer* ⇒ *integer* **where**
mod_to_zero x y = (*let z* = *snd* (*Code_Numeral.divmod_abs x y*) *in*
if x < *0 then* − *z else z*)

lemma *b* ≠ *0* ⟹ *div_to_zero a b* ∗ *b* + *mod_to_zero a b* = *a*
unfolding *div_to_zero_def mod_to_zero_def Let_def*
by (*auto simp*: *minus_mod_eq_mult_div*[*symmetric*] *div_minus_right mod_minus_right ac_simps*)

datatype *event_data* = *EInt integer* | *EString string8*

instantiation *event_data* :: {*ord*, *plus*, *minus*, *uminus*, *times*, *divide*, *modulo*}
begin

fun *less_eq_event_data* **where**
EInt x ≤ *EInt y* ⟷ *x* ≤ *y*
| *EString x* ≤ *EString y* ⟷ *x* ≤ *y*
| *EInt* _ ≤ *EString* _ ⟷ *True*
| (_ :: *event_data*) ≤ _ ⟷ *False*

definition *less_event_data* :: *event_data* ⇒ *event_data* ⇒ *bool* **where**
less_event_data x y ⟷ *x* ≤ *y* ∧ ¬ *y* ≤ *x*

fun *plus_event_data* **where**
EInt x + *EInt y* = *EInt* (*x* + *y*)
| (_::*event_data*) + _ = *undefined*

fun *minus_event_data* **where**
EInt x − *EInt y* = *EInt* (*x* − *y*)

```
| (_::event_data) - _ = undefined

fun uminus_event_data where
  - EInt x = EInt (- x)
| - (_::event_data) = undefined

fun times_event_data where
  EInt x * EInt y = EInt (x * y)
| (_::event_data) * _ = undefined

fun divide_event_data where
  EInt x div EInt y = EInt (div_to_zero x y)
| (_::event_data) div _ = undefined

fun modulo_event_data where
  EInt x mod EInt y = EInt (mod_to_zero x y)
| (_::event_data) mod _ = undefined

instance ..

end

lemma infinite_UNIV_event_data:
  ¬finite (UNIV :: event_data set)
proof -
  define f where f = (λk. EInt k)
  have inj: inj_on f (UNIV :: integer set)
    unfolding f_def by (meson event_data.inject(1) injI)
  show ?thesis using finite_imageD[OF _ inj]
    by (meson infinite_UNIV_char_0 infinite_iff_countable_subset top_greatest)
qed

primrec integer_of_event_data :: event_data ⇒ integer where
  integer_of_event_data (EInt _) = undefined
| integer_of_event_data (EString _) = undefined

instantiation event_data :: default begin

definition default_event_data :: event_data where default = EInt 0

instance proof qed

end

instantiation event_data :: linorder begin
instance
proof (standard, unfold less_event_data_def, goal_cases less refl trans antisym total)
  case (refl x)
  then show ?case
    by (cases x) auto
next
  case (trans x y z)
  then show ?case
    by (cases x; cases y; cases z) auto
next
  case (antisym x y)
  then show ?case
    by (cases x; cases y) auto
```

next
case (*total x y*)
then show *?case*
by (*cases x*; *cases y*) *auto*
qed *simp*

end

10 Code Generation

10.1 Type Class Instances

class *universe* =
fixes *universe* :: *'a list option*
assumes *infinite*: *universe* = *None* ⟹ *infinite* (*UNIV* :: *'a set*)
and *finite*: *universe* = *Some xs* ⟹ *distinct xs* ∧ *set xs* = *UNIV*
begin

lemma *finite_coset*: *finite* (*List.coset* (*xs* :: *'a list*)) = (*case universe of None* ⇒ *False* | _ ⇒ *True*)
using *infinite finite*
by (*auto split*: *option.splits dest*!: *equalityD2 elim*!: *finite_subset*)

end

declare [[*code drop*: *finite*]]
declare *finite_set*[*THEN eqTrueI*, *code*] *finite_coset*[*code*]

instantiation *bool* :: *universe* **begin**
definition *universe_bool* :: *bool list option* **where** *universe_bool* = *Some* [*True*, *False*]
instance by *standard* (*auto simp*: *universe_bool_def*)
end
instantiation *char* :: *universe* **begin**
definition *universe_char* :: *char list option* **where** *universe_char* = *Some* (*map char_of* [*0*::*nat*..<*256*])
instance by *standard* (*use UNIV_char_of_nat* **in** ‹*auto simp*: *universe_char_def distinct_map*›)
end
instantiation *nat* :: *universe* **begin**
definition *universe_nat* :: *nat list option* **where** *universe_nat* = *None*
instance by *standard* (*auto simp*: *universe_nat_def*)
end
instantiation *list* :: (*type*) *universe* **begin**
definition *universe_list* :: *'a list list option* **where** *universe_list* = *None*
instance by *standard* (*auto simp*: *universe_list_def infinite_UNIV_listI*)
end
instantiation *String.literal* :: *universe* **begin**
definition *universe_literal* :: *String.literal list option* **where** *universe_literal* = *None*
instance by *standard* (*auto simp*: *universe_literal_def infinite_literal*)
end
instantiation *string8* :: *universe* **begin**
definition *universe_string8* :: *string8 list option* **where** *universe_string8* = *None*
lemma *UNIV_string8*: *UNIV* = *Abs_string8* ‘ *UNIV*
by (*auto simp*: *image_iff intro*: *Abs_string8_cases*)
instance by *standard*
(*auto simp*: *universe_string8_def UNIV_string8 finite_image_iff Abs_string8_inject inj_on_def infinite_UNIV_listI*)
end
instantiation *prod* :: (*universe*, *universe*) *universe* **begin**

definition *universe_prod* :: (*'a* × *'b*) *list option* **where** *universe_prod* =
(*case* (*universe*, *universe*) *of* (*Some xs*, *Some ys*) ⇒ *Some* (*List.product xs ys*) | _ ⇒ *None*)
instance by *standard*
(*auto simp*: *universe_prod_def finite_prod distinct_product infinite finite split*: *option.splits*)
end
instantiation *sum* :: (*universe*, *universe*) *universe* **begin**
definition *universe_sum* :: (*'a* + *'b*) *list option* **where** *universe_sum* =
(*case* (*universe*, *universe*) *of* (*Some xs*, *Some ys*) ⇒ *Some* (*map Inl xs* @ *map Inr ys*) | _ ⇒ *None*)
instance by *standard*
(*use UNIV_sum* **in** ‹*auto simp*: *universe_sum_def distinct_map infinite finite split*: *option.splits*›)
end
instantiation *option* :: (*universe*) *universe* **begin**
definition *universe_option* = (*case universe of Some xs* ⇒ *Some* (*None* # *map Some xs*) | _ ⇒ *None*)
instance by *standard* (*auto simp*: *universe_option_def distinct_map finite infinite image_iff split*: *option.splits*)
end
instantiation *fun* :: (*universe*, *universe*) *universe* **begin**
definition *universe_fun* :: (*'a* ⇒ *'b*) *list option* **where** *universe_fun* =
(*case* (*universe*, *universe*) *of*
(*Some xs*, *Some ys*) ⇒ *Some* (*map* (λ*zs*. *the* ∘ *map_of* (*zip xs zs*)) (*List.n_lists* (*length xs*) *ys*))
| (_, *Some* [*x*]) ⇒ *Some* [λ_. *x*]
| _ ⇒ *None*)
instance
proof −
have *1*: *False* **if** *infinite* (*UNIV*::*'a set*) *CARD*(*'b*) = *Suc 0 a* ≠ *b* **for** *a b* :: *'b*
using *that* **by** (*metis* (*full_types*) *UNIV_I card_1_singleton_iff singletonD*)
have *2*: *ys* = *zs*
if *distinct* (*xs*::*'a list*) **and** *length ys* = *length xs* **and** *length zs* = *length xs*
and ∀ *a*. *the* (*map_of* (*zip xs ys*) *a*) = *the* (*map_of* (*zip xs zs*) *a*)
for *xs* :: *'a list* **and** *ys zs* :: *'b list*
using *that* **by** (*metis map_fst_zip map_of_eqI map_of_zip_inject map_of_zip_is_None option.expand*)
have *3*: ∃ *zs*. *length zs* = *length xs* ∧ *set zs* ⊆ *set ys* ∧ (∀ *x*. *f x* = *the* (*map_of* (*zip xs zs*) *x*))
if ∀ *x*. *x* ∈ *set xs* ∀ *x*. *x* ∈ *set ys*
for *xs ys* **and** *f* :: *'a* ⇒ *'b*
using *that* **by** (*metis length_map map_of_zip_map option.sel subsetI*)
show *OFCLASS*(*'a* ⇒ *'b*, *universe_class*)
by *standard*
(*auto 0 3 simp*: *universe_fun_def set_eq_iff fun_eq_iff image_iff set_n_lists distinct_map*
inj_on_def distinct_n_lists finite_UNIV_fun dest!: *infinite finite*
split: *option.splits list.splits intro*: *1 2 3*)
qed
end
instantiation *event_data* :: *universe* **begin**
definition *universe_event_data* :: *event_data list option* **where** *universe_event_data* = *None*
instance by *standard* (*simp_all add*: *infinite_UNIV_event_data universe_event_data_def*)
end

instantiation *nat* :: *default* **begin**
definition *default_nat* :: *nat* **where** *default_nat* = *0*
instance proof qed
end

instantiation *list* :: (*type*) *default* **begin**
definition *default_list* :: *'a list* **where** *default_list* = []
instance proof qed
end

instance *event_data* :: *equal* **by** *standard*

instantiation *String.literal* :: *default* **begin**
definition *default_literal* :: *String.literal* **where** *default_literal* = *0*
instance proof qed
end

instantiation *event_data* :: *card_UNIV* **begin**
definition *finite_UNIV* = *Phantom*(*event_data*) *False*
definition *card_UNIV* = *Phantom*(*event_data*) *0*
instance by *intro_classes* (*simp_all add*: *finite_UNIV_event_data_def card_UNIV_event_data_def infinite_UNIV_event_data*)
end

10.2 Progress

primrec *progress* :: ('*n*, '*d*) *trace* ⇒ ('*n*, '*d*) *Formula.formula* ⇒ *nat* ⇒ *nat* **where**
  progress σ *Formula.TT* j = j
| *progress* σ *Formula.FF* j = j
| *progress* σ (*Formula.Eq_Const* _ _) j = j
| *progress* σ (*Formula.Pred* _ _) j = j
| *progress* σ (*Formula.Neg* φ) j = *progress* σ φ j
| *progress* σ (*Formula.Or* φ ψ) j = *min* (*progress* σ φ j) (*progress* σ ψ j)
| *progress* σ (*Formula.And* φ ψ) j = *min* (*progress* σ φ j) (*progress* σ ψ j)
| *progress* σ (*Formula.Imp* φ ψ) j = *min* (*progress* σ φ j) (*progress* σ ψ j)
| *progress* σ (*Formula.Iff* φ ψ) j = *min* (*progress* σ φ j) (*progress* σ ψ j)
| *progress* σ (*Formula.Exists* _ φ) j = *progress* σ φ j
| *progress* σ (*Formula.Forall* _ φ) j = *progress* σ φ j
| *progress* σ (*Formula.Prev I* φ) j = (*if* j = *0 then 0 else min* (*Suc* (*progress* σ φ j)) j)
| *progress* σ (*Formula.Next I* φ) j = *progress* σ φ j − *1*
| *progress* σ (*Formula.Once I* φ) j = *progress* σ φ j
| *progress* σ (*Formula.Historically I* φ) j = *progress* σ φ j
| *progress* σ (*Formula.Eventually I* φ) j =
    Inf {*i*. ∀ *k*. *k* < j ∧ *k* ≤ (*progress* σ φ j) ⟶ (τ σ *k* − τ σ *i*) ≤ *right I*}
| *progress* σ (*Formula.Always I* φ) j =
    Inf {*i*. ∀ *k*. *k* < j ∧ *k* ≤ (*progress* σ φ j) ⟶ (τ σ *k* − τ σ *i*) ≤ *right I*}
| *progress* σ (*Formula.Since* φ *I* ψ) j = *min* (*progress* σ φ j) (*progress* σ ψ j)
| *progress* σ (*Formula.Until* φ *I* ψ) j =
    Inf {*i*. ∀ *k*. *k* < j ∧ *k* ≤ *min* (*progress* σ φ j) (*progress* σ ψ j) ⟶ (τ σ *k* − τ σ *i*) ≤ *right I*}

lemma *Inf_Min*:
  fixes *P* :: *nat* ⇒ *bool*
  assumes *P j*
  shows *Inf* (*Collect P*) = *Min* (*Set.filter P* {..*j*})
  using *Min_in*[**where** *?A*=*Set.filter P* {..*j*}] *assms*
  by (*auto simp*: *Set.filter_def intro*: *cInf_lower intro*!: *antisym*[*OF* _ *Min_le*])
    (*metis Inf_nat_def1 empty_iff mem_Collect_eq*)

lemma *progress_Eventually_code*: *progress* σ (*Formula.Eventually I* φ) j =
  (*let m* = *min j* (*Suc* (*progress* σ φ j)) − *1 in Min* (*Set.filter* (λ*i*. *enat* (δ σ *m i*) ≤ *right I*) {..*j*}))
proof −
  define *P* **where** *P* ≡ (λ*i*. ∀ *k*. *k* < j ∧ *k* ≤ (*progress* σ φ j) ⟶ *enat* (δ σ *k i*) ≤ *right I*)
  have *P_j*: *P j*
    by (*auto simp*: *P_def enat_0*)
  have *all_wit*: (∀ *k* ∈ {..<*m*}. *enat* (δ σ *k i*) ≤ *right I*) ⟷ (*enat* (δ σ (*m* − *1*) *i*) ≤ *right I*) **for** *i m*
  proof (*cases m*)
    case (*Suc ma*)
    have *k* < *Suc ma* ⟹ δ σ *k i* ≤ δ σ *ma i* **for** *k*
      using τ_*mono*

```
    unfolding less_Suc_eq_le
    by (rule diff_le_mono)
  then show ?thesis
    by (auto simp: Suc) (meson order.trans enat_ord_simps(1))
 qed (auto simp: enat_0)
 show ?thesis
  unfolding progress.simps Let_def P_def[symmetric] Inf_Min[where ?P=P, OF P_j] all_wit[symmetric]
  by (fastforce simp: P_def intro: arg_cong[where ?f=Min])
qed

lemma progress_Always_code: progress σ (Formula.Always I φ) j =
 (let m = min j (Suc (progress σ φ j)) − 1 in Min (Set.filter (λi. enat (δ σ m i) ≤ right I) {..j}))
proof −
 define P where P ≡ (λi. ∀k. k < j ∧ k ≤ (progress σ φ j) ⟶ enat (δ σ k i) ≤ right I)
 have P_j: P j
  by (auto simp: P_def enat_0)
 have all_wit: (∀k ∈ {..<m}. enat (δ σ k i) ≤ right I) ⟷ (enat (δ σ (m − 1) i) ≤ right I) for i m
 proof (cases m)
  case (Suc ma)
  have k < Suc ma ⟹ δ σ k i ≤ δ σ ma i for k
   using τ_mono
   unfolding less_Suc_eq_le
   by (rule diff_le_mono)
  then show ?thesis
   by (auto simp: Suc) (meson order.trans enat_ord_simps(1))
 qed (auto simp: enat_0)
 show ?thesis
  unfolding progress.simps Let_def P_def[symmetric] Inf_Min[where ?P=P, OF P_j] all_wit[symmetric]
  by (fastforce simp: P_def intro: arg_cong[where ?f=Min])
qed

lemma progress_Until_code: progress σ (Formula.Until φ I ψ) j =
 (let m = min j (Suc (min (progress σ φ j) (progress σ ψ j))) − 1 in Min (Set.filter (λi. enat (δ σ m i)
≤ right I) {..j}))
proof −
 define P where P ≡ (λi. ∀k. k < j ∧ k ≤ min (progress σ φ j) (progress σ ψ j) ⟶ enat (δ σ k i) ≤
right I)
 have P_j: P j
  by (auto simp: P_def enat_0)
 have all_wit: (∀k ∈ {..<m}. enat (δ σ k i) ≤ right I) ⟷ (enat (δ σ (m − 1) i) ≤ right I) for i m
 proof (cases m)
  case (Suc ma)
  have k < Suc ma ⟹ δ σ k i ≤ δ σ ma i for k
   using τ_mono
   unfolding less_Suc_eq_le
   by (rule diff_le_mono)
  then show ?thesis
   by (auto simp: Suc) (meson order.trans enat_ord_simps(1))
 qed (auto simp: enat_0)
 show ?thesis
  unfolding progress.simps Let_def P_def[symmetric] Inf_Min[where ?P=P, OF P_j] all_wit[symmetric]
  by (fastforce simp: P_def intro: arg_cong[where ?f=Min])
qed

lemmas progress_code[code] = progress.simps(1−15) progress_Eventually_code progress_Always_code
progress.simps(18) progress_Until_code
```

10.3 Trace

lemma *snth_Stream_eq*: $(x \#\# s) !! n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ m \Rightarrow s !! m)$
by (*cases n*) *auto*

lemma *extend_is_stream*:
assumes *sorted* (*map snd list*)
and $\bigwedge x.\ x \in set\ list \Longrightarrow snd\ x \leq m$
and $\bigwedge x.\ x \in set\ list \Longrightarrow finite\ (fst\ x)$
shows *ssorted* (*smap snd* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*)) ∧
sincreasing (*smap snd* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*)) ∧
sfinite (*smap fst* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*))
proof −
have *A*: $\forall x{\in}set\ list.\ n \leq snd\ x \Longrightarrow n \leq m \Longrightarrow$
$n \leq$ (*map snd list* @− *smap* ($\lambda x.\ x + m$) *nats*) !! *i* **for** *n i*
and *list* :: (′*a set* × *nat*) *list*
proof (*induction i arbitrary*: *n*)
case (*Suc i*)
then show *?case*
by (*auto simp*: *shift_snth nth_tl*)
qed (*auto simp add*: *list.map_sel*(*1*))
then have *ssorted* (*smap snd* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*))
using *assms*
by (*induction list*) (*auto simp*: *stream.map_comp o_def ssorted_iff_mono snth_Stream_eq*
split: *nat.splits*)
moreover have *sincreasing* (*smap snd* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*))
using *assms*
proof (*induction list*)
case *Nil*
then show *?case*
by (*simp add*: *sincreasing_def*) *presburger*
next
case (*Cons a as*)
have *IH*: $\bigwedge x.\ \exists i.\ x <$ *smap snd* (*as* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*) !! *i*
using *Cons*
by (*simp add*: *sincreasing_def*)
show *?case*
using *IH*
by (*simp add*: *sincreasing_def*)
(*metis snth_Stream*)
qed
moreover have *sfinite* (*smap fst* (*list* @− *smap* ($\lambda n.$ ({}, $n + m$)) *nats*))
using *assms*(*3*)
proof (*induction list*)
case *Nil*
then show *?case* **by** (*simp add*: *sfinite_def*)
next
case (*Cons a as*)
then have *fin*: *finite* (*fst a*)
by *simp*
show *?case*
using *Cons*
by (*auto simp add*: *sfinite_def snth_Stream_eq split*: *nat.splits*)
qed
ultimately show *?thesis*
by *simp*
qed

typedef ′*a trace_mapping* = {(*n*, *m*, *t*) :: (*nat* × *nat* × (*nat*, ′*a set* × *nat*) *mapping*) |

```
  n m t. Mapping.keys t = {..<n} ∧
  sorted (map (snd ∘ (the ∘ Mapping.lookup t)) [0..<n]) ∧
  (case n of 0 ⇒ True | Suc n' ⇒ (case Mapping.lookup t n' of Some (X', t') ⇒ t' ≤ m | None ⇒ False))
∧
  (∀ n' < n. case Mapping.lookup t n' of Some (X', t') ⇒ finite X' | None ⇒ False)}
  by (rule exI[of _ (0, 0, Mapping.empty)]) auto

setup_lifting type_definition_trace_mapping

lemma lookup_bulkload_Some: i < length list ⟹
  Mapping.lookup (Mapping.bulkload list) i = Some (list ! i)
  by transfer auto

lift_definition trace_mapping_of_list :: ('a set × nat) list ⇒ 'a trace_mapping is
  λxs. if sorted (map snd xs) ∧ (∀ x ∈ set xs. finite (fst x)) then (if xs = [] then (0, 0, Mapping.empty)
  else (length xs, snd (last xs), Mapping.bulkload xs))
  else (0, 0, Mapping.empty)
  by (auto simp: lookup_bulkload_Some sorted_iff_nth_Suc last_conv_nth
    list_all_iff in_set_conv_nth Ball_def Bex_def image_iff split: nat.splits)

lift_definition trace_mapping_nth :: 'a trace_mapping ⇒ nat ⇒ ('a set × nat) is
  λ(n, m, t) i. if i < n then the (Mapping.lookup t i) else ({}, (i − n) + m) .

lift_definition Trace_Mapping :: 'a trace_mapping ⇒ 'a Trace.trace is
  λ(n, m, t). map (the ∘ Mapping.lookup t) [0..<n] @− smap (λn. ({} :: 'a set, n + m)) nats
proof (goal_cases)
  case (1 prod)
  obtain n m t where prod_def: prod = (n, m, t)
    by (cases prod) auto
  have props: Mapping.keys t = {..<n}
    sorted (map (snd ∘ (the ∘ Mapping.lookup t)) [0..<n])
    (case n of 0 ⇒ True | Suc n' ⇒ (case Mapping.lookup t n' of Some (X', t') ⇒ t' ≤ m | None ⇒
False))
    (∀ n' < n. case Mapping.lookup t n' of Some (X', t') ⇒ finite X' | None ⇒ False)
    using 1 by (auto simp add: prod_def)
  have aux: x ∈ set (map (the ∘ Mapping.lookup t) [0..<n]) ⟹ snd x ≤ m for x
    using props(2,3) less_Suc_eq_le
    by (fastforce simp: sorted_iff_nth_mono split: nat.splits option.splits)
  have aux2: x ∈ set (map (the ∘ Mapping.lookup t) [0..<n]) ⟹ finite (fst x) for x
    using props(1,4)
    by (auto split: nat.splits option.splits)
  show ?case
    unfolding prod_def prod.case
    by (rule extend_is_stream[where ?m=m]) (use props aux aux2 in ‹auto simp: prod_def›)
qed

code_datatype Trace_Mapping

definition trace_of_list xs = Trace_Mapping (trace_mapping_of_list xs)

lemma Γ_rbt_code[code]: Γ (Trace_Mapping t) i = fst (trace_mapping_nth t i)
  by transfer (auto split: prod.splits)

lemma τ_rbt_code[code]: τ (Trace_Mapping t) i = snd (trace_mapping_nth t i)
  by transfer (auto split: prod.splits)

lemma trace_mapping_of_list_sound: sorted (map snd xs) ∧ (∀ x ∈ set xs. finite (fst x)) ⟹ i < length
xs ⟹
```

xs ! *i* = (Γ (*trace_of_list xs*) *i*, τ (*trace_of_list xs*) *i*)
unfolding *trace_of_list_def*
by *transfer* (*auto simp*: *lookup_bulkload_Some*)

10.4 Auxiliary results

definition *sum_proofs f xs* = *sum_list* (*map f xs*)

lemma *sum_proofs_empty*[*simp*]: *sum_proofs f* [] = *0*
by (*auto simp*: *sum_proofs_def*)

lemma *sum_proofs_fundef_cong*[*fundef_cong*]: (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *f'* *x*) ⟹
sum_proofs f xs = *sum_proofs f'* *xs*
by (*induction xs*) (*auto simp*: *sum_proofs_def*)

lemma *sum_proofs_Cons*:
fixes *f* :: *'a* ⇒ *nat*
shows *sum_proofs f* (*p* # *qs*) = *f p* + *sum_proofs f qs*
by (*auto simp*: *sum_proofs_def split*: *list.splits*)

lemma *sum_proofs_app*:
fixes *f* :: *'a* ⇒ *nat*
shows *sum_proofs f* (*qs* @ [*p*]) = *f p* + *sum_proofs f qs*
by (*auto simp*: *sum_proofs_def split*: *list.splits*)

context
fixes *w* :: *'n* ⇒ *nat*
begin

function (*sequential*) *s_pred* :: (*'n*, *'d*) *sproof* ⇒ *nat*
and *v_pred* :: (*'n*, *'d*) *vproof* ⇒ *nat* **where**
s_pred (*STT* _) = *1*
| *s_pred* (*SEq_Const* _ _ _) = *1*
| *s_pred* (*SPred* _ *r* _) = *w r*
| *s_pred* (*SNeg vp*) = (*v_pred vp*) + *1*
| *s_pred* (*SOrL sp1*) = (*s_pred sp1*) + *1*
| *s_pred* (*SOrR sp2*) = (*s_pred sp2*) + *1*
| *s_pred* (*SAnd sp1 sp2*) = (*s_pred sp1*) + (*s_pred sp2*) + *1*
| *s_pred* (*SImpL vp1*) = (*v_pred vp1*) + *1*
| *s_pred* (*SImpR sp2*) = (*s_pred sp2*) + *1*
| *s_pred* (*SIffSS sp1 sp2*) = (*s_pred sp1*) + (*s_pred sp2*) + *1*
| *s_pred* (*SIffVV vp1 vp2*) = (*v_pred vp1*) + (*v_pred vp2*) + *1*
| *s_pred* (*SExists* _ _ *sp*) = (*s_pred sp*) + *1*
| *s_pred* (*SForall* _ *part*) = (*sum_proofs s_pred* (*vals part*)) + *1*
| *s_pred* (*SPrev sp*) = (*s_pred sp*) + *1*
| *s_pred* (*SNext sp*) = (*s_pred sp*) + *1*
| *s_pred* (*SOnce* _ *sp*) = (*s_pred sp*) + *1*
| *s_pred* (*SEventually* _ *sp*) = (*s_pred sp*) + *1*
| *s_pred* (*SHistorically* _ _ *sps*) = (*sum_proofs s_pred sps*) + *1*
| *s_pred* (*SHistoricallyOut* _) = *1*
| *s_pred* (*SAlways* _ _ *sps*) = (*sum_proofs s_pred sps*) + *1*
| *s_pred* (*SSince sp2 sp1s*) = (*sum_proofs s_pred* (*sp2* # *sp1s*)) + *1*
| *s_pred* (*SUntil sp1s sp2*) = (*sum_proofs s_pred* (*sp1s* @ [*sp2*])) + *1*
| *v_pred* (*VFF* _) = *1*
| *v_pred* (*VEq_Const* _ _ _) = *1*
| *v_pred* (*VPred* _ *r* _) = *w r*
| *v_pred* (*VNeg sp*) = (*s_pred sp*) + *1*
| *v_pred* (*VOr vp1 vp2*) = ((*v_pred vp1*) + (*v_pred vp2*)) + *1*

```
| v_pred (VAndL vp1) = (v_pred vp1) + 1
| v_pred (VAndR vp2) = (v_pred vp2) + 1
| v_pred (VImp sp1 vp2) = ((s_pred sp1) + (v_pred vp2)) + 1
| v_pred (VIffSV sp1 vp2) = ((s_pred sp1) + (v_pred vp2)) + 1
| v_pred (VIffVS vp1 sp2) = ((v_pred vp1) + (s_pred sp2)) + 1
| v_pred (VExists _ part) = (sum_proofs v_pred (vals part)) + 1
| v_pred (VForall _ _ vp) = (v_pred vp) + 1
| v_pred (VPrev vp) = (v_pred vp) + 1
| v_pred (VPrevZ) = 1
| v_pred (VPrevOutL _) = 1
| v_pred (VPrevOutR _) = 1
| v_pred (VNext vp) = (v_pred vp) + 1
| v_pred (VNextOutL _) = 1
| v_pred (VNextOutR _) = 1
| v_pred (VOnceOut _) = 1
| v_pred (VOnce _ _ vps) = (sum_proofs v_pred vps) + 1
| v_pred (VEventually _ _ vps) = (sum_proofs v_pred vps) + 1
| v_pred (VHistorically _ vp) = (v_pred vp) + 1
| v_pred (VAlways _ vp) = (v_pred vp) + 1
| v_pred (VSinceOut _) = 1
| v_pred (VSince _ vp1 vp2s) = (sum_proofs v_pred (vp1 # vp2s)) + 1
| v_pred (VSinceInf _ _ vp2s) = (sum_proofs v_pred vp2s) + 1
| v_pred (VUntil _ vp2s vp1) = (sum_proofs v_pred (vp2s @ [vp1])) + 1
| v_pred (VUntilInf _ _ vp2s) = (sum_proofs v_pred vp2s) + 1
  by pat_completeness auto
termination
  by (relation measure (case_sum size size))
    (auto simp add: termination_simp)

definition p_pred :: ('n, 'd) proof ⇒ nat where
  p_pred = case_sum s_pred v_pred

end
```

10.5 *v_check_exec* setup

```
lemma ETP_minus_le_iff: ETP σ (τ σ i − n) ≤ j ⟷ δ σ i j ≤ n
  by (simp add: add.commute i_ETP_tau le_diff_conv)

lemma ETP_minus_gt_iff: j < ETP σ (τ σ i − n) ⟷ δ σ i j > n
  by (meson ETP_minus_le_iff leD le_less_linear)

lemma nat_le_iff_less:
  fixes n :: nat
  shows (j ≤ n) ⟷ (j = 0 ∨ j − 1 < n)
  by auto

lemma ETP_minus_eq_iff: j = ETP σ (τ σ i − n) ⟷ ((j = 0 ∨ n < δ σ i (j − 1)) ∧ δ σ i j ≤ n)
  unfolding eq_iff[of j ETP σ (τ σ i − n)] nat_le_iff_less[of j] ETP_minus_le_iff ETP_minus_gt_iff
  by auto

lemma LTP_minus_ge_iff: τ σ 0 + n ≤ τ σ i ⟹ j ≤ LTP σ (τ σ i − n) ⟷
  (case n of 0 ⇒ δ σ j i = 0 | _ ⇒ j ≤ i ∧ δ σ i j ≥ n)
  using τ_mono[of i j σ]
  by (fastforce simp add: i_LTP_tau le_diff_conv2 Suc_le_eq split: nat.splits)

lemma LTP_plus_ge_iff: j ≤ LTP σ (τ σ i + n) ⟷ δ σ j i ≤ n
  by (simp add: add.commute i_LTP_tau le_diff_conv trans_le_add2)
```

lemma *LTP_minus_lt_if*:
assumes $j \leq i\ \tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i\ \delta\ \sigma\ i\ j < n$
shows $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j$
proof −
have *not_in*: $k \notin \{ia.\ \tau\ \sigma\ ia \leq \tau\ \sigma\ i - n\}$ **if** $j \leq k$ **for** k
using *assms* τ*_mono*[*OF that*, *of* σ]
by *auto*
then have $\{ia.\ \tau\ \sigma\ ia \leq \tau\ \sigma\ i - n\} \subseteq \{0..<j\}$
using *not_le_imp_less*
by (*clarsimp*; *blast*)
then have *finite* $\{ia.\ \tau\ \sigma\ ia \leq \tau\ \sigma\ i - n\}$
using *subset_eq_atLeast0_lessThan_finite*
by *blast*
moreover have $0 \in \{ia.\ \tau\ \sigma\ ia \leq \tau\ \sigma\ i - n\}$
using *assms*(*2*)
by *auto*
ultimately show *?thesis*
unfolding *LTP_def*
by (*metis Max_in not_in empty_iff not_le_imp_less*)
qed

lemma *LTP_minus_lt_iff*:
assumes $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i$
shows $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j \longleftrightarrow (\mathit{if}\ \neg\ j \leq i \wedge n = 0\ \mathit{then}\ \delta\ \sigma\ j\ i > 0\ \mathit{else}\ \delta\ \sigma\ i\ j < n)$
proof (*cases* $j \leq i$)
case *True*
then show *?thesis*
using *assms i_le_LTPi_minus*[*of* σ *n i*] *LTP_minus_lt_if*[*of j i* σ *n*]
by (*cases n*)
(*auto simp add*: *i_LTP_tau linorder_not_less Suc_le_eq dest*!: *tau_LTP_k*[*rotated*])
next
case *False*
have *delta*: $\delta\ \sigma\ i\ j = 0$
using *False*
by *auto*
show *?thesis*
proof (*cases n*)
case *0*
then show *?thesis*
using *False assms*
by (*metis add.right_neutral diff_is_0_eq diff_zero i_LTP_tau linorder_not_less*)
next
case (*Suc n'*)
then show *?thesis*
using *False assms*
by (*cases i*)
(*auto simp*: *Suc_le_eq not_le elim*!: *order.strict_trans*[*rotated*] *intro*!: *i_le_LTPi_minus*)
qed
qed

lemma *LTP_minus_eq_iff*:
assumes $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i$
shows $j = LTP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow$
(*case n of* $0 \Rightarrow i \leq j \wedge \delta\ \sigma\ j\ i = 0 \wedge \delta\ \sigma\ (Suc\ j)\ j > 0$
$\mid \_ \Rightarrow j \leq i \wedge n \leq \delta\ \sigma\ i\ j \wedge \delta\ \sigma\ i\ (Suc\ j) < n$)
proof (*cases n*)
case *0*

```
  show ?thesis
    using assms 0 i_LTP_tau[of σ τ σ i LTP σ (τ σ i)]
      i_LTP_tau[of σ τ σ i Suc (LTP σ (τ σ i))] i_LTP_tau[of σ τ σ i j]
      less_τD[of σ (LTP σ (τ σ i)) Suc j]
    by (auto simp: i_le_LTPi not_le elim!: antisym dest!:
      order_antisym_conv[of τ σ i τ σ j, THEN iffD1, rotated]
      order_antisym_conv[of τ σ i τ σ (LTP σ (τ σ i)), THEN iffD1, rotated])
next
  case (Suc n')
  show ?thesis
    using assms
    by (simp add: Suc eq_iff[of j LTP σ (τ σ i − Suc n')] less_Suc_eq_le[of LTP σ (τ σ i − Suc n') j,
symmetric] LTP_minus_ge_iff LTP_minus_lt_iff)
qed

lemma LTP_plus_eq_iff:
  shows j = LTP σ (τ σ i + n) ⟷ (δ σ j i ≤ n ∧ δ σ (Suc j) i > n)
  unfolding eq_iff[of j LTP σ (τ σ i + n)]
  by (meson LTP_plus_ge_iff linorder_not_less not_less_eq_eq)

lemma LTP_p_def: τ σ 0 + left I ≤ τ σ i ⟹ LTP_p σ i I = (case left I of 0 ⇒ i | _ ⇒ LTP σ (τ
σ i − left I))
  using i_le_LTPi by (auto simp: min_def i_LTP_tau split: nat.splits)

definition check_upt_LTP_p σ I li xs i ⟷ (case xs of [] ⇒
  (case left I of 0 ⇒ i < li | Suc n ⇒
    (if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I))
  | _ ⇒ xs = [li..<li + length xs] ∧
    (case left I of 0 ⇒ li + length xs − 1 = i | Suc n ⇒
      (li + length xs − 1 ≤ i ∧ left I ≤ δ σ i (li + length xs − 1) ∧ δ σ i (li + length xs) < left I)))

lemma check_upt_l_cong:
  assumes ⋀j. j ≤ max i li ⟹ τ σ j = τ σ' j
  shows check_upt_LTP_p σ I li xs i = check_upt_LTP_p σ' I li xs i
proof −
  have li + length ys ≤ i ⟹ Suc n ≤ δ σ' i (li + length ys) ⟹
    (Suc (li + length ys)) ≤ i for ys :: nat list and n
    by (cases li + length ys = i) auto
  then show ?thesis
    using assms
    by (fastforce simp: check_upt_LTP_p_def Let_def simp del: upt.simps split: list.splits nat.splits)
qed

lemma check_upt_LTP_p_eq:
  assumes τ σ 0 + left I ≤ τ σ i
  shows xs = [li..<Suc (LTP_p σ i I)] ⟷ check_upt_LTP_p σ I li xs i
proof −
  have li + length xs = Suc (LTP_p σ i I) ⟷ li + length xs − 1 = LTP_p σ i I if xs ≠ []
    using that
    by (cases xs) auto
  then have xs = [li..<Suc (LTP_p σ i I)] ⟷ (xs = [] ∧ LTP_p σ i I < li) ∨
    (xs ≠ [] ∧ xs = [li..<li + length xs] ∧ li + length xs − 1 = LTP_p σ i I)
    by auto
  moreover have ... ⟷ (xs = [] ∧
    (case left I of 0 ⇒ i < li | Suc n ⇒
      (if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I))) ∨
    (xs ≠ [] ∧ xs = [li..<li + length xs] ∧
    (case left I of 0 ⇒ li + length xs − 1 = i | Suc n ⇒
```

```
    (case left I of 0 ⇒ i ≤ li + length xs − 1 ∧
      δ σ (li + length xs − 1) i = 0 ∧ 0 < δ σ (Suc (li + length xs − 1)) (li + length xs − 1)
    | Suc n ⇒ li + length xs − 1 ≤ i ∧
      left I ≤ δ σ i (li + length xs − 1) ∧ δ σ i (Suc (li + length xs − 1)) < left I)))
  using LTP_p_def[OF assms, symmetric]
  unfolding LTP_minus_lt_iff[OF assms, symmetric]
  unfolding LTP_minus_eq_iff[OF assms, symmetric]
  by (auto split: nat.splits)
 moreover have ... ⟷ (case xs of [] ⇒
  (case left I of 0 ⇒ i < li | Suc n ⇒
    (if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I))
  | _ ⇒ xs = [li..<li + length xs] ∧
    (case left I of 0 ⇒ li + length xs − 1 = i | Suc n ⇒
      (li + length xs − 1 ≤ i ∧ left I ≤ δ σ i (li + length xs − 1) ∧ δ σ i (li + length xs) < left I)))
  by (auto split: nat.splits list.splits)
 ultimately show ?thesis
  unfolding check_upt_LTP_p_def
  by simp
qed

lemma v_check_exec_Once_code[code]: v_check_exec σ vs (Formula.Once I φ) vp = (case vp of
  VOnce i li vps ⇒
   (case right I of ∞ ⇒ li = 0 | enat b ⇒ ((li = 0 ∨ b < δ σ i (li − 1)) ∧ δ σ i li ≤ b))
   ∧ τ σ 0 + left I ≤ τ σ i
   ∧ check_upt_LTP_p σ I li (map v_at vps) i ∧ Ball (set vps) (v_check_exec σ vs φ)
 | VOnceOut i ⇒ τ σ i < τ σ 0 + left I
 | _ ⇒ False)
 by (auto simp: Let_def check_upt_LTP_p_eq ETP_minus_le_iff ETP_minus_eq_iff split: vproof.splits
enat.splits simp del: upt_Suc)

lemma s_check_exec_Historically_code[code]: s_check_exec σ vs (Formula.Historically I φ) vp = (case
vp of
  SHistorically i li vps ⇒
   (case right I of ∞ ⇒ li = 0 | enat b ⇒ ((li = 0 ∨ b < δ σ i (li − 1)) ∧ δ σ i li ≤ b))
   ∧ τ σ 0 + left I ≤ τ σ i
   ∧ check_upt_LTP_p σ I li (map s_at vps) i ∧ Ball (set vps) (s_check_exec σ vs φ)
 | SHistoricallyOut i ⇒ τ σ i < τ σ 0 + left I
 | _ ⇒ False)
 by (auto simp: Let_def check_upt_LTP_p_eq ETP_minus_le_iff ETP_minus_eq_iff split: sproof.splits
enat.splits simp del: upt_Suc)

lemma v_check_exec_Since_code[code]: v_check_exec σ vs (Formula.Since φ I ψ) vp = (case vp of
  VSince i vp1 vp2s ⇒
   let j = v_at vp1 in
   (case right I of ∞ ⇒ True | enat b ⇒ δ σ i j ≤ b) ∧ j ≤ i
   ∧ τ σ 0 + left I ≤ τ σ i
   ∧ check_upt_LTP_p σ I j (map v_at vp2s) i
   ∧ v_check_exec σ vs φ vp1 ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
 | VSinceInf i li vp2s ⇒
   (case right I of ∞ ⇒ li = 0 | enat b ⇒ ((li = 0 ∨ b < δ σ i (li − 1)) ∧ δ σ i li ≤ b)) ∧
   τ σ 0 + left I ≤ τ σ i ∧
    check_upt_LTP_p σ I li (map v_at vp2s) i ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
 | VSinceOut i ⇒ τ σ i < τ σ 0 + left I
 | _ ⇒ False)
 by (auto simp: Let_def check_upt_LTP_p_eq ETP_minus_le_iff ETP_minus_eq_iff split: vproof.splits
enat.splits simp del: upt_Suc)

lemma ETP_f_le_iff: max i (ETP σ (τ σ i + a)) ≤ j ⟷ i ≤ j ∧ δ σ j i ≥ a
```

```
  by (metis add.commute max.bounded_iff τ_mono i_ETP_tau le_diff_conv2)

lemma ETP_f_ge_iff: j ≤ max i (ETP σ (τ σ i + n)) ⟷ (case n of 0 ⇒ j ≤ i
  | Suc n' ⇒ (case j of 0 ⇒ True | Suc j' ⇒ δ σ j' i < n))
proof (cases n)
  case 0
  then show ?thesis
    by (auto simp: max_def) (metis i_ge_etpi verit_la_disequality)
next
  case (Suc n')
  have max: max i (ETP σ (τ σ i + n)) = ETP σ (τ σ i + n)
    by (auto simp: max_def Suc)
     (metis Groups.ab_semigroup_add_class.add.commute ETP_ge less_or_eq_imp_le plus_1_eq_Suc)
  have j ≤ max i (ETP σ (τ σ i + n)) ⟷ (∀ ia. τ σ i + n ≤ τ σ ia ⟶ j ≤ ia)
    unfolding max
    unfolding ETP_def
    by (meson LeastI_ex Least_le order.trans ex_le_τ)
  moreover have ... ⟷ (case j of 0 ⇒ True | Suc j' ⇒ ¬τ σ i + n ≤ τ σ j')
    by (auto split: nat.splits) (meson i_ETP_tau le_trans not_less_eq_eq)
  moreover have ... ⟷ (case j of 0 ⇒ True | Suc j' ⇒ δ σ j' i < n)
    by (auto simp: Suc split: nat.splits)
  ultimately show ?thesis
    by (auto simp: Suc)
qed

definition check_upt_ETP_f σ I i xs hi ⟷ (let j = Suc hi − length xs in
  (case xs of [] ⇒ (case left I of 0 ⇒ Suc hi ≤ i | Suc n' ⇒ δ σ hi i < left I)
  | _ ⇒ (xs = [j..<Suc hi] ∧
  (case left I of 0 ⇒ j ≤ i | Suc n' ⇒
  (case j of 0 ⇒ True | Suc j' ⇒ δ σ j' i < left I)) ∧
  i ≤ j ∧ left I ≤ δ σ j i)))

lemma check_upt_lu_cong:
  assumes ⋀j. min i hi ≤ j ∧ j ≤ max i hi ⟹ τ σ j = τ σ' j
  shows check_upt_ETP_f σ I i xs hi = check_upt_ETP_f σ' I i xs hi
  using assms
  unfolding check_upt_ETP_f_def
  by (auto simp add: Let_def le_Suc_eq split: list.splits nat.splits)

lemma check_upt_ETP_f_eq: xs = [ETP_f σ i I..<Suc hi] ⟷ check_upt_ETP_f σ I i xs hi
proof −
  have F1: (case left I of 0 ⇒ Suc hi ≤ i | Suc n' ⇒ δ σ hi i < left I) =
    (Suc hi ≤ ETP_f σ i I)
    unfolding ETP_f_ge_iff[where ?j=Suc hi and ?n=left I]
    by (auto split: nat.splits)
  have xs = [ETP_f σ i I..<Suc hi] ⟷ (let j = Suc hi − length xs in
    (xs = [] ∧ (case left I of 0 ⇒ Suc hi ≤ i | Suc n' ⇒ δ σ hi i < left I)) ∨
    (xs ≠ [] ∧ xs = [j..<Suc hi] ∧
    (case left I of 0 ⇒ j ≤ i | Suc n' ⇒
    (case j of 0 ⇒ True | Suc j' ⇒ δ σ j' i < left I)) ∧
    i ≤ j ∧ left I ≤ δ σ j i))
    unfolding F1
    unfolding Let_def
    unfolding ETP_f_ge_iff[where ?n=left I, symmetric]
    unfolding ETP_f_le_iff[symmetric]
    unfolding eq_iff[of _ ETP_f σ i I, symmetric]
    by auto
  moreover have ... ⟷ (let j = Suc hi − length xs in
```

```
    (case xs of [] ⇒ (case left I of 0 ⇒ Suc hi ≤ i | Suc n' ⇒ δ σ hi i < left I)
    | _ ⇒ (xs = [j..<Suc hi] ∧
    (case left I of 0 ⇒ j ≤ i | Suc n' ⇒
    (case j of 0 ⇒ True | Suc j' ⇒ δ σ j' i < left I)) ∧
    i ≤ j ∧ left I ≤ δ σ j i)))
    by (auto simp: Let_def split: list.splits)
  finally show ?thesis
    unfolding check_upt_ETP_f_def .
qed

lemma v_check_exec_Eventually_code[code]: v_check_exec σ vs (Formula.Eventually I φ) vp = (case
vp of
  VEventually i hi vps ⇒
    (case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i)) ∧
     check_upt_ETP_f σ I i (map v_at vps) hi ∧ Ball (set vps) (v_check_exec σ vs φ)
  | _ ⇒ False)
  by (auto simp: Let_def LTP_plus_ge_iff LTP_plus_eq_iff check_upt_ETP_f_eq simp del: upt_Suc
      split: vproof.splits enat.splits)

lemma s_check_exec_Always_code[code]: s_check_exec σ vs (Formula.Always I φ) sp = (case sp of
  SAlways i hi sps ⇒
    (case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i))
    ∧ check_upt_ETP_f σ I i (map s_at sps) hi ∧ Ball (set sps) (s_check_exec σ vs φ)
  | _ ⇒ False)
  by (auto simp: Let_def LTP_plus_ge_iff LTP_plus_eq_iff check_upt_ETP_f_eq simp del: upt_Suc
      split: sproof.splits enat.splits)

lemma v_check_exec_Until_code[code]: v_check_exec σ vs (Formula.Until φ I ψ) vp = (case vp of
  VUntil i vp2s vp1 ⇒
    let j = v_at vp1 in
    (case right I of ∞ ⇒ True | enat b ⇒ j < LTP_f σ i b)
    ∧ i ≤ j ∧ check_upt_ETP_f σ I i (map v_at vp2s) j
    ∧ v_check_exec σ vs φ vp1 ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
| VUntilInf i hi vp2s ⇒
    (case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i)) ∧
     check_upt_ETP_f σ I i (map v_at vp2s) hi ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
| _ ⇒ False)
  by (auto simp: Let_def LTP_plus_ge_iff LTP_plus_eq_iff check_upt_ETP_f_eq simp del: upt_Suc
      split: vproof.splits enat.splits)
```

10.6 ETP/LTP setup

```
lemma ETP_aux: ¬ t ≤ τ σ i ⟹ i ≤ (LEAST i. t ≤ τ σ i)
  by (meson LeastI_ex τ_mono ex_le_τ nat_le_linear order_trans)

function ETP_rec where
  ETP_rec σ t i = (if τ σ i ≥ t then i else ETP_rec σ t (i + 1))
  by pat_completeness auto
termination
  using ETP_aux
  by (relation measure (λ(σ, t, i). Suc (ETP σ t) − i))
    (fastforce simp: ETP_def)+

lemma ETP_rec_sound: ETP_rec σ t j = (LEAST i. i ≥ j ∧ t ≤ τ σ i)
proof (induction σ t j rule: ETP_rec.induct)
  case (1 σ t i)
  show ?case
  proof (cases τ σ i ≥ t)
```

```
    case True
    then show ?thesis
      by simp (metis (no_types, lifting) Least_equality order_refl)
  next
    case False
    then show ?thesis
      using 1[OF False]
      by (simp add: ETP_rec.simps[of _ _ i] del: ETP_rec.simps)
        (metis Suc_leD le_antisym not_less_eq_eq)
  qed
qed

lemma ETP_code[code]: ETP σ t = ETP_rec σ t 0
  using ETP_rec_sound[of σ t 0]
  by (auto simp: ETP_def)

lemma LTP_aux:
  assumes τ σ (Suc i) ≤ t
  shows i ≤ Max {i. τ σ i ≤ t}
proof −
  have finite {i. τ σ i ≤ t}
   by (smt (verit, del_insts) τ_mono finite_nat_set_iff_bounded_le i_LTP_tau le0 le_trans mem_Collect_eq)

  moreover have i ∈ {i. τ σ i ≤ t}
    using le_trans[OF τ_mono[of i Suc i σ] assms]
    by auto
  ultimately show ?thesis
    by (rule Max_ge)
qed

function (sequential) LTP_rec where
  LTP_rec σ t i = (if τ σ (Suc i) ≤ t then LTP_rec σ t (i + 1) else i)
  by pat_completeness auto
termination
  using LTP_aux
  by (relation measure (λ(σ, t, i). Suc (LTP σ t) − i)) (fastforce simp: LTP_def)+

lemma LTP_rec_sound: LTP_rec σ t j = Max ({i. i ≥ j ∧ (τ σ i) ≤ t} ∪ {j})
proof (induction σ t j rule: LTP_rec.induct)
  case (1 σ t j)
  have fin: finite {i. j ≤ i ∧ τ σ i ≤ t}
    by (smt (verit, del_insts) τ_mono finite_nat_set_iff_bounded_le i_LTP_tau le0 le_trans
      mem_Collect_eq)
  show ?case
  proof (cases τ σ (Suc j) ≤ t)
    case True
    have diffI: {i. Suc j ≤ i ∧ τ σ i ≤ t} = {i. j ≤ i ∧ τ σ i ≤ t} − {j}
      by auto
    show ?thesis
      using 1[OF True] True fin
      by (auto simp del: LTP_rec.simps simp add: LTP_rec.simps[of _ _ j] diffI intro: max_aux)
  next
    case False
    then show ?thesis
      using fin
      by (auto simp: not_le intro!: Max_insert2[symmetric]
        dest!: order.strict_trans1 less_τD)
  qed
```

qed

lemma *LTP_code*[*code*]: *LTP σ t* = (*if t* < *τ σ 0*
then Code.abort (*STR ''LTP: undefined''*) (λ_. *LTP σ t*)
else LTP_rec σ t 0)
using *LTP_rec_sound*[*of σ t 0*]
by (*auto simp*: *LTP_def insert_absorb simp del*: *LTP_rec.simps*)

lemma *map_part_code*[*code*]: *Rep_part* (*map_part f xs*) = *map* (*map_prod id f*) (*Rep_part xs*)
using *Rep_part*[*of xs*]
by (*auto simp*: *map_part_def intro*!: *Abs_part_inverse*)

lemma *coset_subset_set_code*[*code*]:
(*List.coset* (*xs* :: _ :: *universe list*) ⊆ *set ys*) = (*case universe of None* ⇒ *False*
| *Some zs* ⇒ ∀ *z* ∈ *set zs*. *z* ∈ *set xs* ∨ *z* ∈ *set ys*)
using *finite_compl finite_subset*
by (*auto split*: *option.splits dest*!: *infinite finite*)

lemma *is_empty_coset*[*code*]: *Set.is_empty* (*List.coset* (*xs* :: _ :: *universe list*)) =
(*case universe of None* ⇒ *False*
| *Some zs* ⇒ ∀ *z* ∈ *set zs*. *z* ∈ *set xs*)
using *coset_subset_set_code*[*of xs*] **by** (*auto simp*: *Set.is_empty_def split*: *option.splits dest*: *infinite finite*)

10.7 Exported functions

type_synonym *name* = *string8*

declare *Formula.future_bounded.simps*[*code*]

definition *collect_paths* :: ('*n*, '*d* :: {*default*, *linorder*}) *trace* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *expl* ⇒ '*d set list set option* **where**
collect_paths σ φ e = (*if* (*distinct_paths e* ∧ *check_all_aux σ* (λ_. *UNIV*) *φ e*) *then None else Some* (*collect_paths_aux* {[]} *σ* (λ_. *UNIV*) *φ e*))

definition *check* :: (*name*, *event_data*) *trace* ⇒ (*name*, *event_data*) *formula* ⇒ (*name*, *event_data*) *expl* ⇒ *bool* **where**
check = *check_all*

definition *collect_paths_specialized* :: (*name*, *event_data*) *trace* ⇒ (*name*, *event_data*) *formula* ⇒ (*name*, *event_data*) *expl* ⇒ *event_data set list set option* **where**
collect_paths_specialized = *collect_paths*

definition *trace_of_list_specialized* :: ((*name* × *event_data list*) *set* × *nat*) *list* ⇒ (*name*, *event_data*) *trace* **where**
trace_of_list_specialized xs = *trace_of_list xs*

definition *specialized_set* :: (*name* × *event_data list*) *list* ⇒ (*name* × *event_data list*) *set* **where**
specialized_set = *set*

definition *ed_set* :: *event_data list* ⇒ *event_data set* **where**
ed_set = *set*

definition *sum_nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
sum_nat m n = *m* + *n*

definition *sub_nat* :: *nat* ⇒ *nat* ⇒ *nat* **where**
sub_nat m n = *m* − *n*

lift_definition *abs_part* :: (*event_data set* × *'a*) *list* ⇒ (*event_data*, *'a*) *part* **is**
λxs.
let Ds = map fst xs in
if {} ∈ set Ds
∨ (∃ D ∈ set Ds. ∃ E ∈ set Ds. D ≠ E ∧ D ∩ E ≠ {})
∨ ¬ distinct Ds
∨ (⋃ D ∈ set Ds. D) ≠ UNIV then [(UNIV, undefined)] else xs
by (*auto simp*: *partition_on_def disjoint_def*)

export_code *interval enat nat_of_integer integer_of_nat*
STT Formula.TT Inl EInt Formula.Var Leaf set part_hd sum_nat sub_nat subsvals
check trace_of_list_specialized specialized_set ed_set abs_part
collect_paths_specialized
in *OCaml* **module_name** *Checker* **file_prefix** *checker*

11 Unverified Explanation-Producing Monitoring Algorithm

fun *merge_part2_raw* :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'d set* × *'a*) *list* ⇒ (*'d set* × *'b*) *list* ⇒ (*'d set* × *'c*) *list*
where
merge_part2_raw f [] _ = []
| *merge_part2_raw f ((P1, v1) # part1) part2 =*
(let part12 = List.map_filter (λ(P2, v2). if P1 ∩ P2 ≠ {} then Some(P1 ∩ P2, f v1 v2) else None) part2 in
let part2not1 = List.map_filter (λ(P2, v2). if P2 − P1 ≠ {} then Some(P2 − P1, v2) else None) part2 in
part12 @ (merge_part2_raw f part1 part2not1))

fun *merge_part3_raw* :: (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'e*) ⇒ (*'d set* × *'a*) *list* ⇒ (*'d set* × *'b*) *list* ⇒ (*'d set* × *'c*) *list* ⇒ (*'d set* × *'e*) *list* **where**
merge_part3_raw f [] _ _ = []
| *merge_part3_raw f _ [] _ = []*
| *merge_part3_raw f _ _ [] = []*
| *merge_part3_raw f part1 part2 part3 = merge_part2_raw (λpt3 f'. f' pt3) part3 (merge_part2_raw f part1 part2)*

lemma *partition_on_empty_iff*:
partition_on X P ⟹ P = {} ⟷ X = {}
partition_on X P ⟹ P ≠ {} ⟷ X ≠ {}
by (*auto simp*: *partition_on_def*)

lemma *wf_part_list_filter_inter*:
defines *inP1 P1 f v1 part2*
≡ List.map_filter (λ(P2, v2). if P1 ∩ P2 ≠ {} then Some(P1 ∩ P2, f v1 v2) else None) part2
assumes *partition_on X (set (map fst ((P1, v1) # part1)))*
and *partition_on X (set (map fst part2))*
shows *partition_on P1 (set (map fst (inP1 P1 f v1 part2)))*
and *distinct (map fst ((P1, v1) # part1)) ⟹ distinct (map fst (part2)) ⟹*
distinct (map fst (inP1 P1 f v1 part2))
proof (*rule partition_onI*)
show ⋃ (*set (map fst (inP1 P1 f v1 part2))*) = *P1*
proof −
have ∃ *P2*. (*P1 ∩ P2 ≠ {} ⟶* (∃ *v2*. (*P2*, *v2*) ∈ *set part2*) ∧ *x ∈ P2*) ∧ *P1 ∩ P2 ≠ {}*
if ⋃ (*fst ' set part2*) = *P1 ∪* ⋃ (*fst ' set part1*) **and** *x ∈ P1* **for** *x*
using *that* **by** (*metis* (*no_types*, *lifting*) *Int_iff UN_iff Un_Int_eq*(*3*) *empty_iff prod.collapse*)
with *partition_onD1*[*OF assms*(*2*)] *partition_onD1*[*OF assms*(*3*)] **show** *?thesis*
by (*auto simp*: *map_filter_def inP1_def split*: *if_splits*)

qed
show $\bigwedge A1\ A2.\ A1 \in set\ (map\ fst\ (inP1\ P1\ f\ v1\ part2)) \Longrightarrow$
$A2 \in set\ (map\ fst\ (inP1\ P1\ f\ v1\ part2)) \Longrightarrow A1 \neq A2 \Longrightarrow disjnt\ A1\ A2$
using *partition_onD2*[*OF assms*(*2*)] *partition_onD2*[*OF assms*(*3*)]
by (*force simp*: *disjnt_iff map_filter_def disjoint_def inP1_def split*: *if_splits*)
show $\{\} \notin set\ (map\ fst\ (inP1\ P1\ f\ v1\ part2))$
using *assms*
by (*auto simp*: *map_filter_def split*: *if_splits*)
show *distinct* (*map fst* ((*P1*, *v1*) # *part1*)) $\Longrightarrow$ *distinct* (*map fst part2*) $\Longrightarrow$
distinct (*map fst* (*inP1 P1 f v1 part2*))
using *partition_onD2*[*OF assms*(*3*), *unfolded disjoint_def*, *simplified*, *rule_format*]
by (*clarsimp simp*: *inP1_def map_filter_def distinct_map inj_on_def Ball_def*) *blast*
qed

lemma *wf_part_list_filter_minus*:
defines *notinP2 P1 f v1 part2*
$\equiv$ *List.map_filter* (λ(*P2*, *v2*). *if P2* − *P1* $\neq \{\}$ *then Some*(*P2* − *P1*, *v2*) *else None*) *part2*
assumes *partition_on X* (*set* (*map fst* ((*P1*, *v1*) # *part1*)))
and *partition_on X* (*set* (*map fst part2*))
shows *partition_on* (*X* − *P1*) (*set* (*map fst* (*notinP2 P1 f v1 part2*)))
and *distinct* (*map fst* ((*P1*, *v1*) # *part1*)) $\Longrightarrow$ *distinct* (*map fst* (*part2*)) $\Longrightarrow$
distinct (*map fst* (*notinP2 P1 f v1 part2*))
proof (*rule partition_onI*)
show $\bigcup$ (*set* (*map fst* (*notinP2 P1 f v1 part2*))) = *X* − *P1*
proof −
have $\exists\, P2.\ ((\exists\, x{\in}P2.\ x \notin P1) \longrightarrow (\exists\, v2.\ (P2, v2) \in set\ part2)) \wedge (\exists\, x{\in}P2.\ x \notin P1) \wedge x \in P2$
if $\bigcup$ (*fst* ‘ *set part2*) = *P1* $\cup \bigcup$ (*fst* ‘ *set part1*) *x* $\notin$ *P1* (*P1*′, *v1*) $\in$ *set part1 x* $\in$ *P1*′ **for** *x P1*′ *v1*
using *that* **by** (*metis* (*no_types*, *lifting*) *UN_iff Un_iff fst_conv prod.collapse*)
with *partition_onD1*[*OF assms*(*2*)] *partition_onD1*[*OF assms*(*3*)] **show** *?thesis*
by (*auto simp*: *map_filter_def subset_eq split_beta notinP2_def split*: *if_splits*)
qed
show $\bigwedge A1\ A2.\ A1 \in set\ (map\ fst\ (notinP2\ P1\ f\ v1\ part2)) \Longrightarrow$
$A2 \in set\ (map\ fst\ (notinP2\ P1\ f\ v1\ part2)) \Longrightarrow A1 \neq A2 \Longrightarrow disjnt\ A1\ A2$
using *partition_onD2*[*OF assms*(*3*)]
by (*auto simp*: *disjnt_def map_filter_def disjoint_def notinP2_def Ball_def Bex_def image_iff split*: *if_splits*)
show $\{\} \notin set\ (map\ fst\ (notinP2\ P1\ f\ v1\ part2))$
using *assms*
by (*auto simp*: *map_filter_def split*: *if_splits*)
show *distinct* (*map fst* ((*P1*, *v1*) # *part1*)) $\Longrightarrow$ *distinct* (*map fst part2*) $\Longrightarrow$
distinct (*map fst* ((*notinP2 P1 f v1 part2*)))
using *partition_onD2*[*OF assms*(*3*), *unfolded disjoint_def*]
by (*clarsimp simp*: *notinP2_def map_filter_def distinct_map inj_on_def Ball_def Bex_def image_iff*) *blast*
qed

lemma *wf_part_list_tail*:
assumes *partition_on X* (*set* (*map fst* ((*P1*, *v1*) # *part1*)))
and *distinct* (*map fst* ((*P1*, *v1*) # *part1*))
shows *partition_on* (*X* − *P1*) (*set* (*map fst part1*))
and *distinct* (*map fst part1*)
proof (*rule partition_onI*)
show $\bigcup$ (*set* (*map fst part1*)) = *X* − *P1*
using *partition_onD1*[*OF assms*(*1*)] *partition_onD2*[*OF assms*(*1*)] *assms*(*2*)
by (*auto simp*: *disjoint_def image_iff*)
show $\bigwedge A1\ A2.\ A1 \in set\ (map\ fst\ part1) \Longrightarrow A2 \in set\ (map\ fst\ part1) \Longrightarrow A1 \neq A2 \Longrightarrow disjnt\ A1\ A2$
using *partition_onD2*[*OF assms*(*1*)]

```
    by (clarsimp simp: disjnt_def disjoint_def)
      (smt (verit, ccfv_SIG) Diff_disjoint Int_Diff Int_commute fst_conv)
  show {} ∉ set (map fst part1)
    using partition_onD3[OF assms(1)]
    by (auto simp: map_filter_def split: if_splits)
  show distinct (map fst (part1))
    using assms(2)
    by auto
qed

lemma partition_on_append: partition_on X (set xs) ⟹ partition_on Y (set ys) ⟹ X ∩ Y = {} ⟹
  partition_on (X ∪ Y) (set (xs @ ys))
  by (auto simp: partition_on_def intro!: disjoint_union)

lemma wf_part_list_merge_part2_raw:
  partition_on X (set (map fst part1)) ∧ distinct (map fst part1) ⟹
   partition_on X (set (map fst part2)) ∧ distinct (map fst part2) ⟹
   partition_on X (set (map fst (merge_part2_raw f part1 part2)))
    ∧ distinct (map fst (merge_part2_raw f part1 part2))
proof(induct f part1 part2 arbitrary: X rule: merge_part2_raw.induct)
  case (2 f P1 v1 part1 part2)
  let ?inP1 = List.map_filter (λ(P2, v2). if P1 ∩ P2 ≠ {} then Some (P1 ∩ P2, f v1 v2) else None)
part2
    and ?notinP1 = List.map_filter (λ(P2, v2). if P2 − P1 ≠ {} then Some (P2 − P1, v2) else None)
part2
  have P1 ∪ X = X
    using 2.prems
    by (auto simp: partition_on_def)
  have wf_part1: partition_on (X − P1) (set (map fst part1))
    distinct (map fst part1)
    using wf_part_list_tail 2.prems by auto
  moreover have wf_notinP1: partition_on (X − P1) (set (map fst ?notinP1))
    distinct (map fst (?notinP1))
    using wf_part_list_filter_minus[OF 2(2)[THEN conjunct1]]
      2.prems by auto
  ultimately have IH: partition_on (X − P1) (set (map fst (merge_part2_raw f part1 (?notinP1))))
    distinct (map fst (merge_part2_raw f part1 (?notinP1)))
    using 2.hyps[OF refl refl] by auto
  moreover have wf_inP1: partition_on P1 (set (map fst ?inP1)) distinct (map fst ?inP1)
    using wf_part_list_filter_inter[OF 2(2)[THEN conjunct1]]
      2.prems by auto
  moreover have (fst ‘ set ?inP1) ∩ (fst ‘ set (merge_part2_raw f part1 (?notinP1))) = {}
    using IH(1)[THEN partition_onD1]
    by (fastforce simp: map_filter_def split: prod.splits if_splits)
  ultimately show ?case
    using partition_on_append[OF wf_inP1(1) IH(1)] ‹P1 ∪ X = X› wf_inP1(2) IH(2)
    by simp
qed simp

lemma wf_part_list_merge_part3_raw:
  partition_on X (set (map fst part1)) ∧ distinct (map fst part1) ⟹
   partition_on X (set (map fst part2)) ∧ distinct (map fst part2) ⟹
   partition_on X (set (map fst part3)) ∧ distinct (map fst part3) ⟹
   partition_on X (set (map fst (merge_part3_raw f part1 part2 part3)))
    ∧ distinct (map fst (merge_part3_raw f part1 part2 part3))
proof(induct f part1 part2 part3 arbitrary: X rule: merge_part3_raw.induct)
  case (4 f v va vb vc vd ve)
  have partition_on X (set (map fst (v # va))) ∧ distinct (map fst (vb # vc))
```

```
    using 4 by blast
  moreover have partition_on X (set (map fst (vb # vc))) ∧ distinct (map fst (vb # vc))
    using 4 by blast
  ultimately have partition_on X (set (map fst (merge_part2_raw f (v # va) (vb # vc))))
  ∧ distinct (map fst (merge_part2_raw f (v # va) (vb # vc)))
    using wf_part_list_merge_part2_raw[of X (v # va) (vb # vc) f] 4
    by fastforce
  moreover have partition_on X (set (map fst (vd # ve))) ∧ distinct (map fst (vd # ve))
    using 4 by blast
  ultimately show ?case
    using wf_part_list_merge_part2_raw[of X (vd # ve) (merge_part2_raw f (v # va) (vb # vc)) (λpt3
f'. f' pt3)]
    by simp
qed auto

lift_definition merge_part2 :: ('a ⇒ 'a ⇒ 'a) ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒ ('d, 'a) part is
merge_part2_raw
  by (rule wf_part_list_merge_part2_raw)

lift_definition merge_part3 :: ('a ⇒ 'a ⇒ 'a ⇒ 'a) ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒
('d, 'a) part is merge_part3_raw
  by (rule wf_part_list_merge_part3_raw)

definition proof_app :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof (infixl ⊕ 65) where
  p ⊕ q = (case (p, q) of
    (Inl (SHistorically i li sps), Inl q) ⇒ Inl (SHistorically (i+1) li (sps @ [q]))
  | (Inl (SAlways i hi sps), Inl q) ⇒ Inl (SAlways (i−1) hi (q # sps))
  | (Inl (SSince sp2 sp1s), Inl q) ⇒ Inl (SSince sp2 (sp1s @ [q]))
  | (Inl (SUntil sp1s sp2), Inl q) ⇒ Inl (SUntil (q # sp1s) sp2)
  | (Inr (VSince i vp1 vp2s), Inr q) ⇒ Inr (VSince (i+1) vp1 (vp2s @ [q]))
  | (Inr (VOnce i li vps), Inr q) ⇒ Inr (VOnce (i+1) li (vps @ [q]))
  | (Inr (VEventually i hi vps), Inr q) ⇒ Inr (VEventually (i−1) hi (q # vps))
  | (Inr (VSinceInf i li vp2s), Inr q) ⇒ Inr (VSinceInf (i+1) li (vp2s @ [q]))
  | (Inr (VUntil i vp2s vp1), Inr q) ⇒ Inr (VUntil (i−1) (q # vp2s) vp1)
  | (Inr (VUntilInf i hi vp2s), Inr q) ⇒ Inr (VUntilInf (i−1) hi (q # vp2s)))

definition proof_incr :: ('n, 'd) proof ⇒ ('n, 'd) proof where
  proof_incr p = (case p of
    Inl (SOnce i sp) ⇒ Inl (SOnce (i+1) sp)
  | Inl (SEventually i sp) ⇒ Inl (SEventually (i−1) sp)
  | Inl (SHistorically i li sps) ⇒ Inl (SHistorically (i+1) li sps)
  | Inl (SAlways i hi sps) ⇒ Inl (SAlways (i−1) hi sps)
  | Inr (VSince i vp1 vp2s) ⇒ Inr (VSince (i+1) vp1 vp2s)
  | Inr (VOnce i li vps) ⇒ Inr (VOnce (i+1) li vps)
  | Inr (VEventually i hi vps) ⇒ Inr (VEventually (i−1) hi vps)
  | Inr (VHistorically i vp) ⇒ Inr (VHistorically (i+1) vp)
  | Inr (VAlways i vp) ⇒ Inr (VAlways (i−1) vp)
  | Inr (VSinceInf i li vp2s) ⇒ Inr (VSinceInf (i+1) li vp2s)
  | Inr (VUntil i vp2s vp1) ⇒ Inr (VUntil (i−1) vp2s vp1)
  | Inr (VUntilInf i hi vp2s) ⇒ Inr (VUntilInf (i−1) hi vp2s))

definition min_list_wrt :: (('n, 'd) proof ⇒ ('n, 'd) proof ⇒ bool) ⇒ ('n, 'd) proof list ⇒ ('n, 'd) proof
where
  min_list_wrt r xs = hd [x ← xs. ∀ y ∈ set xs. r x y]

definition do_neg :: ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_neg p = (case p of
  Inl sp ⇒ [Inr (VNeg sp)]
```

```
| Inr vp ⇒ [Inl (SNeg vp)])

definition do_or :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_or p1 p2 = (case (p1, p2) of
  (Inl sp1, Inl sp2) ⇒ [Inl (SOrL sp1), Inl (SOrR sp2)]
| (Inl sp1, Inr _  ) ⇒ [Inl (SOrL sp1)]
| (Inr _  , Inl sp2) ⇒ [Inl (SOrR sp2)]
| (Inr vp1, Inr vp2) ⇒ [Inr (VOr vp1 vp2)])

definition do_and :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_and p1 p2 = (case (p1, p2) of
  (Inl sp1, Inl sp2) ⇒ [Inl (SAnd sp1 sp2)]
| (Inl _  , Inr vp2) ⇒ [Inr (VAndR vp2)]
| (Inr vp1, Inl _  ) ⇒ [Inr (VAndL vp1)]
| (Inr vp1, Inr vp2) ⇒ [Inr (VAndL vp1), Inr (VAndR vp2)])

definition do_imp :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_imp p1 p2 = (case (p1, p2) of
  (Inl _  , Inl sp2) ⇒ [Inl (SImpR sp2)]
| (Inl sp1, Inr vp2) ⇒ [Inr (VImp sp1 vp2)]
| (Inr vp1, Inl sp2) ⇒ [Inl (SImpL vp1), Inl (SImpR sp2)]
| (Inr vp1, Inr _  ) ⇒ [Inl (SImpL vp1)])

definition do_iff :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_iff p1 p2 = (case (p1, p2) of
  (Inl sp1, Inl sp2) ⇒ [Inl (SIffSS sp1 sp2)]
| (Inl sp1, Inr vp2) ⇒ [Inr (VIffSV sp1 vp2)]
| (Inr vp1, Inl sp2) ⇒ [Inr (VIffVS vp1 sp2)]
| (Inr vp1, Inr vp2) ⇒ [Inl (SIffVV vp1 vp2)])

definition do_exists :: 'n ⇒ ('n, 'd::{default,linorder}) proof + ('d, ('n, 'd) proof) part ⇒ ('n, 'd) proof
list where
  do_exists x p_part = (case p_part of
  Inl p ⇒ (case p of
    Inl sp ⇒ [Inl (SExists x default sp)]
  | Inr vp ⇒ [Inr (VExists x (trivial_part vp))])
| Inr part ⇒ (if (∃ x∈Vals part. isl x) then
               map (λ(D,p). map_sum (SExists x (Min D)) id p) (filter (λ(_, p). isl p) (subsvals part))
             else
               [Inr (VExists x (map_part projr part))]))

definition do_forall :: 'n ⇒ ('n, 'd::{default,linorder}) proof + ('d, ('n, 'd) proof) part ⇒ ('n, 'd) proof
list where
  do_forall x p_part = (case p_part of
  Inl p ⇒ (case p of
    Inl sp ⇒ [Inl (SForall x (trivial_part sp))]
  | Inr vp ⇒ [Inr (VForall x default vp)])
| Inr part ⇒ (if (∀ x∈Vals part. isl x) then
               [Inl (SForall x (map_part projl part))]
             else
              map (λ(D,p). map_sum id (VForall x (Min D)) p) (filter (λ(_, p). ¬isl p) (subsvals part))))

definition do_prev :: nat ⇒ ℐ ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list where
  do_prev i I t p = (case (p, t < left I) of
  (Inl _ , True) ⇒ [Inr (VPrevOutL i)]
| (Inl sp, False) ⇒ (if mem t I then [Inl (SPrev sp)] else [Inr (VPrevOutR i)])
| (Inr vp, True) ⇒ [Inr (VPrev vp), Inr (VPrevOutL i)]
| (Inr vp, False) ⇒ (if mem t I then [Inr (VPrev vp)] else [Inr (VPrev vp), Inr (VPrevOutR i)]))
```

definition *do_next* :: *nat* ⇒ *ℐ* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_next i I t p = (*case* (*p*, *t* < *left I*) *of*
(*Inl* _ , *True*) ⇒ [*Inr* (*VNextOutL i*)]
| (*Inl sp*, *False*) ⇒ (*if mem t I then* [*Inl* (*SNext sp*)] *else* [*Inr* (*VNextOutR i*)])
| (*Inr vp*, *True*) ⇒ [*Inr* (*VNext vp*), *Inr* (*VNextOutL i*)]
| (*Inr vp*, *False*) ⇒ (*if mem t I then* [*Inr* (*VNext vp*)] *else* [*Inr* (*VNext vp*), *Inr* (*VNextOutR i*)]))

definition *do_once_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_once_base i a p' = (*case* (*p'*, *a* = *0*) *of*
(*Inl sp'*, *True*) ⇒ [*Inl* (*SOnce i sp'*)]
| (*Inr vp'*, *True*) ⇒ [*Inr* (*VOnce i i* [*vp'*])]
| (_ , *False*) ⇒ [*Inr* (*VOnce i i* [])])

definition *do_once* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_once i a p p' = (*case* (*p*, *a* = *0*, *p'*) *of*
(*Inl sp*, *True*, *Inr* _) ⇒ [*Inl* (*SOnce i sp*)]
| (*Inl sp*, *True*, *Inl* (*SOnce* _ *sp'*)) ⇒ [*Inl* (*SOnce i sp'*), *Inl* (*SOnce i sp*)]
| (*Inl* _ , *False*, *Inl* (*SOnce* _ *sp'*)) ⇒ [*Inl* (*SOnce i sp'*)]
| (*Inl* _ , *False*, *Inr* (*VOnce* _ *li vps'*)) ⇒ [*Inr* (*VOnce i li vps'*)]
| (*Inr* _ , *True*, *Inl* (*SOnce* _ *sp'*)) ⇒ [*Inl* (*SOnce i sp'*)]
| (*Inr vp*, *True*, *Inr vp'*) ⇒ [(*Inr vp'*) ⊕ (*Inr vp*)]
| (*Inr* _ , *False*, *Inl* (*SOnce* _ *sp'*)) ⇒ [*Inl* (*SOnce i sp'*)]
| (*Inr* _ , *False*, *Inr* (*VOnce* _ *li vps'*)) ⇒ [*Inr* (*VOnce i li vps'*)])

definition *do_eventually_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_eventually_base i a p' = (*case* (*p'*, *a* = *0*) *of*
(*Inl sp'*, *True*) ⇒ [*Inl* (*SEventually i sp'*)]
| (*Inr vp'*, *True*) ⇒ [*Inr* (*VEventually i i* [*vp'*])]
| (_ , *False*) ⇒ [*Inr* (*VEventually i i* [])])

definition *do_eventually* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_eventually i a p p' = (*case* (*p*, *a* = *0*, *p'*) *of*
(*Inl sp*, *True*, *Inr* _) ⇒ [*Inl* (*SEventually i sp*)]
| (*Inl sp*, *True*, *Inl* (*SEventually* _ *sp'*)) ⇒ [*Inl* (*SEventually i sp'*), *Inl* (*SEventually i sp*)]
| (*Inl* _ , *False*, *Inl* (*SEventually* _ *sp'*)) ⇒ [*Inl* (*SEventually i sp'*)]
| (*Inl* _ , *False*, *Inr* (*VEventually* _ *hi vps'*)) ⇒ [*Inr* (*VEventually i hi vps'*)]
| (*Inr* _ , *True*, *Inl* (*SEventually* _ *sp'*)) ⇒ [*Inl* (*SEventually i sp'*)]
| (*Inr vp*, *True*, *Inr vp'*) ⇒ [(*Inr vp'*) ⊕ (*Inr vp*)]
| (*Inr* _ , *False*, *Inl* (*SEventually* _ *sp'*)) ⇒ [*Inl* (*SEventually i sp'*)]
| (*Inr* _ , *False*, *Inr* (*VEventually* _ *hi vps'*)) ⇒ [*Inr* (*VEventually i hi vps'*)])

definition *do_historically_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_historically_base i a p' = (*case* (*p'*, *a* = *0*) *of*
(*Inl sp'*, *True*) ⇒ [*Inl* (*SHistorically i i* [*sp'*])]
| (*Inr vp'*, *True*) ⇒ [*Inr* (*VHistorically i vp'*)]
| (_ , *False*) ⇒ [*Inl* (*SHistorically i i* [])])

definition *do_historically* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_historically i a p p' = (*case* (*p*, *a* = *0*, *p'*) *of*
(*Inl* _ , *True*, *Inr* (*VHistorically* _ *vp'*)) ⇒ [*Inr* (*VHistorically i vp'*)]
| (*Inl sp*, *True*, *Inl sp'*) ⇒ [(*Inl sp'*) ⊕ (*Inl sp*)]
| (*Inl* _ , *False*, *Inl* (*SHistorically* _ *li sps'*)) ⇒ [*Inl* (*SHistorically i li sps'*)]
| (*Inl* _ , *False*, *Inr* (*VHistorically* _ *vp'*)) ⇒ [*Inr* (*VHistorically i vp'*)]
| (*Inr vp*, *True*, *Inl* _) ⇒ [*Inr* (*VHistorically i vp*)]
| (*Inr vp*, *True*, *Inr* (*VHistorically* _ *vp'*)) ⇒ [*Inr* (*VHistorically i vp*), *Inr* (*VHistorically i vp'*)]
| (*Inr* _ , *False*, *Inl* (*SHistorically* _ *li sps'*)) ⇒ [*Inl* (*SHistorically i li sps'*)]
| (*Inr* _ , *False*, *Inr* (*VHistorically* _ *vp'*)) ⇒ [*Inr* (*VHistorically i vp'*)])

definition *do_always_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_always_base i a p' = (*case* (*p'*, *a* = *0*) *of*
(*Inl sp'*, *True*) ⇒ [*Inl* (*SAlways i i* [*sp'*])]
| (*Inr vp'*, *True*) ⇒ [*Inr* (*VAlways i vp'*)]
| (_ , *False*) ⇒ [*Inl* (*SAlways i i* [])])

definition *do_always* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_always i a p p' = (*case* (*p*, *a* = *0*, *p'*) *of*
(*Inl* _ , *True*, *Inr* (*VAlways* _ *vp'*)) ⇒ [*Inr* (*VAlways i vp'*)]
| (*Inl sp*, *True*, *Inl sp'*) ⇒ [(*Inl sp'*) ⊕ (*Inl sp*)]
| (*Inl* _ , *False*, *Inl* (*SAlways* _ *hi sps'*)) ⇒ [*Inl* (*SAlways i hi sps'*)]
| (*Inl* _ , *False*, *Inr* (*VAlways* _ *vp'*)) ⇒ [*Inr* (*VAlways i vp'*)]
| (*Inr vp*, *True*, *Inl* _) ⇒ [*Inr* (*VAlways i vp*)]
| (*Inr vp*, *True*, *Inr* (*VAlways* _ *vp'*)) ⇒ [*Inr* (*VAlways i vp*), *Inr* (*VAlways i vp'*)]
| (*Inr* _ , *False*, *Inl* (*SAlways* _ *hi sps'*)) ⇒ [*Inl* (*SAlways i hi sps'*)]
| (*Inr* _ , *False*, *Inr* (*VAlways* _ *vp'*)) ⇒ [*Inr* (*VAlways i vp'*)])

definition *do_since_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_since_base i a p1 p2 = (*case* (*p1*, *p2*, *a* = *0*) *of*
(_ , *Inl sp2*, *True*) ⇒ [*Inl* (*SSince sp2* [])]
| (*Inl* _ , _ , *False*) ⇒ [*Inr* (*VSinceInf i i* [])]
| (*Inl* _ , *Inr vp2*, *True*) ⇒ [*Inr* (*VSinceInf i i* [*vp2*])]
| (*Inr vp1*, _ , *False*) ⇒ [*Inr* (*VSince i vp1* []), *Inr* (*VSinceInf i i* [])]
| (*Inr vp1*, *Inr sp2*, *True*) ⇒ [*Inr* (*VSince i vp1* [*sp2*]), *Inr* (*VSinceInf i i* [*sp2*])])

definition *do_since* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_since i a p1 p2 p' = (*case* (*p1*, *p2*, *a* = *0*, *p'*) *of*
(*Inl sp1*, *Inr* _ , *True*, *Inl sp'*) ⇒ [(*Inl sp'*) ⊕ (*Inl sp1*)]
| (*Inl sp1*, _ , *False*, *Inl sp'*) ⇒ [(*Inl sp'*) ⊕ (*Inl sp1*)]
| (*Inl sp1*, *Inl sp2*, *True*, *Inl sp'*) ⇒ [(*Inl sp'*) ⊕ (*Inl sp1*), *Inl* (*SSince sp2* [])]
| (*Inl* _ , *Inr vp2*, *True*, *Inr* (*VSinceInf* _ _ _)) ⇒ [*p'* ⊕ (*Inr vp2*)]
| (*Inl* _ , _ , *False*, *Inr* (*VSinceInf* _ *li vp2s'*)) ⇒ [*Inr* (*VSinceInf i li vp2s'*)]
| (*Inl* _ , *Inr vp2*, *True*, *Inr* (*VSince* _ _ _)) ⇒ [*p'* ⊕ (*Inr vp2*)]
| (*Inl* _ , _ , *False*, *Inr* (*VSince* _ *vp1'* *vp2s'*)) ⇒ [*Inr* (*VSince i vp1'* *vp2s'*)]
| (*Inr vp1*, *Inr vp2*, *True*, *Inl* _) ⇒ [*Inr* (*VSince i vp1* [*vp2*])]
| (*Inr vp1*, _ , *False*, *Inl* _) ⇒ [*Inr* (*VSince i vp1* [])]
| (*Inr* _ , *Inl sp2*, *True*, *Inl* _) ⇒ [*Inl* (*SSince sp2* [])]
| (*Inr vp1*, *Inr vp2*, *True*, *Inr* (*VSinceInf* _ _ _)) ⇒ [*Inr* (*VSince i vp1* [*vp2*]), *p'* ⊕ (*Inr vp2*)]
| (*Inr vp1*, _, *False*, *Inr* (*VSinceInf* _ *li vp2s'*)) ⇒ [*Inr* (*VSince i vp1* []), *Inr* (*VSinceInf i li vp2s'*)]
| (_ , *Inl sp2*, *True*, *Inr* (*VSinceInf* _ _ _)) ⇒ [*Inl* (*SSince sp2* [])]
| (*Inr vp1*, *Inr vp2*, *True*, *Inr* (*VSince* _ _ _)) ⇒ [*Inr* (*VSince i vp1* [*vp2*]), *p'* ⊕ (*Inr vp2*)]
| (*Inr vp1*, _ , *False*, *Inr* (*VSince* _ *vp1'* *vp2s'*)) ⇒ [*Inr* (*VSince i vp1* []), *Inr* (*VSince i vp1'* *vp2s'*)]
| (_ , *Inl vp2*, *True*, *Inr* (*VSince* _ _ _)) ⇒ [*Inl* (*SSince vp2* [])])

definition *do_until_base* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_until_base i a p1 p2 = (*case* (*p1*, *p2*, *a* = *0*) *of*
(_ , *Inl sp2*, *True*) ⇒ [*Inl* (*SUntil* [] *sp2*)]
| (*Inl sp1*, _ , *False*) ⇒ [*Inr* (*VUntilInf i i* [])]
| (*Inl sp1*, *Inr vp2*, *True*) ⇒ [*Inr* (*VUntilInf i i* [*vp2*])]
| (*Inr vp1*, _ , *False*) ⇒ [*Inr* (*VUntil i* [] *vp1*), *Inr* (*VUntilInf i i* [])]
| (*Inr vp1*, *Inr vp2*, *True*) ⇒ [*Inr* (*VUntil i* [*vp2*] *vp1*), *Inr* (*VUntilInf i i* [*vp2*])])

definition *do_until* :: *nat* ⇒ *nat* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof* ⇒ (*'n*, *'d*) *proof list* **where**
do_until i a p1 p2 p' = (*case* (*p1*, *p2*, *a* = *0*, *p'*) *of*
(*Inl sp1*, *Inr* _ , *True*, *Inl* (*SUntil* _ _)) ⇒ [*p'* ⊕ (*Inl sp1*)]

```
| (Inl sp1, _ , False, Inl (SUntil _ _ )) ⇒ [p' ⊕ (Inl sp1)]
| (Inl sp1, Inl sp2, True, Inl (SUntil _ _ )) ⇒ [p' ⊕ (Inl sp1), Inl (SUntil [] sp2)]
| (Inl _ , Inr vp2, True, Inr (VUntilInf _ _ _ )) ⇒ [p' ⊕ (Inr vp2)]
| (Inl _ , _ , False, Inr (VUntilInf _ hi vp2s')) ⇒ [Inr (VUntilInf i hi vp2s')]
| (Inl _ , Inr vp2, True, Inr (VUntil _ _ _ )) ⇒ [p' ⊕ (Inr vp2)]
| (Inl _ , _ , False, Inr (VUntil _ vp2s' vp1')) ⇒ [Inr (VUntil i vp2s' vp1')]
| (Inr vp1, Inr vp2, True, Inl (SUntil _ _ )) ⇒ [Inr (VUntil i [vp2] vp1)]
| (Inr vp1, _ , False, Inl (SUntil _ _ )) ⇒ [Inr (VUntil i [] vp1)]
| (Inr vp1, Inl sp2, True, Inl (SUntil _ _ )) ⇒ [Inl (SUntil [] sp2)]
| (Inr vp1, Inr vp2, True, Inr (VUntilInf _ _ _ )) ⇒ [Inr (VUntil i [vp2] vp1), p' ⊕ (Inr vp2)]
| (Inr vp1, _ , False, Inr (VUntilInf _ hi vp2s')) ⇒ [Inr (VUntil i [] vp1), Inr (VUntilInf i hi vp2s')]
| ( _ , Inl sp2, True, Inr (VUntilInf _ hi vp2s')) ⇒ [Inl (SUntil [] sp2)]
| (Inr vp1, Inr vp2, True, Inr (VUntil _ _ _ )) ⇒ [Inr (VUntil i [vp2] vp1), p' ⊕ (Inr vp2)]
| (Inr vp1, _ , False, Inr (VUntil _ vp2s' vp1')) ⇒ [Inr (VUntil i [] vp1), Inr (VUntil i vp2s' vp1')]
| ( _ , Inl sp2, True, Inr (VUntil _ _ _ )) ⇒ [Inl (SUntil [] sp2)])

fun match :: ('n, 'd) Formula.trm list ⇒ 'd list ⇒ ('n ⇀ 'd) option where
  match [] [] = Some Map.empty
| match (Formula.Const x # ts) (y # ys) = (if x = y then match ts ys else None)
| match (Formula.Var x # ts) (y # ys) = (case match ts ys of
      None ⇒ None
    | Some f ⇒ (case f x of
        None ⇒ Some (f(x ↦ y))
      | Some z ⇒ if y = z then Some f else None))
| match _ _ = None

fun pdt_of :: nat ⇒ 'n ⇒ ('n, 'd :: linorder) Formula.trm list ⇒ 'n list ⇒ ('n ⇀ 'd) list ⇒ ('n, 'd) expl
where
  pdt_of i r ts [] V = (if List.null V then Leaf (Inr (VPred i r ts)) else Leaf (Inl (SPred i r ts)))
| pdt_of i r ts (x # vs) V =
    (let ds = remdups (List.map_filter (λv. v x) V);
         part = tabulate ds (λd. pdt_of i r ts vs (filter (λv. v x = Some d) V)) (pdt_of i r ts vs [])
    in Node x part)

fun apply_pdt1 :: 'n list ⇒ (('n, 'd) proof ⇒ ('n, 'd) proof) ⇒ ('n, 'd) expl ⇒ ('n, 'd) expl where
  apply_pdt1 vs f (Leaf pt) = Leaf (f pt)
| apply_pdt1 (z # vs) f (Node x part) =
  (if x = z then
     Node x (map_part (λexpl. apply_pdt1 vs f expl) part)
   else
     apply_pdt1 vs f (Node x part))
| apply_pdt1 [] _ (Node _ _) = undefined

fun apply_pdt2 :: 'n list ⇒ (('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof) ⇒ ('n, 'd) expl ⇒ ('n, 'd)
expl ⇒ ('n, 'd) expl where
  apply_pdt2 vs f (Leaf pt1) (Leaf pt2) = Leaf (f pt1 pt2)
| apply_pdt2 vs f (Leaf pt1) (Node x part2) = Node x (map_part (apply_pdt1 vs (f pt1)) part2)
| apply_pdt2 vs f (Node x part1) (Leaf pt2) = Node x (map_part (apply_pdt1 vs (λpt1. f pt1 pt2)) part1)
| apply_pdt2 (z # vs) f (Node x part1) (Node y part2) =
    (if x = z ∧ y = z then
       Node z (merge_part2 (apply_pdt2 vs f) part1 part2)
     else if x = z then
       Node x (map_part (λexpl1. apply_pdt2 vs f expl1 (Node y part2)) part1)
     else if y = z then
       Node y (map_part (λexpl2. apply_pdt2 vs f (Node x part1) expl2) part2)
     else
       apply_pdt2 vs f (Node x part1) (Node y part2))
| apply_pdt2 [] _ (Node _ _) (Node _ _) = undefined
```

```
fun apply_pdt3 :: 'n list ⇒ (('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof) ⇒ ('n, 'd) expl ⇒ ('n, 'd) expl ⇒ ('n, 'd) expl ⇒ ('n, 'd) expl where
  apply_pdt3 vs f (Leaf pt1) (Leaf pt2) (Leaf pt3) = Leaf (f pt1 pt2 pt3)
| apply_pdt3 vs f (Leaf pt1) (Leaf pt2) (Node x part3) = Node x (map_part (apply_pdt2 vs (f pt1) (Leaf pt2)) part3)
| apply_pdt3 vs f (Leaf pt1) (Node x part2) (Leaf pt3) = Node x (map_part (apply_pdt2 vs (λpt2. f pt1 pt2) (Leaf pt3)) part2)
| apply_pdt3 vs f (Node x part1) (Leaf pt2) (Leaf pt3) = Node x (map_part (apply_pdt2 vs (λpt1. f pt1 pt2) (Leaf pt3)) part1)
| apply_pdt3 (w # vs) f (Leaf pt1) (Node y part2) (Node z part3) =
  (if y = w ∧ z = w then
     Node w (merge_part2 (apply_pdt2 vs (f pt1)) part2 part3)
   else if y = w then
     Node y (map_part (λexpl2. apply_pdt2 vs (f pt1) expl2 (Node z part3)) part2)
   else if z = w then
     Node z (map_part (λexpl3. apply_pdt2 vs (f pt1) (Node y part2) expl3) part3)
   else
     apply_pdt3 vs f (Leaf pt1) (Node y part2) (Node z part3))
| apply_pdt3 (w # vs) f (Node x part1) (Node y part2) (Leaf pt3) =
  (if x = w ∧ y = w then
     Node w (merge_part2 (apply_pdt2 vs (λpt1 pt2. f pt1 pt2 pt3)) part1 part2)
   else if x = w then
     Node x (map_part (λexpl1. apply_pdt2 vs (λpt1 pt2. f pt1 pt2 pt3) expl1 (Node y part2)) part1)
   else if y = w then
     Node y (map_part (λexpl2. apply_pdt2 vs (λpt1 pt2. f pt1 pt2 pt3) (Node x part1) expl2) part2)
   else
     apply_pdt3 vs f (Node x part1) (Node y part2) (Leaf pt3))
| apply_pdt3 (w # vs) f (Node x part1) (Leaf pt2) (Node z part3) =
  (if x = w ∧ z = w then
     Node w (merge_part2 (apply_pdt2 vs (λpt1. f pt1 pt2)) part1 part3)
   else if x = w then
     Node x (map_part (λexpl1. apply_pdt2 vs (λpt1. f pt1 pt2) expl1 (Node z part3)) part1)
   else if z = w then
     Node z (map_part (λexpl3. apply_pdt2 vs (λpt1. f pt1 pt2) (Node x part1) expl3) part3)
   else
     apply_pdt3 vs f (Node x part1) (Leaf pt2) (Node z part3))
| apply_pdt3 (w # vs) f (Node x part1) (Node y part2) (Node z part3) =
  (if x = w ∧ y = w ∧ z = w then
     Node z (merge_part3 (apply_pdt3 vs f) part1 part2 part3)
   else if x = w ∧ y = w then
     Node w (merge_part2 (apply_pdt3 vs (λpt3 pt1 pt2. f pt1 pt2 pt3) (Node z part3)) part1 part2)
   else if x = w ∧ z = w then
     Node w (merge_part2 (apply_pdt3 vs (λpt2 pt1 pt3. f pt1 pt2 pt3) (Node y part2)) part1 part3)
   else if y = w ∧ z = w then
     Node w (merge_part2 (apply_pdt3 vs (λpt1. f pt1) (Node x part1)) part2 part3)
   else if x = w then
     Node x (map_part (λexpl1. apply_pdt3 vs f expl1 (Node y part2) (Node z part3)) part1)
   else if y = w then
     Node y (map_part (λexpl2. apply_pdt3 vs f (Node x part1) expl2 (Node z part3)) part2)
   else if z = w then
     Node z (map_part (λexpl3. apply_pdt3 vs f (Node x part1) (Node y part2) expl3) part3)
   else
     apply_pdt3 vs f (Node x part1) (Node y part2) (Node z part3))
| apply_pdt3 [] _ _ _ _ = undefined

fun hide_pdt :: 'n list ⇒ (('n, 'd) proof + ('d, ('n, 'd) proof) part ⇒ ('n, 'd) proof) ⇒ ('n, 'd) expl ⇒ ('n, 'd) expl where
```

```
  hide_pdt vs f (Leaf pt) = Leaf (f (Inl pt))
| hide_pdt [x] f (Node y part) = Leaf (f (Inr (map_part unleaf part)))
| hide_pdt (x # xs) f (Node y part) =
  (if x = y then
     Node y (map_part (hide_pdt xs f) part)
   else
     hide_pdt xs f (Node y part))
| hide_pdt [] _ _ = undefined

context
  fixes σ :: ('n, 'd :: {default, linorder}) trace and
  cmp :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ bool
begin

function (sequential) eval :: 'n list ⇒ nat ⇒ ('n, 'd) Formula.formula ⇒ ('n, 'd) expl where
  eval vs i Formula.TT = Leaf (Inl (STT i))
| eval vs i Formula.FF = Leaf (Inr (VFF i))
| eval vs i (Eq_Const x c) = Node x (tabulate [c] (λc. Leaf (Inl (SEq_Const i x c))) (Leaf (Inr
(VEq_Const i x c))))
| eval vs i (Formula.Pred r ts) =
 (pdt_of i r ts (filter (λx. x ∈ Formula.fv (Formula.Pred r ts)) vs) (List.map_filter (match ts) (sorted_list_of_set
(snd ` {rd ∈ Γ σ i. fst rd = r}))))
| eval vs i (Formula.Neg φ) = apply_pdt1 vs (λp. min_list_wrt cmp (do_neg p)) (eval vs i φ)
| eval vs i (Formula.Or φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_or p1 p2)) (eval vs i φ)
(eval vs i ψ)
| eval vs i (Formula.And φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_and p1 p2)) (eval vs i
φ) (eval vs i ψ)
| eval vs i (Formula.Imp φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_imp p1 p2)) (eval vs i
φ) (eval vs i ψ)
| eval vs i (Formula.Iff φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_iff p1 p2)) (eval vs i φ)
(eval vs i ψ)
| eval vs i (Formula.Exists x φ) = hide_pdt (vs @ [x]) (λp. min_list_wrt cmp (do_exists x p)) (eval (vs
@ [x]) i φ)
| eval vs i (Formula.Forall x φ) = hide_pdt (vs @ [x]) (λp. min_list_wrt cmp (do_forall x p)) (eval (vs
@ [x]) i φ)
| eval vs i (Formula.Prev I φ) = (if i = 0 then Leaf (Inr VPrevZ)
                                  else apply_pdt1 vs (λp. min_list_wrt cmp (do_prev i I (Δ σ i) p)) (eval vs
(i−1) φ))
| eval vs i (Formula.Next I φ) = apply_pdt1 vs (λl. min_list_wrt cmp (do_next i I (Δ σ (i+1)) l)) (eval
vs (i+1) φ)
| eval vs i (Formula.Once I φ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inr (VOnceOut i))
   else (let expl = eval vs i φ in
       (if i = 0 then
          apply_pdt1 vs (λp. min_list_wrt cmp (do_once_base 0 0 p)) expl
        else (if right I ≥ enat (Δ σ i) then
                apply_pdt2 vs (λp p'. min_list_wrt cmp (do_once i (left I) p p')) expl
                  (eval vs (i−1) (Formula.Once (subtract (Δ σ i) I) φ))
              else apply_pdt1 vs (λp. min_list_wrt cmp (do_once_base i (left I) p)) expl))))
| eval vs i (Formula.Historically I φ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inl (SHistoricallyOut i))
   else (let expl = eval vs i φ in
       (if i = 0 then
          apply_pdt1 vs (λp. min_list_wrt cmp (do_historically_base 0 0 p)) expl
        else (if right I ≥ enat (Δ σ i) then
                apply_pdt2 vs (λp p'. min_list_wrt cmp (do_historically i (left I) p p')) expl
                  (eval vs (i−1) (Formula.Historically (subtract (Δ σ i) I) φ))
              else apply_pdt1 vs (λp. min_list_wrt cmp (do_historically_base i (left I) p)) expl))))
```

```
| eval vs i (Formula.Eventually I φ) =
  (let expl = eval vs i φ in
  (if right I = ∞ then undefined
   else (if right I ≥ enat (Δ σ (i+1)) then
          apply_pdt2 vs (λp p'. min_list_wrt cmp (do_eventually i (left I) p p')) expl
                        (eval vs (i+1) (Formula.Eventually (subtract (Δ σ (i+1)) I) φ))
         else apply_pdt1 vs (λp. min_list_wrt cmp (do_eventually_base i (left I) p)) expl)))
| eval vs i (Formula.Always I φ) =
  (let expl = eval vs i φ in
  (if right I = ∞ then undefined
   else (if right I ≥ enat (Δ σ (i+1)) then
          apply_pdt2 vs (λp p'. min_list_wrt cmp (do_always i (left I) p p')) expl
                        (eval vs (i+1) (Formula.Always (subtract (Δ σ (i+1)) I) φ))
         else apply_pdt1 vs (λp. min_list_wrt cmp (do_always_base i (left I) p)) expl)))
| eval vs i (Formula.Since φ I ψ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inr (VSinceOut i))
   else (let expl1 = eval vs i φ in
         let expl2 = eval vs i ψ in
         (if i = 0 then
            apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_since_base 0 0 p1 p2)) expl1 expl2
          else (if right I ≥ enat (Δ σ i) then
                  apply_pdt3 vs (λp1 p2 p'. min_list_wrt cmp (do_since i (left I) p1 p2 p')) expl1 expl2
                             (eval vs (i−1) (Formula.Since φ (subtract (Δ σ i) I) ψ))
                else apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_since_base i (left I) p1 p2)) expl1
expl2))))
| eval vs i (Formula.Until φ I ψ) =
  (let expl1 = eval vs i φ in
   let expl2 = eval vs i ψ in
   (if right I = ∞ then undefined
    else (if right I ≥ enat (Δ σ (i+1)) then
           apply_pdt3 vs (λp1 p2 p'. min_list_wrt cmp (do_until i (left I) p1 p2 p')) expl1 expl2
                      (eval vs (i+1) (Formula.Until φ (subtract (Δ σ (i+1)) I) ψ))
          else apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_until_base i (left I) p1 p2)) expl1 expl2)))
  by pat_completeness auto

fun dist where
  dist i (Formula.Once _ _) = i
| dist i (Formula.Historically _ _) = i
| dist i (Formula.Eventually I _) = LTP σ (case right I of ∞ ⇒ 0 | enat b ⇒ (τ σ i + b)) − i
| dist i (Formula.Always I _) = LTP σ (case right I of ∞ ⇒ 0 | enat b ⇒ (τ σ i + b)) − i
| dist i (Formula.Since _ _ _) = i
| dist i (Formula.Until _ I _) = LTP σ (case right I of ∞ ⇒ 0 | enat b ⇒ (τ σ i + b)) − i
| dist _ _ = undefined

lemma i_less_LTP: τ σ (Suc i) ≤ b + τ σ i ⟹ i < LTP σ (b + τ σ i)
  by (metis Suc_le_lessD i_le_LTPi_add le_iff_add)

termination eval
  by (relation measures [λ(_, _, φ). size φ, λ(_, i, φ). dist i φ])
    (auto simp: add.commute le_diff_conv i_less_LTP intro!: diff_less_mono2)

end

end
```

12 Examples

definition *monitor* :: (('*n* :: *linorder* × '*d* :: {*default*, *linorder*} *list*) *set* × *nat*) *list* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *expl list* **where**
  monitor π φ = *map* (λ*i*. *eval* (*trace_of_list* π) (λ*p q*. *size p* ≤ *size q*) (*sorted_list_of_set* (*fv* φ)) *i* φ) [*0* ..< *length* π]
definition *check* :: (('*n* :: *linorder* × '*d* :: {*default*, *linorder*} *list*) *set* × *nat*) *list* ⇒ ('*n*, '*d*) *formula* ⇒ *bool* **where**
  check π φ = *list_all* (*check_all* (*trace_of_list* π) φ) (*monitor* π φ)

12.1 Infinite Domain

definition *prefix* :: ((*string* × *string list*) *set* × *nat*) *list* **where**
  prefix =
    [({(''*mgr_S*'', [''*Mallory*'', ''*Alice*'']),
       (''*mgr_S*'', [''*Merlin*'', ''*Bob*'']),
       (''*mgr_S*'', [''*Merlin*'', ''*Charlie*''])}, *1307532861*::*nat*),
     ({(''*approve*'', [''*Mallory*'', ''*152*''])}, *1307532861*),
     ({(''*approve*'', [''*Merlin*'', ''*163*'']),
       (''*publish*'', [''*Alice*'', ''*160*'']),
       (''*mgr_F*'', [''*Merlin*'', ''*Charlie*''])}, *1307955600*),
     ({(''*approve*'', [''*Merlin*'', ''*187*'']),
       (''*publish*'', [''*Bob*'', ''*163*'']),
       (''*publish*'', [''*Alice*'', ''*163*'']),
       (''*publish*'', [''*Charlie*'', ''*163*'']),
       (''*publish*'', [''*Charlie*'', ''*152*''])}, *1308477599*)]

definition *phi* :: (*string*, *string*) *Formula.formula* **where**
  phi = *Formula.Imp* (*Formula.Pred* ''*publish*'' [*Formula.Var* ''*a*'', *Formula.Var* ''*f*'']) (*Formula.Once* (*init 604800*) (*Formula.Exists* ''*m*'' (*Formula.Since* (*Formula.Neg* (*Formula.Pred* ''*mgr_F*'' [*Formula.Var* ''*m*'', *Formula.Var* ''*a*'']))) *all* (*Formula.And* (*Formula.Pred* ''*mgr_S*'' [*Formula.Var* ''*m*'', *Formula.Var* ''*a*'']) (*Formula.Pred* ''*approve*'' [*Formula.Var* ''*m*'', *Formula.Var* ''*f*'']))))))

value *monitor prefix phi*
lemma *check prefix phi*
  by *eval*

12.2 Finite Domain

datatype *Domain* = *Mallory* | *Merlin* | *Martin* | *Alice* | *Bob* | *Charlie* | *David* | *Default* | *R42* | *R152* | *R160* | *R163* | *R187*

definition *ord* :: *Domain* ⇒ *nat* **where**
  ord d = (*case d of*
     Mallory ⇒ *0*
   | *Merlin* ⇒ *1*
   | *Martin* ⇒ *2*
   | *Alice* ⇒ *3*
   | *Bob* ⇒ *4*
   | *Charlie* ⇒ *5*
   | *David* ⇒ *6*
   | *Default* ⇒ *7*
   | *R42* ⇒ *8*
   | *R152* ⇒ *9*
   | *R160* ⇒ *10*
   | *R163* ⇒ *11*
   | *R187* ⇒ *12*)

```
instantiation Domain :: default begin
definition default_Domain = Default
instance ..
end
instantiation Domain :: universe begin
definition universe_Domain = Some [Mallory, Merlin, Martin, Alice, Bob, Charlie, David, Default,
R42, R152, R160, R163, R187]
instance by standard (auto simp: universe_Domain_def intro: Domain.exhaust)
end
instantiation Domain :: linorder begin
definition less_eq_Domain d d' = (ord d ≤ ord d')
definition less_Domain d d' = (ord d < ord d')
instance by standard (auto simp: less_eq_Domain_def less_Domain_def ord_def split: Domain.splits)
end

definition fprefix :: ((string × Domain list) set × nat) list where
  fprefix =
    [({(''mgr_S'', [Mallory, Alice]),
       (''mgr_S'', [Merlin, Bob]),
       (''mgr_S'', [Merlin, Charlie])}, 1307532861::nat),
     ({(''approve'', [Mallory, R152])}, 1307532861),
     ({(''approve'', [Merlin, R163]),
       (''publish'', [Alice, R160]),
       (''mgr_F'', [Merlin, Charlie])}, 1307955600),
     ({(''approve'', [Merlin, R187]),
       (''publish'', [Bob, R163]),
       (''publish'', [Alice, R163]),
       (''publish'', [Charlie, R163]),
       (''publish'', [Charlie, R152])}, 1308477599)]

definition fphi :: (string, Domain) Formula.formula where
  fphi = Formula.Imp (Formula.Pred ''publish'' [Formula.Var ''a'', Formula.Var ''f''])
    (Formula.Once (init 604800) (Formula.Exists ''m'' (Formula.Since
      (Formula.Neg (Formula.Pred ''mgr_F'' [Formula.Var ''m'', Formula.Var ''a'']))) all
      (Formula.And (Formula.Pred ''mgr_S'' [Formula.Var ''m'', Formula.Var ''a''])
                   (Formula.Pred ''approve'' [Formula.Var ''m'', Formula.Var ''f'']))))))

value monitor fprefix fphi
lemma check fprefix fphi
  by eval
```

References

[1] L. Lima, A. Herasimau, M. Raszyk, D. Traytel, and S. Yuan. Explainable online monitoring of metric temporal logic. In S. Sankaranarayanan and N. Sharygina, editors, *TACAS 2023*, volume 13994 of *LNCS*, pages 473–491. Springer, 2023.

[2] L. Lima, J. J. H. y Munive, and D. Traytel. Explainable online monitoring of metric first-order temporal logic. In B. Finkbeiner and L. Kovács, editors, *TACAS 2024*, volume 14570 of *LNCS*, pages 288–307. Springer, 2024.