

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	2
1.1	Infinite Traces	2
1.2	Finite Trace Prefixes	4
1.3	Earliest and Latest Time-Points	7
2	Metric First-Order Temporal Logic	10
2.1	Syntax	10
2.2	Semantics	11
3	Valued Partitions	16
3.1	<i>size</i> setup	17
3.2	Functions on Valued Partitions	18
4	Partitioned Decision Trees	20
5	Proof System	21
5.1	Soundness and Completeness	22
6	Proof Objects	30
7	Auxiliary Lemmas	32
8	Proof Checker	34
8.1	Checker Soundness	37
8.2	Executable Variant of the Checker	52
8.3	Latest Relevant Time-Point	68
8.4	Active Domain	69
8.5	Congruence Modulo Active Domain	70
8.6	Checker Completeness	80

8.7	Lifting the Checker to PDTs	90
9	Type of Events	93
9.1	Code Adaptation for 8-bit strings	93
9.2	Event Parameters	94
10	Code Generation	96
10.1	Type Class Instances	96
10.2	Progress	98
10.3	Trace	100
10.4	Auxiliary results	102
10.5	<i>v_check_exec</i> setup	103
10.6	ETP/LTP setup	108
10.7	Exported functions	110
11	Unverified Explanation-Producing Monitoring Algorithm	111
12	Examples	122
12.1	Infinite Domain	122
12.2	Finite Domain	122

1 Traces and Trace Prefixes

1.1 Infinite Traces

coinductive *sorted* :: 'a :: linorder stream \Rightarrow bool **where**
shd s \leq *shd* (stl s) \Longrightarrow *sorted* (stl s) \Longrightarrow *sorted* s

lemma *sorted_siterate*[simp]: $(\bigwedge n. n \leq f n) \Longrightarrow$ *sorted* (siterate f n)
by (coinduction arbitrary: n) auto

lemma *sortedD*: *sorted* s \Longrightarrow s !! i \leq stl s !! i
by (induct i arbitrary: s) (auto elim: sorted.cases)

lemma *sorted_sdrop*: *sorted* s \Longrightarrow *sorted* (sdrop i s)
by (coinduction arbitrary: i s) (auto elim: sorted.cases sortedD)

lemma *sorted_monoD*: *sorted* s \Longrightarrow i \leq j \Longrightarrow s !! i \leq s !! j

proof (induct j - i arbitrary: j)
case (Suc x)
from Suc(1)[of j - 1] Suc(2-4) sortedD[of s j - 1]
show ?case **by** (cases j) (auto simp: le_Suc_eq Suc_diff_le)
qed simp

lemma *sorted_stake*: *sorted* s \Longrightarrow *sorted* (stake i s)
by (induct i arbitrary: s)
(auto elim: sorted.cases simp: in_set_conv_nth
intro!: sorted_monoD[of _ 0, simplified, THEN order_trans, OF _ sortedD])

lemma *sorted_monoI*: $\forall i j. i \leq j \longrightarrow$ s !! i \leq s !! j \Longrightarrow *sorted* s
by (coinduction arbitrary: s)
(auto dest: spec2[of _ Suc _ Suc _] spec2[of _ 0 Suc 0])

lemma *sorted_iff_mono*: *sorted* s \longleftrightarrow $(\forall i j. i \leq j \longrightarrow$ s !! i \leq s !! j)
using sorted_monoI sorted_monoD **by** metis

lemma *sorted_iff_le_Suc*: *sorted* s \longleftrightarrow $(\forall i. s !! i \leq s !!$ Suc i)

```

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

definition sincreasing s = ( $\forall x. \exists i. x < s !! i$ )

lemma sincreasingI: ( $\bigwedge x. \exists i. x < s !! i$ )  $\implies$  sincreasing s
  by (simp add: sincreasing_def)

lemma sincreasing_grD:
  fixes x :: 'a :: semilattice_sup
  assumes sincreasing s
  shows  $\exists j > i. x < s !! j$ 
proof -
  let ?A = insert x {s !! n | n. n  $\leq$  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  $\neg i < j$ 
    then have s !! j  $\in$  ?A by (auto simp: not_less)
    then have s !! j  $\leq$  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 ( $\bigwedge n. n < f n$ )
  shows sincreasing (siterate f n)
unfolding sincreasing_def proof
  fix x
  show  $\exists i. x < \textit{siterate} f n !! i$ 
  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  $\implies$  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 = ( $\forall i. \textit{finite} (s !! i)$ )

lemma sfiniteI: ( $\bigwedge i. \textit{finite} (s !! i)$ )  $\implies$  sfinite s
  by (simp add: sfinite_def)

typedef 'a trace = {s :: ('a set  $\times$  nat) stream. sorted (smap snd s)  $\wedge$  sincreasing (smap snd s)  $\wedge$  sfinite

```

$(\text{smap fst } s)\}$
by (*intro exI*[*of_ smap* ($\lambda i. (\{\}, i)$) *nats*])
(auto simp: stream.map_comp stream.map_ident sfinite_def cong: stream.map_cong)

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **is**
 $\lambda s i. \text{fst } (s !! i)$.

lift_definition $\tau :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **is**
 $\lambda s i. \text{snd } (s !! i)$.

lemma *stream_eq_iff*: $s = s' \iff (\forall n. s !! n = s' !! n)$
by (*metis stream.map_cong0 stream_smap_nats*)

lemma *trace_eqI*: $(\bigwedge i. \Gamma \sigma i = \Gamma \sigma' i) \implies (\bigwedge i. \tau \sigma i = \tau \sigma' i) \implies \sigma = \sigma'$
by *transfer (auto simp: stream_eq_iff intro!: prod_eqI)*

lemma τ_mono [*simp*]: $i \leq j \implies \tau s i \leq \tau s j$
by *transfer (auto simp: sorted_iff_mono)*

lemma *ex_le_τ*: $\exists j \geq i. x \leq \tau s j$
by (*transfer fixing: i x*) (*auto dest!: sincreasing_grD*[*of_ i x*] *less_imp_le*)

lemma *le_τ_less*: $\tau \sigma i \leq \tau \sigma j \implies j < i \implies \tau \sigma i = \tau \sigma j$
by (*simp add: antisym*)

lemma *less_τD*: $\tau \sigma i < \tau \sigma j \implies i < j$
by (*meson τ_mono less_le_not_le not_le_imp_less*)

abbreviation $\Delta s i \equiv \tau s i - \tau s (i - 1)$

1.2 Finite Trace Prefixes

typedef $'a \text{ prefix} = \{p :: ('a \text{ set} \times \text{nat}) \text{ list. sorted } (\text{map snd } p)\}$
by (*auto intro!: exI*[*of_ []*])

setup_lifting *type_definition_prefix*

lift_definition *pmap_Γ* :: $('a \text{ set} \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ prefix} \Rightarrow 'b \text{ prefix}$ **is**
 $\lambda f. \text{map } (\lambda(x, i). (f x, i))$
by (*simp add: split_beta comp_def*)

lift_definition *last_ts* :: $'a \text{ prefix} \Rightarrow \text{nat}$ **is**
 $\lambda p. (\text{case } p \text{ of } [] \Rightarrow 0 \mid _ \Rightarrow \text{snd } (\text{last } p))$.

lift_definition *first_ts* :: $\text{nat} \Rightarrow 'a \text{ prefix} \Rightarrow \text{nat}$ **is**
 $\lambda n p. (\text{case } p \text{ of } [] \Rightarrow n \mid _ \Rightarrow \text{snd } (\text{hd } p))$.

lift_definition *pnil* :: $'a \text{ prefix}$ **is** $[]$ **by** *simp*

lift_definition *plen* :: $'a \text{ prefix} \Rightarrow \text{nat}$ **is** *length* .

lift_definition *psnoc* :: $'a \text{ prefix} \Rightarrow 'a \text{ set} \times \text{nat} \Rightarrow 'a \text{ prefix}$ **is**
 $\lambda p x. \text{if } (\text{case } p \text{ of } [] \Rightarrow 0 \mid _ \Rightarrow \text{snd } (\text{last } p)) \leq \text{snd } x \text{ then } p @ [x] \text{ else } []$
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 \Rightarrow 'a prefix \Rightarrow bool **is**
 $\lambda p q. \exists r. q = p @ r .$

definition *less_prefix* :: 'a prefix \Rightarrow 'a prefix \Rightarrow 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]:

$last_ts\ p \leq snd\ x \Longrightarrow last_ts\ q \leq snd\ y \Longrightarrow psnoc\ p\ x = psnoc\ q\ y \longleftrightarrow (p = q \wedge x = y)$
by transfer auto

lift_definition *prefix_of* :: 'a prefix \Rightarrow 'a trace \Rightarrow 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*: $\pi \leq \pi' \Longrightarrow plen\ \pi \leq plen\ \pi'$

by transfer auto

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc* *p* *x*) *s* $\Longrightarrow last_ts\ p \leq snd\ x \Longrightarrow$

$(prefix_of\ p\ s \Longrightarrow \Gamma\ s\ (plen\ p) = fst\ x \Longrightarrow \tau\ s\ (plen\ p) = snd\ x \Longrightarrow P) \Longrightarrow P$

by transfer (simp del: *stake_simps* add: *stake_Suc*)

lemma *le_pnil*[simp]: *pnil* $\leq \pi$

by transfer auto

lift_definition *take_prefix* :: nat \Rightarrow 'a trace \Rightarrow '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]: $last_ts\ \pi \leq snd\ x \Longrightarrow plen\ (psnoc\ \pi\ x) = plen\ \pi + 1$

by transfer auto

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

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

lemma *pdrop_0[simp]*: *pdrop 0 π = π*
by *transfer auto*

lemma *prefix_of_antimono*: $\pi \leq \pi' \implies \text{prefix_of } \pi' s \implies \text{prefix_of } \pi s$
by *transfer (auto simp del: stake_add simp add: stake_add[symmetric])*

lemma *prefix_of_imp_linear*: $\text{prefix_of } \pi \sigma \implies \text{prefix_of } \pi' \sigma \implies \pi \leq \pi' \vee \pi' \leq \pi$

proof *transfer*

fix $\pi \pi'$ **and** σ :: (*'a set × nat*) *stream*

assume *assms*: *stake (length π) σ = π stake (length π') σ = π'*

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 $\pi' = \text{take } (\text{length } \pi) \pi' @ \text{drop } (\text{length } \pi) \pi'$

by *simp*

moreover have $\text{take } (\text{length } \pi) \pi' = \pi$

using *assms le* **by** (*metis min.absorb1 take_stake*)

ultimately show *?thesis* **by** *auto*

next

case *ge*

then have $\pi = \text{take } (\text{length } \pi') \pi @ \text{drop } (\text{length } \pi') \pi$

by *simp*

moreover have $\text{take } (\text{length } \pi') \pi = \pi'$

using *assms ge* **by** (*metis min.absorb1 take_stake*)

ultimately show *?thesis* **by** *auto*

qed

qed

lemma *τ_prefix_conv*: $\text{prefix_of } p s \implies \text{prefix_of } p s' \implies i < \text{plen } p \implies \tau s i = \tau s' i$
by *transfer (simp add: stake_nth[symmetric])*

lemma *Γ_prefix_conv*: $\text{prefix_of } p s \implies \text{prefix_of } p s' \implies i < \text{plen } p \implies \Gamma s i = \Gamma s' i$
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 increasingI*)

fix *x*

obtain *i* **where** $n < i$ **and** $x < s !! i$

using *sincreasing_grD[OF assms]* **by** *blast*

then have $x < \text{sdrop } n s !! (i - n)$

by (*simp add: sdrop_nth*)

then show $\exists i. x < \text{sdrop } n s !! i$ **..**

qed

lemma *sorted_shift*:

sorted (xs @- s) = (sorted xs ∧ sorted s ∧ (∀ x ∈ set xs. ∀ y ∈ sset s. x ≤ y))

proof *safe*

assume ***: *sorted (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 sorted s ∀ x ∈ set xs. ∀ y ∈ sset s. x ≤ y
then show sorted (xs @- s)
proof (coinduction arbitrary: xs s)
  case (ssorted xs s)
  with ‹sorted 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  $i_j$ :  $\tau \sigma i \geq \tau \sigma j$  using  $P$  by auto
from  $j\_def$  have  $\tau \sigma j \geq n$ 
  unfolding  $ETP\_def$  using  $LeastI\_ex\ ex\_le\_tau$  by force
then show  $\tau \sigma i \geq n$  using  $i_j$  by auto
next
assume  $Q$ :  $\tau \sigma i \geq n$ 
then show  $ETP\ \sigma\ n \leq i$  unfolding  $ETP\_def$ 
  by (auto simp add:  $Least\_le$ )
qed

lemma  $tau\_LTP\_k$ :
  assumes  $\tau\ \sigma\ 0 \leq n$   $LTP\ \sigma\ n < k$ 
  shows  $\tau\ \sigma\ k > n$ 
proof -
  have  $finite\ \{i.\ \tau\ \sigma\ i \leq n\}$ 
  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  $n\_asm: n \geq \tau\ \sigma\ 0$ 
  shows  $(i \leq LTP\ \sigma\ n \longleftrightarrow \tau\ \sigma\ i \leq n)$ 
proof
  define  $A$  and  $j$  where  $A\_def: A \equiv \{i.\ \tau\ \sigma\ i \leq n\}$  and  $j\_def: j \equiv LTP\ \sigma\ n$ 
  assume  $P: i \leq LTP\ \sigma\ n$ 
  from  $n\_asm\ A\_def$  have  $A\_ne: A \neq \{\}$  by auto
  from  $j\_def$  have  $i_j: \tau\ \sigma\ i \leq \tau\ \sigma\ j$  using  $P$  by auto
  have  $not\_in: k \notin A$  if  $j < k$  for  $k$ 
    using  $n\_asm\ that\ tau\_LTP\_k\ leD$ 
    unfolding  $A\_def\ j\_def$  by blast
  then have  $A \subseteq \{0..<Suc\ j\}$ 
    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  $\tau\ \sigma\ j \leq n$ 
    using  $Max\_in[of\ A]\ A\_def\ fin\_A$ 
    unfolding  $LTP\_def$ 
    by simp
  then show  $\tau\ \sigma\ i \leq n$  using  $i_j$  by auto
next
  define  $A$  and  $j$  where  $A\_def: A \equiv \{i.\ \tau\ \sigma\ i \leq n\}$  and  $j\_def: j \equiv LTP\ \sigma\ n$ 
  assume  $Q: \tau\ \sigma\ i \leq n$ 
  have  $not\_in: k \notin A$  if  $j < k$  for  $k$ 
    using  $n\_asm\ that\ tau\_LTP\_k\ leD$ 
    unfolding  $A\_def\ j\_def$  by blast
  then have  $A \subseteq \{0..<Suc\ j\}$ 
    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  $i \in A$  using  $Q\ A\_def$  by auto
ultimately show  $i \leq LTP\ \sigma\ n$ 
  using  $Max\_ge[of\ A]\ A\_def$ 
  unfolding  $LTP\_def$ 
  by auto
qed

lemma  $ETP\_delta: i \geq ETP\ \sigma\ (\tau\ \sigma\ l + n) \implies \delta\ \sigma\ i\ l \geq n$ 
proof -
  assume  $P: i \geq ETP\ \sigma\ (\tau\ \sigma\ l + n)$ 
  then have  $\tau\ \sigma\ i \geq \tau\ \sigma\ l + n$  by (auto simp add: i_ETP_tau)
  then show ?thesis by auto
qed

lemma  $ETP\_ge: ETP\ \sigma\ (\tau\ \sigma\ l + n + 1) > l$ 
proof -
  define  $j$  where  $j\_def: j \equiv \tau\ \sigma\ l + n + 1$ 
  then have  $etp\_j: \tau\ \sigma\ (ETP\ \sigma\ j) \geq j$  unfolding  $ETP\_def$ 
    using  $LeastI\_ex\ ex\_le\_tau$  by force
  then have  $\tau\ \sigma\ (ETP\ \sigma\ j) > \tau\ \sigma\ l$  using  $j\_def$  by auto
  then show ?thesis using  $j\_def\ less\_tauD$  by blast
qed

lemma  $i\_le\_LTPi: i \leq LTP\ \sigma\ (\tau\ \sigma\ i)$ 
  using  $\tau\_mono\ i\_LTP\_tau[of\ \sigma\ \tau\ \sigma\ i\ i]$ 
  by auto

lemma  $i\_le\_LTPi\_add: i \leq LTP\ \sigma\ (\tau\ \sigma\ i + n)$ 
  using  $i\_le\_LTPi$ 
  by (simp add: add_increasing2 i_LTP_tau)

lemma  $i\_le\_LTPi\_minus:$ 
  assumes  $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i\ i > 0\ n > 0$ 
  shows  $LTP\ \sigma\ (\tau\ \sigma\ i - n) < i$ 
  unfolding  $LTP\_def$ 
proof (subst Max_less_iff; (intro ballI; elim CollectE)?)
  show  $finite\ \{j. \tau\ \sigma\ j \leq \tau\ \sigma\ i - n\}$ 
    unfolding  $finite\_nat\_set\_iff\_bounded\_le$ 
  proof (intro exI[of _ i], safe)
    fix  $j$ 
    assume  $\tau\ \sigma\ j \leq \tau\ \sigma\ i - n$ 
    with  $assms(1,3)$  show  $j < i$ 
    by (metis add_leD2 add_strict_increasing le_add_diff_inverse less_tauD less_or_eq_imp_le)
  qed
next
  from  $assms(1)$  show  $\{j. \tau\ \sigma\ j \leq \tau\ \sigma\ i - n\} \neq \{\}$ 
    by (auto simp: le_diff_conv2)
next
  fix  $j$ 
  assume  $\tau\ \sigma\ j \leq \tau\ \sigma\ i - n$ 
  with  $assms(1,3)$  show  $j < i$ 
    by (metis add_leD2 add_strict_increasing le_add_diff_inverse less_tauD)
qed

lemma  $i\_ge\_etpi: ETP\ \sigma\ (\tau\ \sigma\ i) \leq i$ 
  using  $i\_ETP\_tau$  by auto

```

2 Metric First-Order Temporal Logic

2.1 Syntax

type_synonym ('n, 'a) event = ('n × 'a list)
type_synonym ('n, 'a) database = ('n, 'a) event set
type_synonym ('n, 'a) prefix = ('n × 'a list) prefix
type_synonym ('n, 'a) trace = ('n × 'a list) trace
type_synonym ('n, 'a) env = 'n ⇒ 'a
type_synonym ('n, 'a) envset = 'n ⇒ 'a set

datatype (fv_trm: 'n, 'a) trm = is_Var: Var 'n (v) | is_Const: Const 'a (c)

lemma in_fv_trm_conv: $x \in \text{fv_trm } t \iff t = \mathbf{v } x$
by (cases t) auto

datatype ('n, 'a) formula =

TT	(⊤)
FF	(⊥)
Eq_Const 'n 'a	(⊆ ≈ ⊆ [85, 85] 85)
Pred 'n ('n, 'a) trm list	(⊆ † ⊆ [85, 85] 85)
Neg ('n, 'a) formula	(¬ _F ⊆ [82] 82)
Or ('n, 'a) formula ('n, 'a) formula	(infixr ∨ _F 80)
And ('n, 'a) formula ('n, 'a) formula	(infixr ∧ _F 80)
Imp ('n, 'a) formula ('n, 'a) formula	(infixr → _F 79)
Iff ('n, 'a) formula ('n, 'a) formula	(infixr ↔ _F 79)
Exists 'n ('n, 'a) formula	(∃ _F ⊆ ⊆ [70, 70] 70)
Forall 'n ('n, 'a) formula	(∀ _F ⊆ ⊆ [70, 70] 70)
Prev \mathcal{I} ('n, 'a) formula	(Y ⊆ ⊆ [1000, 65] 65)
Next \mathcal{I} ('n, 'a) formula	(X ⊆ ⊆ [1000, 65] 65)
Once \mathcal{I} ('n, 'a) formula	(P ⊆ ⊆ [1000, 65] 65)
Historically \mathcal{I} ('n, 'a) formula	(H ⊆ ⊆ [1000, 65] 65)
Eventually \mathcal{I} ('n, 'a) formula	(F ⊆ ⊆ [1000, 65] 65)
Always \mathcal{I} ('n, 'a) formula	(G ⊆ ⊆ [1000, 65] 65)
Since ('n, 'a) formula \mathcal{I} ('n, 'a) formula	(⊆ S ⊆ ⊆ [60, 1000, 60] 60)
Until ('n, 'a) formula \mathcal{I} ('n, 'a) formula	(⊆ U ⊆ ⊆ [60, 1000, 60] 60)

primrec fv :: ('n, 'a) formula ⇒ 'n set **where**

fv (r † ts) = ⋃ (fv_trm ' set ts)
 fv ⊤ = {}
 fv ⊥ = {}
 fv (x ≈ c) = {x}
 fv (¬_F φ) = fv φ
 fv (φ ∨_F ψ) = fv φ ∪ fv ψ
 fv (φ ∧_F ψ) = fv φ ∪ fv ψ
 fv (φ →_F ψ) = fv φ ∪ fv ψ
 fv (φ ↔_F ψ) = fv φ ∪ fv ψ
 fv (∃_Fx. φ) = fv φ - {x}
 fv (∀_Fx. φ) = fv φ - {x}
 fv (**Y** \mathcal{I} φ) = fv φ
 fv (**X** \mathcal{I} φ) = fv φ
 fv (**P** \mathcal{I} φ) = fv φ
 fv (**H** \mathcal{I} φ) = fv φ
 fv (**F** \mathcal{I} φ) = fv φ
 fv (**G** \mathcal{I} φ) = fv φ
 fv (φ **S** \mathcal{I} ψ) = fv φ ∪ fv ψ
 fv (φ **U** \mathcal{I} ψ) = fv φ ∪ fv ψ

primrec consts :: ('n, 'a) formula ⇒ '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)

| $consts \top = \{\}$
| $consts \perp = \{\}$
| $consts (x \approx c) = \{c\}$
| $consts (\neg_F \varphi) = consts \varphi$
| $consts (\varphi \vee_F \psi) = consts \varphi \cup consts \psi$
| $consts (\varphi \wedge_F \psi) = consts \varphi \cup consts \psi$
| $consts (\varphi \longrightarrow_F \psi) = consts \varphi \cup consts \psi$
| $consts (\varphi \longleftarrow_F \psi) = consts \varphi \cup consts \psi$
| $consts (\exists_F x. \varphi) = consts \varphi$
| $consts (\forall_F x. \varphi) = consts \varphi$
| $consts (\mathbf{Y} I \varphi) = consts \varphi$
| $consts (\mathbf{X} I \varphi) = consts \varphi$
| $consts (\mathbf{P} I \varphi) = consts \varphi$
| $consts (\mathbf{H} I \varphi) = consts \varphi$
| $consts (\mathbf{F} I \varphi) = consts \varphi$
| $consts (\mathbf{G} I \varphi) = consts \varphi$
| $consts (\varphi \mathbf{S} I \psi) = consts \varphi \cup consts \psi$
| $consts (\varphi \mathbf{U} I \psi) = consts \varphi \cup consts \psi$

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

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

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

definition *nfv* :: ('n, 'a) formula \Rightarrow nat **where**
nfv \varphi = card (fv \varphi)

fun *future_bounded* :: ('n, 'a) formula \Rightarrow bool **where**
future_bounded \top = True
| *future_bounded \perp = True*
| *future_bounded (_ \dagger _) = True*
| *future_bounded (_ \approx _) = True*
| *future_bounded (\neg_F \varphi) = future_bounded \varphi*
| *future_bounded (\varphi \vee_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)*
| *future_bounded (\varphi \wedge_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)*
| *future_bounded (\varphi \longrightarrow_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)*
| *future_bounded (\varphi \longleftarrow_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)*
| *future_bounded (\exists_F _. \varphi) = future_bounded \varphi*
| *future_bounded (\forall_F _. \varphi) = future_bounded \varphi*
| *future_bounded (\mathbf{Y} I \varphi) = future_bounded \varphi*
| *future_bounded (\mathbf{X} I \varphi) = future_bounded \varphi*
| *future_bounded (\mathbf{P} I \varphi) = future_bounded \varphi*
| *future_bounded (\mathbf{H} I \varphi) = future_bounded \varphi*
| *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 \psi)*
| *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 \Rightarrow ('n, 'a) trm \Rightarrow 'a ($___$) [70,89] 89 **where**
eval_trm v (v x) = v x

| $eval_trm\ v\ (c\ x) = x$

lemma $eval_trm_fv_cong$: $\forall x \in fv_trm\ t.\ v\ x = v'\ x \implies v[[t]] = v'[[t]]$
by ($induction\ t$) $simp_all$

definition $eval_trms$:: $('n, 'a)\ env \Rightarrow ('n, 'a)\ trm\ list \Rightarrow 'a\ list\ (_ \llbracket _ \rrbracket [70,89]\ 89)$ **where**
 $eval_trms\ v\ ts = map\ (eval_trm\ v)\ ts$

lemma $eval_trms_fv_cong$:
 $\forall t \in set\ ts.\ \forall x \in fv_trm\ t.\ v\ x = v'\ x \implies v[[ts]] = v'[[ts]]$
using $eval_trm_fv_cong[of\ _ v v']$
by ($auto\ simp: eval_trms_def$)

primrec $eval_trm_set$:: $('n, 'a)\ envset \Rightarrow ('n, 'a)\ trm \Rightarrow ('n, 'a)\ trm \times 'a\ set\ (_ \llbracket _ \rrbracket [70,89]\ 89)$ **where**
 $eval_trm_set\ vs\ (v\ x) = (v\ x, vs\ x)$
| $eval_trm_set\ vs\ (c\ x) = (c\ x, \{x\})$

definition $eval_trms_set$:: $('n, 'a)\ envset \Rightarrow ('n, 'a)\ trm\ list \Rightarrow (('n, 'a)\ trm \times 'a\ set)\ list\ (_ \llbracket _ \rrbracket [70,89]\ 89)$
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 :: $('n, 'a)\ trace \Rightarrow ('n, 'a)\ env \Rightarrow nat \Rightarrow ('n, 'a)\ formula \Rightarrow bool\ (\langle _ \rangle, _ \rangle \models _ [56, 56, 56, 56]\ 55)$ **where**

$\langle \sigma, v, i \rangle \models \top = True$
| $\langle \sigma, v, i \rangle \models \perp = False$
| $\langle \sigma, v, i \rangle \models r\ \dagger\ ts = ((r, v[[ts]]) \in \Gamma\ \sigma\ i)$
| $\langle \sigma, v, i \rangle \models x \approx c = (v\ x = c)$
| $\langle \sigma, v, i \rangle \models \neg_F\ \varphi = (\neg\ \langle \sigma, v, i \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \varphi \vee_F\ \psi = (\langle \sigma, v, i \rangle \models \varphi \vee \langle \sigma, v, i \rangle \models \psi)$
| $\langle \sigma, v, i \rangle \models \varphi \wedge_F\ \psi = (\langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i \rangle \models \psi)$
| $\langle \sigma, v, i \rangle \models \varphi \longrightarrow_F\ \psi = (\langle \sigma, v, i \rangle \models \varphi \longrightarrow \langle \sigma, v, i \rangle \models \psi)$
| $\langle \sigma, v, i \rangle \models \varphi \longleftarrow_F\ \psi = (\langle \sigma, v, i \rangle \models \varphi \longleftarrow \langle \sigma, v, i \rangle \models \psi)$
| $\langle \sigma, v, i \rangle \models \exists_F x.\ \varphi = (\exists z.\ \langle \sigma, v(x := z), i \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \forall_F x.\ \varphi = (\forall z.\ \langle \sigma, v(x := z), i \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{Y}\ I\ \varphi = (case\ i\ of\ 0 \Rightarrow False\ |\ Suc\ j \Rightarrow mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \langle \sigma, v, j \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{X}\ I\ \varphi = (mem\ (\tau\ \sigma\ (Suc\ i) - \tau\ \sigma\ i)\ I \wedge \langle \sigma, v, Suc\ i \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{P}\ I\ \varphi = (\exists j \leq i.\ mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \langle \sigma, v, j \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{H}\ I\ \varphi = (\forall j \leq i.\ mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \longrightarrow \langle \sigma, v, j \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{F}\ I\ \varphi = (\exists j \geq i.\ mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge \langle \sigma, v, j \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \mathbf{G}\ I\ \varphi = (\forall j \geq i.\ mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \longrightarrow \langle \sigma, v, j \rangle \models \varphi)$
| $\langle \sigma, v, i \rangle \models \varphi\ \mathbf{S}\ I\ \psi = (\exists j \leq i.\ mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{i..i\}.\ \langle \sigma, v, k \rangle \models \varphi))$
| $\langle \sigma, v, i \rangle \models \varphi\ \mathbf{U}\ I\ \psi = (\exists j \geq i.\ mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{i..<j\}.\ \langle \sigma, v, k \rangle \models \varphi))$

lemma sat_fv_cong : $\forall x \in fv\ \varphi.\ v\ x = v'\ x \implies \langle \sigma, v, i \rangle \models \varphi = \langle \sigma, v', i \rangle \models \varphi$

proof ($induct\ \varphi\ arbitrary: v\ v'\ i$)

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  $\varphi$ )
then show ?case unfolding sat.simps
  by (intro iff_exI) (simp add: nth_Cons')
next
case (Forall t  $\varphi$ )
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:  $\langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi \longleftrightarrow$ 
  ( $mem\ 0\ I \wedge \langle \sigma, v, i \rangle \models \psi \vee$ 
   $\Delta\ \sigma\ (i + 1) \leq right\ I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \psi$ )
  (is ?L  $\longleftrightarrow$  ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j:  $i \leq j\ mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I\ \langle \sigma, v, j \rangle \models \psi \ \forall k \in \{i ..< j\}. \langle \sigma, v, k \rangle \models \varphi$ 
    by auto
  then show ?R
  proof (cases i = j)
    case False
    with j(1,2) have  $\Delta\ \sigma\ (i + 1) \leq right\ I$ 
      by (auto elim: order_trans[rotated] simp: diff_le_mono)
    moreover from False j(1,4) have  $\langle \sigma, v, i \rangle \models \varphi$  by auto
    moreover from False j have  $\langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \psi$ 
      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  $\Delta: \Delta\ \sigma\ (i + 1) \leq right\ I$  and now:  $\langle \sigma, v, i \rangle \models \varphi$  and
  next:  $\langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \psi$ 
  from next obtain j where j:  $i + 1 \leq j\ mem\ (\tau\ \sigma\ j - \tau\ \sigma\ (i + 1))\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)$ 
     $\langle \sigma, v, j \rangle \models \psi \ \forall k \in \{i + 1 ..< j\}. \langle \sigma, v, k \rangle \models \varphi$ 
    by auto
  from  $\Delta$  j(1,2) have  $mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$ 
    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:  $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi \longleftrightarrow$ 
  ( $mem\ 0\ I \wedge \langle \sigma, v, i \rangle \models \psi \vee$ 
   $(i > 0 \wedge \Delta\ \sigma\ i \leq right\ I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (subtract\ (\Delta\ \sigma\ i)\ I)\ \psi$ )
  (is ?L  $\longleftrightarrow$  ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j:  $j \leq i\ mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I\ \langle \sigma, v, j \rangle \models \psi \ \forall k \in \{j <.. i\}. \langle \sigma, v, k \rangle \models \varphi$ 
    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  $\Delta\ \sigma\ i \leq right\ I$ 
      by (auto elim: order_trans[rotated] simp: diff_le_mono2 le_Suc_eq)
    moreover from False j(1,4) have  $\langle \sigma, v, i \rangle \models \varphi$  by auto
    moreover from False j have  $\langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (subtract\ (\Delta\ \sigma\ i)\ I)\ \psi$ 
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
    ultimately show ?thesis by auto
  qed simp
qed simp

```

next

assume $i: 0 < i$ and $\Delta: \Delta \sigma i \leq \text{right } I$ and now: $\langle \sigma, v, i \rangle \models \varphi$ and
prev: $\langle \sigma, v, i - 1 \rangle \models \varphi$ **S** ($\text{subtract } (\Delta \sigma i) I$) ψ
from prev **obtain** j where $j: j \leq i - 1$ mem $(\tau \sigma (i - 1) - \tau \sigma j)$ ($(\text{subtract } (\Delta \sigma i) I)$)
 $\langle \sigma, v, j \rangle \models \psi \forall k \in \{j <.. i - 1\}. \langle \sigma, v, k \rangle \models \varphi$
by *auto*
from $\Delta i j(1,2)$ **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*: $\langle \sigma, v, 0 \rangle \models \varphi$ **S** $I \psi \longleftrightarrow \text{mem } 0 I \wedge \langle \sigma, v, 0 \rangle \models \psi$
by *auto*

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

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

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

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

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

lemma *sat_Eventually_rec*: $\langle \sigma, v, i \rangle \models \mathbf{F} I \varphi \longleftrightarrow$
 $\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$
 $(\Delta \sigma (i + 1) \leq \text{right } I \wedge \langle \sigma, v, i + 1 \rangle \models \mathbf{F} (\text{subtract } (\Delta \sigma (i + 1)) I) \varphi)$
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*: $\langle \sigma, v, i \rangle \models \mathbf{G} I \varphi \longleftrightarrow$
 $(\text{mem } 0 I \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$
 $(\Delta \sigma (i + 1) \leq \text{right } I \longrightarrow \langle \sigma, v, i + 1 \rangle \models \mathbf{G} (\text{subtract } (\Delta \sigma (i + 1)) I) \varphi)$
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* (\top)
and *FF* (\perp)
and *Pred* ($_ \dagger _$ [85, 85] 85)
and *Eq_Const* ($_ \approx _$ [85, 85] 85)
and *Neg* ($\neg_F _$ [82] 82)
and *And* (**infixr** \wedge_F 80)
and *Or* (**infixr** \vee_F 80)
and *Imp* (**infixr** \longrightarrow_F 79)
and *Iff* (**infixr** \longleftrightarrow_F 79)
and *Exists* ($\exists_{F_} _$ [70,70] 70)
and *Forall* ($\forall_{F_} _$ [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* ($_ \mathbf{S} _ _$ [60,1000,60] 60)
and *Until* ($_ \mathbf{U} _ _$ [60,1000,60] 60)

no_notation *eval_trm* ($_ \llbracket _ \rrbracket$ [70,89] 89)
and *eval_trms* ($_ \llbracket _ \rrbracket$ [70,89] 89)
and *eval_trm_set* ($_ \{_ \}$ [70,89] 89)
and *eval_trms_set* ($_ \{_ \}$ [70,89] 89)
and *sat* ($\langle _ , _ , _ \rangle \models _$ [56, 56, 56, 56] 55)
and *Interval.interval* ($\llbracket _ , _ \rrbracket$)

end

bundle *MFOTL_notation* **begin**

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

notation *TT* (\top)
and *FF* (\perp)
and *Pred* ($_ \dagger _$ [85, 85] 85)
and *Eq_Const* ($_ \approx _$ [85, 85] 85)
and *Neg* ($\neg_F _$ [82] 82)
and *And* (**infixr** \wedge_F 80)
and *Or* (**infixr** \vee_F 80)
and *Imp* (**infixr** \longrightarrow_F 79)
and *Iff* (**infixr** \longleftrightarrow_F 79)
and *Exists* ($\exists_{F_} _$ [70,70] 70)
and *Forall* ($\forall_{F_} _$ [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* ($_ \mathbf{S} _ _$ [60,1000,60] 60)
and *Until* ($_ \mathbf{U} _ _$ [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
   $\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j \longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\}$ 
   $\{\} \notin \text{fst } ' \text{set } xs$ 
shows disjoint (fst ' set xs)  $\wedge$  distinct (map fst xs)
proof -
from assms(1) have disjoint (fst ' set xs)
  by (metis disjoint_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)
   $i < \text{length } xs$ 
   $j < \text{length } xs$ 
   $i \neq j$ 
shows  $\text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\}$ 
proof -
from assms have  $\text{fst } (xs ! i) \in \text{fst } ' \text{set } xs$ 
  and  $\text{fst } (xs ! j) \in \text{fst } ' \text{set } xs$ 
  by auto
moreover have  $\text{fst } (xs ! i) \neq \text{fst } (xs ! j)$ 
  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:

```

 $(\bigcup X \in \text{fst } ' \text{set } xs. X) = \text{UNIV}$ 
 $\wedge (\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j$ 
   $\longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\}) \wedge \{\} \notin \text{fst } ' \text{set } xs$ 
 $\longleftrightarrow \text{partition\_on UNIV } (\text{set } (\text{map } \text{fst } xs)) \wedge \text{distinct } (\text{map } \text{fst } xs)$ 
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 λf 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 o map_part h = size_part f (g o 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*]: $(\bigwedge x. x \in \text{Vals } xs \implies f x \leq g x) \implies \text{size_part } h f xs \leq \text{size_part } h g xs$
by *transfer* (*force simp: image_iff intro!: size_list_pointwise*)

3.2 Functions on Valued Partitions

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

lemma *Vals_transfer*[*transfer_rule*]: $\text{rel_fun } (\text{pcr_part } (=) (=)) (=) (\text{set } \circ \text{map } \text{snd}) \text{ Vals}$
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*]: $\text{set } (\text{vals } xs) = \text{Vals } xs$
by *transfer simp*

lift_definition *part_hd* :: $('d, 'a) \text{ part} \Rightarrow 'a \text{ is } \text{snd} \circ \text{hd}$.

lift_definition *tabulate* :: $'d \text{ list} \Rightarrow ('d \Rightarrow 'n) \Rightarrow 'n \Rightarrow ('d, 'n) \text{ part is}$
 $\lambda ds f z. \text{if } \text{distinct } ds \text{ then if } \text{set } ds = \text{UNIV} \text{ then } \text{map } (\lambda d. (\{d\}, f d)) ds \text{ else } (- \text{set } ds, z) \# \text{map } (\lambda d. (\{d\}, f d)) ds \text{ else } [(UNIV, z)]$
by (*auto simp: o_def distinct_map inj_on_def partition_on_def disjoint_def*)

lift_definition *lookup_part* :: $('d, 'a) \text{ part} \Rightarrow 'd \Rightarrow 'a \text{ is } \lambda xs d. \text{snd } (\text{the } (\text{find } (\lambda(D, _). d \in D) xs))$.

lemma *Vals_tabulate*[*simp*]: $\text{Vals } (\text{tabulate } xs f z) =$
 $(\text{if } \text{distinct } xs \text{ then if } \text{set } xs = \text{UNIV} \text{ then } f \text{ ' set } xs \text{ else } \{z\} \cup f \text{ ' set } xs \text{ else } \{z\})$
by *transfer* (*auto simp: image_iff*)

lemma *lookup_part_tabulate*[*simp*]: $\text{lookup_part } (\text{tabulate } xs f z) x =$
 $(\text{if } \text{distinct } xs \wedge x \in \text{set } xs \text{ then } f x \text{ else } z)$
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*]: $\text{part_hd } \text{part} \in \text{Vals } \text{part}$
by *transfer* (*auto simp: partition_on_def image_iff intro!: hd_in_set*)

lemma *lookup_part_Vals*[*simp*]: $\text{lookup_part } \text{part } d \in \text{Vals } \text{part}$

proof (*transfer, goal_cases part*)

case (*part xs d*)

then have *unique*: $(\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j \longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\})$

using *part_list_eq*[*of xs*]

by *simp*

from part obtain *D x where* *D*: $(D, x) \in \text{set } xs \text{ } d \in D$

unfolding *partition_on_def*

by *fastforce*

with unique have *find* $(\lambda(D, _). d \in D) xs = \text{Some } (D, x)$

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_SubVals*: $\exists D. d \in D \wedge (D, \text{lookup_part } \text{part } d) \in \text{SubVals } \text{part}$

proof (*transfer, goal_cases part*)

case (*part d xs*)

then have *unique*: $(\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j \longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\})$

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 (subvals 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 subvals_part_estimation[termination_simp]: (D, e) ∈ set (subvals 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!: $exI[\mathbf{where} P=\lambda x. \exists y z. P x y z \wedge Q x y z \mathbf{for} P Q, OF exI, OF exI, OF conjI]$

lemma *fst_lookup*: $d \in fst$ (lookup part d)
proof (transfer fixing: d, goal_cases)
 case (1 part)
 then obtain y where find $(\lambda(D, _). d \in D)$ 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 \in set (subsvals part)
proof (transfer fixing: d, goal_cases)
 case (1 part)
 then obtain y where find $(\lambda(D, _). d \in D)$ 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 \Rightarrow ('d, 'pt) part is $\lambda pt. [(UNIV, pt)]$
 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 \Rightarrow ('d, 'l, 'n) pdt \Rightarrow bool **where**
 vars_order vs (Leaf _)
 | $\forall expl \in Vals part1. vars_order\ vs\ expl \Longrightarrow vars_order\ (x \# vs)\ (Node\ x\ part1)$
 | $vars_order\ vs\ (Node\ x\ part1) \Longrightarrow x \neq z \Longrightarrow vars_order\ (z \# vs)\ (Node\ x\ part1)$

lemma *vars_order_Node*:
 assumes distinct xs
 shows $vars_order\ xs\ (Node\ x\ part) = (\exists ys\ zs. xs = ys @ x \# zs \wedge (\forall e \in Vals\ part. vars_order\ zs\ e))$
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
 | $distinct_paths\ (Node\ x\ part) = (\forall e \in Vals\ part. x \notin vars\ e \wedge distinct_paths\ e)$

```

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:  $\forall x \in \text{vars } e. v x = v' x \implies \text{eval\_pdt } v e = \text{eval\_pdt } v' e$ 
by (induct e) auto

lemma vars_order_vars: vars_order vs e  $\implies \text{vars } e \subseteq \text{set } vs$ 
by (induct vs e rule: vars_order.induct) auto

lemma vars_order_distinct_paths: vars_order vs e  $\implies \text{distinct } vs \implies \text{distinct\_paths } e$ 
by (induct vs e rule: vars_order.induct) (auto dest!: vars_order_vars)

lemma eval_pdt_fun_upd: vars_order vs e  $\implies x \notin \text{set } vs \implies \text{eval\_pdt } (v(x := d)) e = \text{eval\_pdt } v e$ 
by (induct vs e rule: vars_order.induct) auto

```

5 Proof System

unbundle MFOTL_notation

context begin

inductive SAT and VIO :: ('n, 'd) trace \Rightarrow ('n, 'd) env \Rightarrow nat \Rightarrow ('n, 'd) formula \Rightarrow bool **for** σ **where**

```

  STT: SAT  $\sigma v i$  TT
| VFF: VIO  $\sigma v i$  FF
| SPred: (r, eval_trms v ts)  $\in \Gamma \sigma i \implies \text{SAT } \sigma v i$  (Pred r ts)
| VPred: (r, eval_trms v ts)  $\notin \Gamma \sigma i \implies \text{VIO } \sigma v i$  (Pred r ts)
| SEq_Const: v x = c  $\implies \text{SAT } \sigma v i$  (Eq_Const x c)
| VEq_Const: v x  $\neq$  c  $\implies \text{VIO } \sigma v i$  (Eq_Const x c)
| SNeg: VIO  $\sigma v i \varphi \implies \text{SAT } \sigma v i$  (Neg  $\varphi$ )
| VNeg: SAT  $\sigma v i \varphi \implies \text{VIO } \sigma v i$  (Neg  $\varphi$ )
| SOrL: SAT  $\sigma v i \varphi \implies \text{SAT } \sigma v i$  (Or  $\varphi \psi$ )
| SOrR: SAT  $\sigma v i \psi \implies \text{SAT } \sigma v i$  (Or  $\varphi \psi$ )
| VOr: VIO  $\sigma v i \varphi \implies \text{VIO } \sigma v i \psi \implies \text{VIO } \sigma v i$  (Or  $\varphi \psi$ )
| SAnd: SAT  $\sigma v i \varphi \implies \text{SAT } \sigma v i \psi \implies \text{SAT } \sigma v i$  (And  $\varphi \psi$ )
| VAndL: VIO  $\sigma v i \varphi \implies \text{VIO } \sigma v i$  (And  $\varphi \psi$ )
| VAndR: VIO  $\sigma v i \psi \implies \text{VIO } \sigma v i$  (And  $\varphi \psi$ )
| SImpL: VIO  $\sigma v i \varphi \implies \text{SAT } \sigma v i$  (Imp  $\varphi \psi$ )
| SImpR: SAT  $\sigma v i \psi \implies \text{SAT } \sigma v i$  (Imp  $\varphi \psi$ )
| VImp: SAT  $\sigma v i \varphi \implies \text{VIO } \sigma v i \psi \implies \text{VIO } \sigma v i$  (Imp  $\varphi \psi$ )
| SIffSS: SAT  $\sigma v i \varphi \implies \text{SAT } \sigma v i \psi \implies \text{SAT } \sigma v i$  (Iff  $\varphi \psi$ )
| SIffVV: VIO  $\sigma v i \varphi \implies \text{VIO } \sigma v i \psi \implies \text{SAT } \sigma v i$  (Iff  $\varphi \psi$ )
| VIffSV: SAT  $\sigma v i \varphi \implies \text{VIO } \sigma v i \psi \implies \text{VIO } \sigma v i$  (Iff  $\varphi \psi$ )
| VIffVS: VIO  $\sigma v i \varphi \implies \text{SAT } \sigma v i \psi \implies \text{VIO } \sigma v i$  (Iff  $\varphi \psi$ )
| SExists:  $\exists z. \text{SAT } \sigma (v(x := z)) i \varphi \implies \text{SAT } \sigma v i$  (Exists x  $\varphi$ )
| VExists:  $\forall z. \text{VIO } \sigma (v(x := z)) i \varphi \implies \text{VIO } \sigma v i$  (Exists x  $\varphi$ )
| SForall:  $\forall z. \text{SAT } \sigma (v(x := z)) i \varphi \implies \text{SAT } \sigma v i$  (Forall x  $\varphi$ )
| VForall:  $\exists z. \text{VIO } \sigma (v(x := z)) i \varphi \implies \text{VIO } \sigma v i$  (Forall x  $\varphi$ )
| SPrev:  $i > 0 \implies \text{mem } (\Delta \sigma i) I \implies \text{SAT } \sigma v (i-1) \varphi \implies \text{SAT } \sigma v i$  (Y I  $\varphi$ )
| VPrev:  $i > 0 \implies \text{VIO } \sigma v (i-1) \varphi \implies \text{VIO } \sigma v i$  (Y I  $\varphi$ )
| VPrevZ:  $i = 0 \implies \text{VIO } \sigma v i$  (Y I  $\varphi$ )
| VPrevOutL:  $i > 0 \implies (\Delta \sigma i) < (\text{left } I) \implies \text{VIO } \sigma v i$  (Y I  $\varphi$ )
| VPrevOutR:  $i > 0 \implies \text{enat } (\Delta \sigma i) > (\text{right } I) \implies \text{VIO } \sigma v i$  (Y I  $\varphi$ )
| SNext:  $\text{mem } (\Delta \sigma (i+1)) I \implies \text{SAT } \sigma v (i+1) \varphi \implies \text{SAT } \sigma v i$  (X I  $\varphi$ )
| VNext: VIO  $\sigma v (i+1) \varphi \implies \text{VIO } \sigma v i$  (X I  $\varphi$ )
| VNextOutL:  $(\Delta \sigma (i+1)) < (\text{left } I) \implies \text{VIO } \sigma v i$  (X I  $\varphi$ )

```

$| VNextOutR: \text{enat } (\Delta \sigma (i+1)) > (\text{right } I) \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $| SOnce: j \leq i \implies \text{mem } (\delta \sigma i j) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{P} I \varphi)$
 $| VOnceOut: \tau \sigma i < \tau \sigma 0 + \text{left } I \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $| VOnce: j = (\text{case right } I \text{ of } \infty \Rightarrow 0$
 $\quad | \text{enat } b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + \text{left } I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \varphi) \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $| SEventually: j \geq i \implies \text{mem } (\delta \sigma j i) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{F} I \varphi)$
 $| VEventually: (\bigwedge k. k \in (\text{case right } I \text{ of } \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | \text{enat } b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies VIO \sigma v k \varphi) \implies$
 $\quad VIO \sigma v i (\mathbf{F} I \varphi)$
 $| SHistorically: j = (\text{case right } I \text{ of } \infty \Rightarrow 0$
 $\quad | \text{enat } b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + \text{left } I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies SAT \sigma v k \varphi) \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $| SHistoricallyOut: \tau \sigma i < \tau \sigma 0 + \text{left } I \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $| VHistorically: j \leq i \implies \text{mem } (\delta \sigma i j) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{H} I \varphi)$
 $| SAlways: (\bigwedge k. k \in (\text{case right } I \text{ of } \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | \text{enat } b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies SAT \sigma v k \varphi) \implies$
 $\quad SAT \sigma v i (\mathbf{G} I \varphi)$
 $| VAlways: j \geq i \implies \text{mem } (\delta \sigma j i) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{G} I \varphi)$
 $| SSince: j \leq i \implies \text{mem } (\delta \sigma i j) I \implies SAT \sigma v j \psi \implies (\bigwedge k. k \in \{j <.. i\} \implies$
 $\quad SAT \sigma v k \varphi) \implies SAT \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSinceOut: \tau \sigma i < \tau \sigma 0 + \text{left } I \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSince: (\text{case right } I \text{ of } \infty \Rightarrow \text{True}$
 $\quad | \text{enat } b \Rightarrow ETP \sigma ((\tau \sigma i) - b) \leq j) \implies$
 $\quad j \leq i \implies (\tau \sigma 0) + \text{left } I \leq (\tau \sigma i) \implies VIO \sigma v j \varphi \implies$
 $\quad (\bigwedge k. k \in \{j .. (LTP_p \sigma i I)\} \implies VIO \sigma v k \psi) \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSinceInf: j = (\text{case right } I \text{ of } \infty \Rightarrow 0$
 $\quad | \text{enat } b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + \text{left } I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \psi) \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| SUntil: j \geq i \implies \text{mem } (\delta \sigma j i) I \implies SAT \sigma v j \psi \implies (\bigwedge k. k \in \{i .. j\} \implies SAT \sigma v k \varphi) \implies$
 $\quad SAT \sigma v i (\varphi \mathbf{U} I \psi)$
 $| VUntil: (\text{case right } I \text{ of } \infty \Rightarrow \text{True}$
 $\quad | \text{enat } b \Rightarrow j < LTP_f \sigma i b) \implies$
 $\quad j \geq i \implies VIO \sigma v j \varphi \implies (\bigwedge k. k \in \{ETP_f \sigma i I .. j\} \implies VIO \sigma v k \psi) \implies$
 $\quad VIO \sigma v i (\varphi \mathbf{U} I \psi)$
 $| VUntilInf: (\bigwedge k. k \in (\text{case right } I \text{ of } \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | \text{enat } b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies VIO \sigma v k \psi) \implies$
 $\quad VIO \sigma v i (\varphi \mathbf{U} I \psi)$

5.1 Soundness and Completeness

lemma *not_sat_SinceD*:

assumes *unsat*: $\neg \langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi$ **and**

witness: $\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \psi$

shows $\exists j \leq i. ETP \sigma (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \tau \sigma i - n) \leq j \wedge \neg \langle \sigma, v, j \rangle \models \varphi$
 $\wedge (\forall k \in \{j .. (\text{min } i (LTP \sigma (\tau \sigma i - \text{left } I)))\}. \neg \langle \sigma, v, k \rangle \models \psi)$

proof –

define *A* **and** *j* **where** *A_def*: $A \equiv \{j. j \leq i \wedge \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \psi\}$

and *j_def*: $j \equiv \text{Max } A$

from *witness* **have** *j*: $j \leq i \wedge \langle \sigma, v, j \rangle \models \psi \text{ mem } (\tau \sigma i - \tau \sigma j) I$

using *Max_in*[of *A*] **unfolding** *j_def*[*symmetric*] **unfolding** *A_def*

by *auto*

moreover

from *j*(β) **have** *ETP* $\sigma (\text{case right } I \text{ of } \text{enat } n \Rightarrow \tau \sigma i - n \mid \infty \Rightarrow 0) \leq j$

unfolding *ETP_def* **by** (*intro* *Least_le*) (*auto* *split*: *enat.splits*)

```

moreover
{ fix  $j$ 
  assume  $j: \tau \sigma j \leq \tau \sigma i$ 
  then obtain  $k$  where  $k: \tau \sigma i < \tau \sigma k$ 
    by (meson ex_le_τ gt_ex less_le_trans)
  have  $j \leq ETP \sigma (Suc (\tau \sigma i))$ 
    unfolding ETP_def
  proof (intro LeastI2[of _ k λi. j ≤ i])
    fix  $l$ 
    assume  $Suc (\tau \sigma i) \leq \tau \sigma l$ 
    with  $j$  show  $j \leq l$ 
      by (metis lessI less_τD nless_le order_less_le_trans)
    qed (auto simp: Suc_le_eq k(1))
} note  $*$  = this
{ fix  $k$ 
  assume  $k \in \{j <.. (min i (LTP \sigma (\tau \sigma i - left I)))\}$ 
  with  $j(\mathcal{B})$  have  $k: j < k k \leq i k \leq Max \{j. left I + \tau \sigma j \leq \tau \sigma i\}$ 
    by (auto simp: LTP_def le_diff_conv2 add commute)
  with  $j(\mathcal{B})$  obtain  $l$  where  $left I + \tau \sigma l \leq \tau \sigma i k \leq l$ 
    by (subst (asm) Max_ge_iff) (auto simp: le_diff_conv2 *)
    intro!: finite_subset[of _ {0 .. ETP \sigma (\tau \sigma i + 1)}])
  then have  $mem (\tau \sigma i - \tau \sigma k) I$ 
    using  $k(1,2) j(\mathcal{B})$ 
    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  $\neg \langle \sigma, v, k \rangle \models \psi$ 
    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:  $\neg \langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi$ 
and witness:  $\exists j \geq i. mem (\delta \sigma j i) I \wedge \langle \sigma, v, j \rangle \models \psi$ 
shows  $\exists j \geq i. (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ n \Rightarrow j < LTP \sigma (\tau \sigma i + n))$ 
 $\wedge \neg (\langle \sigma, v, j \rangle \models \varphi) \wedge (\forall k \in \{(max\ i\ (ETP \sigma (\tau \sigma i + left\ I))) .. j\}.$ 
 $\neg \langle \sigma, v, k \rangle \models \psi)$ 
proof -
from  $\tau\_mono$  have  $i0: \tau \sigma 0 \leq \tau \sigma i$  by auto
from witness obtain  $jmax$  where  $jmax: jmax \geq i \langle \sigma, v, jmax \rangle \models \psi$ 
 $mem (\delta \sigma jmax i) I$  by blast
define  $A$  and  $j$  where  $A\_def: A \equiv \{j. j \geq i \wedge j \leq jmax$ 
 $\wedge mem (\delta \sigma j i) I \wedge \langle \sigma, v, j \rangle \models \psi\}$  and  $j\_def: j \equiv Min A$ 
have  $j: j \geq i \langle \sigma, v, j \rangle \models \psi mem (\delta \sigma j i) I$ 
using  $A\_def j\_def jmax Min\_in$ [of A]
unfolding  $j\_def$ [symmetric] unfolding  $A\_def$ 
by fastforce+
moreover have  $case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ n \Rightarrow j \leq LTP \sigma (\tau \sigma i + n)$ 
using  $i0 j(1,\mathcal{B})$ 
by (auto simp: i_LTP_tau trans_le_add1 split: enat.splits)
moreover
{ fix  $k$ 
  assume  $k\_def: k \in \{(max\ i\ (ETP \sigma (\tau \sigma i + left\ I))) ..< j\}$ 
  then have  $ki: \tau \sigma k \geq \tau \sigma i + left\ I$  using  $i\_ETP\_tau$  by auto
  with  $k\_def$  have  $kj: k < j$  by auto
  then have  $\tau \sigma k \leq \tau \sigma j$  by auto
}

```

```

then have  $\delta \sigma k i \leq \delta \sigma j i$  by auto
with this  $j(3)$  have  $\text{enat } (\delta \sigma k i) \leq \text{right } I$ 
  by (meson enat_ord_simps(1) order_subst2)
with this  $ki j(3)$  have  $\text{mem\_}k: \text{mem } (\delta \sigma k i) I$ 
  unfolding ETP_def by (auto simp: Least_le)

with  $j\_def$  have  $j \leq jmax$  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  $kjm: k \leq jmax$  by auto

with this  $\text{mem\_}k ki$  Min_le[of A k]  $k\_def$  have  $k \notin A$ 
  unfolding  $j\_def[symmetric]$  unfolding  $A\_def$  unfolding ETP_def
  using finite_nat_set_iff_bounded_le kj leD by blast
with this  $\text{mem\_}k k\_def kjm$  have  $\neg \langle \sigma, v, k \rangle \models \psi$ 
  by (simp add: A_def) }
ultimately show ?thesis using unsat
  by (auto split: enat.splits dest!: spec[of _ j])
qed

lemma soundness_raw:  $(SAT \sigma v i \varphi \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge (VIO \sigma v i \varphi \longrightarrow \neg \langle \sigma, v, i \rangle \models \varphi)$ 
proof (induct v i \varphi rule: SAT_VIO.induct)
  case (VOnceOut i I v \varphi)
  { fix  $j$ 
    from  $\tau\_mono$  have  $j0: \tau \sigma 0 \leq \tau \sigma j$  by auto
    then have  $\tau \sigma i < \tau \sigma j + \text{left } I$  using VOnceOut by linarith
    then have  $\delta \sigma i j < \text{left } I$ 
      using VOnceOut less_\tauD verit_comp_simplify1(3) by fastforce
    then have  $\neg \text{mem } (\delta \sigma i j) I$  by auto }
  then show ?case
    by auto
  next
  case (VOnce j I i v \varphi)
  { fix  $k$ 
    assume  $k\_def: \langle \sigma, v, k \rangle \models \varphi \wedge \text{mem } (\delta \sigma i k) I \wedge k \leq i$ 
    then have  $k\_tau: \tau \sigma k \leq \tau \sigma i - \text{left } I$ 
      using diff_le_mono2 by fastforce
    then have  $k\_ltp: k \leq LTP \sigma (\tau \sigma i - \text{left } I)$ 
      using VOnce i_LTP_tau add_le_imp_le_diff
      by blast
    then have  $k \notin \{j .. LTP\_p \sigma i I\}$ 
      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 = \infty)
    (auto 0 0 simp: i_ETP_tau i_LTP_tau le_diff_conv2)
  next
  case (VEventually I i v \varphi)
  { fix  $k n$ 
    assume  $r: \text{right } I = \text{enat } n$ 
    from this have  $tin0: \tau \sigma i + n \geq \tau \sigma 0$ 
      by (auto simp add: trans_le_add1)
    define  $j$  where  $j = LTP \sigma ((\tau \sigma i) + n)$ 
    then have  $j\_i: i \leq j$ 
      by (auto simp add: i_LTP_tau trans_le_add1 j_def)
  }

```



```

assume  $k\_def: \langle \sigma, v, k \rangle \models \varphi \wedge mem (\delta \sigma k i) I \wedge i \leq k$ 
then have  $\tau \sigma k \geq \tau \sigma i + left I$ 
  using  $le\_diff\_conv2$  by  $auto$ 
then have  $k\_etp: k \geq ETP \sigma (\tau \sigma i + left I)$ 
  using  $i\_ETP\_tau$  by  $blast$ 
from  $this k\_def$  VEventually have  $k \notin \{ETP\_f \sigma i I .. j\}$ 
  by  $(auto simp: r j\_def)$ 
then have  $j < k$  using  $r k\_def k\_etp$  by  $auto$ 
from  $k\_def r$  have  $\delta \sigma k i \leq n$  by  $auto$ 
then have  $\tau \sigma k \leq \tau \sigma i + n$  by  $auto$ 
then have  $k \leq j$  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  $(simp add: i\_ETP\_tau enat)$ 
       $(metis \tau\_mono le\_add\_diff\_inverse nat\_add\_left\_cancel\_le)$ 
    next
      case  $infinity$ 
      show  $?thesis$ 
        using  $VEventually$ 
        by  $(auto simp: infinity i\_ETP\_tau le\_diff\_conv2)$ 
      qed
    next
      case  $(SHistorically j I i v \varphi)$ 
      { fix  $k$ 
        assume  $k\_def: \neg \langle \sigma, v, k \rangle \models \varphi \wedge mem (\delta \sigma i k) I \wedge k \leq i$ 
        then have  $k\_tau: \tau \sigma k \leq \tau \sigma i - left I$ 
          using  $diff\_le\_mono2$  by  $fastforce$ 
        then have  $k\_ltp: k \leq LTP \sigma (\tau \sigma i - left I)$ 
          using  $SHistorically i\_LTP\_tau add\_le\_imp\_le\_diff$ 
          by  $blast$ 
        then have  $k \notin \{j .. LTP\_p \sigma i I\}$ 
          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  $(cases right I = \infty)$ 
           $(auto 0 0 simp add: le\_diff\_conv2 i\_ETP\_tau i\_LTP\_tau)$ 
      }
    next
      case  $(SHistoricallyOut i I v \varphi)$ 
      { fix  $j$ 
        from  $\tau\_mono$  have  $j0: \tau \sigma 0 \leq \tau \sigma j$  by  $auto$ 
        then have  $\tau \sigma i < \tau \sigma j + left I$  using  $SHistoricallyOut$  by  $linarith$ 
        then have  $\delta \sigma i j < left I$ 
          using  $SHistoricallyOut less\_tauD not\_le$  by  $fastforce$ 
        then have  $\neg mem (\delta \sigma i j) I$  by  $auto \}$ 
        then show  $?case$  by  $auto$ 
      }
    next
      case  $(SAlways I i v \varphi)$ 
      { fix  $k n$ 
        assume  $r: right I = enat n$ 
        from  $this SAlways$  have  $tin0: \tau \sigma i + n \geq \tau \sigma 0$ 
          by  $(auto simp add: trans\_le\_add1)$ 
        define  $j$  where  $j = LTP \sigma ((\tau \sigma i) + n)$ 
      }

```

```

from SAlways have j_i:  $i \leq j$ 
  by (auto simp add: i_LTP_tau trans_le_add1 j_def)
assume k_def:  $\neg \langle \sigma, v, k \rangle \models \varphi \wedge \text{mem } (\delta \sigma k i) I \wedge i \leq k$ 
then have  $\tau \sigma k \geq \tau \sigma i + \text{left } I$ 
  using le_diff_conv2 by auto
then have k_etp:  $k \geq \text{ETP } \sigma (\tau \sigma i + \text{left } I)$ 
  using SAlways i_ETP_tau by blast
from this k_def SAlways have  $k \notin \{\text{ETP}_f \sigma i I .. j\}$ 
  by (auto simp: r_j_def)
then have  $j < k$  using SAlways k_def k_etp by simp
from k_def r have  $\delta \sigma k i \leq n$  by simp
then have  $\tau \sigma k \leq \tau \sigma i + n$  by simp
then have  $k \leq j$ 
  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  $\varphi$   $\psi$ )
  { fix j
    from  $\tau$ _mono have j0:  $\tau \sigma 0 \leq \tau \sigma j$  by auto
    then have  $\tau \sigma i < \tau \sigma j + \text{left } I$  using VSinceOut by linarith
    then have  $\delta \sigma i j < \text{left } I$  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  $\neg \text{mem } (\delta \sigma i j) I$  by auto }
  then show ?case using VSinceOut by auto
next
case (VSince I i j v  $\varphi$   $\psi$ )
  { fix k
    assume k_def:  $\langle \sigma, v, k \rangle \models \psi \wedge \text{mem } (\delta \sigma i k) I \wedge k \leq i$ 
    then have  $\tau \sigma k \leq \tau \sigma i - \text{left } I$  using diff_le_mono2 by fastforce
    then have k_ltp:  $k \leq \text{LTP } \sigma (\tau \sigma i - \text{left } I)$ 
    using VSince i_LTP_tau add_le_imp_le_diff
    by blast
    then have  $k < j$  using k_def VSince( $\gamma$ )[of k]
    by force
    then have  $j \in \{k <.. i\} \wedge \neg \langle \sigma, v, j \rangle \models \varphi$  using VSince
    by auto }
  then show ?case using VSince
  by force
next
case (VSinceInf j I i v  $\varphi$ )
  { fix k
    assume k_def:  $\langle \sigma, v, k \rangle \models \psi \wedge \text{mem } (\delta \sigma i k) I \wedge k \leq i$ 
    then have k_tau:  $\tau \sigma k \leq \tau \sigma i - \text{left } I$ 
    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: ( $\langle \sigma, v, i \rangle \models \varphi \longrightarrow SAT \sigma v i \varphi$ )  $\wedge$  ( $\neg \langle \sigma, v, i \rangle \models \varphi \longrightarrow VIO \sigma v i \varphi$ )
proof (induct  $\varphi$  arbitrary: v i)
  case (Prev I  $\varphi$ )
  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  $\varphi$ )
  { assume  $\langle \sigma, v, i \rangle \models \mathbf{P} I \varphi$ 
    with Once have  $SAT \sigma v i (\mathbf{P} I \varphi)$ 
    by (auto intro: SAT_VIO.SOnce) }
  moreover
  { assume i_l:  $\tau \sigma i < \tau \sigma 0 + left I$ 
    with Once have  $VIO \sigma v i (\mathbf{P} I \varphi)$ 
    by (auto intro: SAT_VIO.VOnceOut) }
  moreover
  { assume unsat:  $\neg \langle \sigma, v, i \rangle \models \mathbf{P} I \varphi$ 
    and i_ge:  $\tau \sigma 0 + left I \leq \tau \sigma i$ 
    with Once have  $VIO \sigma v i (\mathbf{P} I \varphi)$ 
    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  $\varphi$ )
  from  $\tau\_mono$  have i0:  $\tau \sigma 0 \leq \tau \sigma i$  by auto
  { assume sat:  $\langle \sigma, v, i \rangle \models \mathbf{H} I \varphi$ 
    and i_ge:  $\tau \sigma i \geq \tau \sigma 0 + left I$ 
    with Historically have  $SAT \sigma v i (\mathbf{H} I \varphi)$ 
    using le_diff_conv
    by (auto intro!: SAT_VIO.SHistorically simp: i_LTP_tau i_ETP_tau
split: enat.splits) }
  moreover
  { assume  $\neg \langle \sigma, v, i \rangle \models \mathbf{H} I \varphi$ 
    with Historically have  $VIO \sigma v i (\mathbf{H} I \varphi)$ 
    by (auto intro: SAT_VIO.VHistorically) }
  moreover
  { assume i_l:  $\tau \sigma i < \tau \sigma 0 + left I$ 
    with Historically have  $SAT \sigma v i (\mathbf{H} I \varphi)$ 
    by (auto intro: SAT_VIO.SHistoricallyOut) }
  ultimately show ?case
  by force
next
  case (Eventually I  $\varphi$ )
  from  $\tau\_mono$  have i0:  $\tau \sigma 0 \leq \tau \sigma i$  by auto
  { assume  $\langle \sigma, v, i \rangle \models \mathbf{F} I \varphi$ 
    with Eventually have  $SAT \sigma v i (\mathbf{F} I \varphi)$ 
    by (auto intro: SAT_VIO.SEventually) }
  moreover
  { assume unsat:  $\neg \langle \sigma, v, i \rangle \models \mathbf{F} I \varphi$ 

```

```

with Eventually have VIO  $\sigma v i$  (F  $I \varphi$ )
  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  $I \varphi$ )
  from  $\tau\_mono$  have  $i0: \tau \sigma 0 \leq \tau \sigma i$  by auto
  { assume  $\neg \langle \sigma, v, i \rangle \models \mathbf{G} I \varphi$ 
    with Always have VIO  $\sigma v i$  (G  $I \varphi$ )
      by (auto intro: SAT_VIO.VAlways) }
  moreover
  { assume sat:  $\langle \sigma, v, i \rangle \models \mathbf{G} I \varphi$ 
    with Always have SAT  $\sigma v i$  (G  $I \varphi$ )
      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  $\varphi I \psi$ )
  { assume  $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi$ 
    with Since have SAT  $\sigma v i$  ( $\varphi \mathbf{S} I \psi$ )
      by (auto intro: SAT_VIO.SSince) }
  moreover
  { assume  $i\_l: \tau \sigma i < \tau \sigma 0 + left I$ 
    with Since have VIO  $\sigma v i$  ( $\varphi \mathbf{S} I \psi$ )
      by (auto intro: SAT_VIO.VSinceOut) }
  moreover
  { assume unsat:  $\neg \langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi$ 
    and nw:  $\forall j \leq i. \neg mem (\delta \sigma i j) I \vee \neg \langle \sigma, v, j \rangle \models \psi$ 
    and  $i\_ge: \tau \sigma 0 + left I \leq \tau \sigma i$ 
    with Since have VIO  $\sigma v i$  ( $\varphi \mathbf{S} I \psi$ )
      by (auto intro!: SAT_VIO.VSinceInf simp:  $i\_LTP\_tau i\_ETP\_tau$ 
          split: enat.splits)}
  moreover
  { assume unsat:  $\neg \langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi$ 
    and  $jw: \exists j \leq i. mem (\delta \sigma i j) I \wedge \langle \sigma, v, j \rangle \models \psi$ 
    and  $i\_ge: \tau \sigma 0 + left I \leq \tau \sigma i$ 
    from unsat  $jw$  not_sat_SinceD[of  $\sigma v i \varphi I \psi$ ]
    obtain  $j$  where  $j: j \leq i$ 
      case right  $I$  of  $\infty \Rightarrow True \mid enat n \Rightarrow ETP \sigma (\tau \sigma i - n) \leq j$ 
       $\neg \langle \sigma, v, j \rangle \models \varphi (\forall k \in \{j .. (\min i (LTP \sigma (\tau \sigma i - left I)))\})$ .
       $\neg \langle \sigma, v, k \rangle \models \psi$  by (auto split: enat.splits)
    with Since have VIO  $\sigma v i$  ( $\varphi \mathbf{S} I \psi$ )
      using  $i\_ge$  unsat  $jw$ 
      by (auto intro!: SAT_VIO.VSince) }
  ultimately show ?case
    by (force simp del: sat.simps)
next
case (Until  $\varphi I \psi$ )
  from  $\tau\_mono$  have  $i0: \tau \sigma 0 \leq \tau \sigma i$  by auto
  { assume  $\langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi$ 
    with Until have SAT  $\sigma v i$  ( $\varphi \mathbf{U} I \psi$ )
      by (auto intro: SAT_VIO.SUntil) }
  moreover
  { assume unsat:  $\neg \langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi$ 
    and witness:  $\exists j \geq i. mem (\delta \sigma j i) I \wedge \langle \sigma, v, j \rangle \models \psi$ 
    from this Until not_sat_UntilD[of  $\sigma v i \varphi I \psi$ ] obtain  $j$ 
      where  $j: j \geq i$  (case right  $I$  of  $\infty \Rightarrow True \mid enat n$ 
         $\Rightarrow j < LTP \sigma (\tau \sigma i + n) \wedge \neg (\langle \sigma, v, j \rangle \models \varphi)$ )
  }

```

```

    (∀ 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  $\Rightarrow$  nat and
  v_at :: ('n, 'd) vproof  $\Rightarrow$  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 []  $\Rightarrow$  s_at sp2 | _  $\Rightarrow$  s_at (last sp1s))
| s_at (SUntil sp1s sp2) = (case sp1s of []  $\Rightarrow$  s_at sp2 | sp1 # _  $\Rightarrow$  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 $p_at :: ('n, 'd) \text{proof} \Rightarrow \text{nat}$ **where** $p_at\ p = \text{case_sum}\ s_at\ v_at\ p$

7 Auxiliary Lemmas

lemma $\text{Cons_eq_upt_conv}: x \# xs = [m \dots n] \longleftrightarrow m < n \wedge x = m \wedge xs = [\text{Suc } m \dots n]$
by (induct n arbitrary: xs) (force simp: Cons_eq_append_conv)+

lemma $\text{map_setE}[\text{elim_format}]: \text{map } f\ xs = ys \Longrightarrow y \in \text{set } ys \Longrightarrow \exists x \in \text{set } xs. f\ x = y$
by (induct xs arbitrary: ys) auto

lemma $\text{set_Cons_eq}: \text{set_Cons } X\ XS = (\bigcup xs \in XS. (\lambda x. x \# xs) \text{ ` } X)$
by (auto simp: set_Cons_def)

lemma $\text{set_Cons_empty_iff}: \text{set_Cons } X\ XS = \{\} \longleftrightarrow (X = \{\} \vee XS = \{\})$
by (auto simp: set_Cons_eq)

lemma $\text{infinite_set_ConsI}: XS \neq \{\} \Longrightarrow \text{infinite } X \Longrightarrow \text{infinite } (\text{set_Cons } X\ XS)$
 $X \neq \{\} \Longrightarrow \text{infinite } XS \Longrightarrow \text{infinite } (\text{set_Cons } X\ XS)$

proof(unfold set_Cons_eq)

assume $\text{infinite } X$ **and** $XS \neq \{\}$

then obtain xs **where** $xs \in XS$

by blast

hence $\text{inj } (\lambda x. x \# xs)$

by (clarsimp simp: inj_on_def)

hence $\text{infinite } ((\lambda x. x \# xs) \text{ ` } X)$

using $\langle \text{infinite } X \rangle$ $\text{finite_imageD inj_on_def}$

by blast

moreover have $((\lambda x. x \# xs) \text{ ` } X) \subseteq (\bigcup xs \in XS. (\lambda x. x \# xs) \text{ ` } X)$

using $\langle xs \in XS \rangle$ **by** auto

ultimately show $\text{infinite } (\bigcup xs \in XS. (\lambda x. x \# xs) \text{ ` } X)$

by (simp add: infinite_super)

next

assume $\text{infinite } XS$ **and** $X \neq \{\}$

then show $\text{infinite } (\bigcup xs \in XS. (\lambda x. x \# xs) \text{ ` } X)$

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

primrec fst_pos :: 'a list \Rightarrow 'a \Rightarrow 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 \Rightarrow None | Some n \Rightarrow Some (Suc n)))

lemma fst_pos_None_iff: fst_pos xs x = None \longleftrightarrow x \notin set xs
by (induct xs arbitrary: x; force split: option.splits)

lemma nth_fst_pos: x \in set xs \Longrightarrow 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 \Rightarrow 'a \Rightarrow 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
 \Longrightarrow positions xs x = positions ys y \longleftrightarrow (\forall n < length xs. xs ! n = x \longleftrightarrow 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 = [] \longleftrightarrow x \notin set xs
by (induct xs) simp_all

lemma positions_nth: n \in set (positions xs x) \Longrightarrow 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 \wedge n < length xs}
by (induct xs arbitrary: x)
(use less_Suc_eq_0_disj in \langle auto simp: positions_eq_nil_iff[symmetric] image_iff split: if_splits \rangle)

lemma positions_length: n \in set (positions xs x) \Longrightarrow n < length xs
by (induct xs arbitrary: n x)
(auto simp: positions_eq_nil_iff[symmetric] split: if_splits)

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

lemma fst_pos_in_positions: x \in set xs \Longrightarrow the (fst_pos xs x) \in 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 \in set xs \Longrightarrow 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 \Longrightarrow xs \neq [] \Longrightarrow Min (set xs) = hd xs
by (induct xs)
(auto simp: Min_insert2)

lemma Min_positions: x \in set xs \Longrightarrow 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*: $set (positions\ tXs\ tX) \subseteq set (positions (map\ fst\ tXs) (fst\ tX))$
by (*induct tXs arbitrary: tX*)
(auto simp: subset_eq)

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

lemma *Max_eqI*: $finite\ A \implies A \neq \{\} \implies (\bigwedge a. a \in A \implies a \leq b) \implies \exists a \in A. b \leq a \implies Max\ A = b$
by (*rule antisym[OF Max.boundedI Max_ge_iff[THEN iffD2]]; clarsimp*)

lemma *Max_Suc*: $X \neq \{\} \implies finite\ X \implies Max (Suc\ 'X) = Suc (Max\ X)$
using *Max_ge Max_in*
by (*intro Max_eqI*) *blast+*

lemma *Max_insert0*: $X \neq \{\} \implies finite\ X \implies Max (insert\ (0::nat)\ X) = Max\ X$
using *Max_ge Max_in*
by (*intro Max_eqI*) *blast+*

lemma *positions_Cons_notin_tail*: $x \notin set\ xs \implies positions (x \# xs) x = [0::nat]$
by (*cases xs*) (*auto simp: positions_eq_nil_iff*)

lemma *Max_set_positions_Cons_hd*:
 $x \notin set\ xs \implies Max (set (positions (x \# xs) x)) = 0$
by (*subst positions_Cons_notin_tail*) *simp_all*

lemma *Max_set_positions_Cons_tl*:
 $y \in set\ xs \implies Max (set (positions (x \# xs) y)) = Suc (Max (set (positions xs y)))$
by (*auto simp: Max_Suc positions_eq_nil_iff*)

lemma *max_aux*: $finite\ X \implies Suc\ j \in X \implies Max (insert (Suc\ j) (X - \{j\})) = Max (insert\ j\ X)$
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*: $(\forall x \in A. \forall y \in B. P\ x\ y) = (\forall y \in B. \forall x \in A. P\ x\ y)$
by *auto*

lemma *ball_triv_nonempty*: $A \neq \{\} \implies (\forall x \in A. P) = P$
by *auto*

8 Proof Checker

unbundle *MFOTL_notation*

context *fixes* $\sigma :: ('n, 'd :: \{default, linorder\})\ trace$

begin

fun *s_check* :: $('n, 'd)\ env \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ sproof \Rightarrow bool$
and *v_check* :: $('n, 'd)\ env \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ vproof \Rightarrow bool$ **where**
s_check v f p = (case (f, p) of
*(\top , *STT* *i*) $\Rightarrow True$*
*| (*r* \dagger *ts*, *SPred* *i* *s* *ts'*) \Rightarrow*
($r = s \wedge ts = ts' \wedge (r, v[[ts]]) \in \Gamma\ \sigma\ i$)
*| ($x \approx c$, *SEq_Const* *i* *x'* *c'*) \Rightarrow*

$(c = c' \wedge x = x' \wedge v x = c)$
 $| (\neg_F \varphi, SNeg\ vp) \Rightarrow v_check\ v\ \varphi\ vp$
 $| (\varphi \vee_F \psi, SOrL\ sp1) \Rightarrow s_check\ v\ \varphi\ sp1$
 $| (\varphi \vee_F \psi, SOrR\ sp2) \Rightarrow s_check\ v\ \psi\ sp2$
 $| (\varphi \wedge_F \psi, SAnd\ sp1\ sp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge s_check\ v\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \rightarrow_F \psi, SImpL\ vp1) \Rightarrow v_check\ v\ \varphi\ vp1$
 $| (\varphi \rightarrow_F \psi, SImpR\ sp2) \Rightarrow s_check\ v\ \psi\ sp2$
 $| (\varphi \leftrightarrow_F \psi, SIffSS\ sp1\ sp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge s_check\ v\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \leftrightarrow_F \psi, SIffVV\ vp1\ vp2) \Rightarrow v_check\ v\ \varphi\ vp1 \wedge v_check\ v\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\exists_F x. \varphi, SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s_check\ (v\ (x := val))\ \varphi\ sp)$
 $| (\forall_F x. \varphi, SForall\ y\ sp_part) \Rightarrow (let\ i = s_at\ (part_hd\ sp_part)$
 $\quad in\ x = y \wedge (\forall (sub, sp) \in SubsVals\ sp_part. s_at\ sp = i \wedge (\forall z \in sub. s_check\ (v\ (x := z))\ \varphi\ sp)))$
 $| (\mathbf{Y}\ I\ \varphi, SPrev\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp; i = s_at\ (SPrev\ sp)\ in$
 $\quad i = j+1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{X}\ I\ \varphi, SNext\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp; i = s_at\ (SNext\ sp)\ in$
 $\quad j = i+1 \wedge mem\ (\Delta\ \sigma\ j)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{P}\ I\ \varphi, SOnce\ i\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{F}\ I\ \varphi, SEventually\ i\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp\ in$
 $\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{H}\ I\ \varphi, SHistoricallyOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{H}\ I\ \varphi, SHistorically\ i\ li\ sps) \Rightarrow$
 $\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - b))$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ s_at\ sps = [li\ ..<\ (LTP_p\ \sigma\ i\ I) + 1]$
 $\quad \wedge (\forall sp \in set\ sps. s_check\ v\ \varphi\ sp))$
 $| (\mathbf{G}\ I\ \varphi, SAlways\ i\ hi\ sps) \Rightarrow$
 $\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b)$
 $\quad \wedge right\ I \neq \infty$
 $\quad \wedge map\ s_at\ sps = [(ETP_f\ \sigma\ i\ I) ..<\ hi + 1]$
 $\quad \wedge (\forall sp \in set\ sps. s_check\ v\ \varphi\ sp))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, SSince\ sp2\ sp1s) \Rightarrow$
 $\quad (let\ i = s_at\ (SSince\ sp2\ sp1s); j = s_at\ sp2\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
 $\quad \wedge map\ s_at\ sp1s = [j+1\ ..<\ i+1]$
 $\quad \wedge s_check\ v\ \psi\ sp2$
 $\quad \wedge (\forall sp1 \in set\ sp1s. s_check\ v\ \varphi\ sp1))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, SUntil\ sp1s\ sp2) \Rightarrow$
 $\quad (let\ i = s_at\ (SUntil\ sp1s\ sp2); j = s_at\ sp2\ in$
 $\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$
 $\quad \wedge map\ s_at\ sp1s = [i\ ..<\ j] \wedge s_check\ v\ \psi\ sp2$
 $\quad \wedge (\forall sp1 \in set\ sp1s. s_check\ v\ \varphi\ sp1))$
 $| (_ , _) \Rightarrow False$
 $| v_check\ v\ fp = (case\ (f, p)\ of$
 $\quad (\perp, VFF\ i) \Rightarrow True$
 $\quad (r \dagger ts, VPred\ i\ pred\ ts') \Rightarrow$
 $\quad (r = pred \wedge ts = ts' \wedge (r, v\llbracket ts \rrbracket) \notin \Gamma\ \sigma\ i)$
 $\quad (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow$
 $\quad (c = c' \wedge x = x' \wedge v\ x \neq c)$
 $\quad (\neg_F \varphi, VNeg\ sp) \Rightarrow s_check\ v\ \varphi\ sp$
 $\quad (\varphi \vee_F \psi, VOr\ vp1\ vp2) \Rightarrow v_check\ v\ \varphi\ vp1 \wedge v_check\ v\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $\quad (\varphi \wedge_F \psi, VAndL\ vp1) \Rightarrow v_check\ v\ \varphi\ vp1$
 $\quad (\varphi \wedge_F \psi, VAndR\ vp2) \Rightarrow v_check\ v\ \psi\ vp2$

$(\varphi \longrightarrow_F \psi, VImp\ sp1\ vp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge v_check\ v\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $(\varphi \longleftarrow_F \psi, VIffSV\ sp1\ vp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge v_check\ v\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $(\varphi \longleftrightarrow_F \psi, VIffVS\ vp1\ sp2) \Rightarrow v_check\ v\ \varphi\ vp1 \wedge s_check\ v\ \psi\ sp2 \wedge v_at\ vp1 = s_at\ sp2$
 $(\exists_F x. \varphi, VExists\ y\ vp_part) \Rightarrow (let\ i = v_at\ (part_hd\ vp_part)$
 $\quad in\ x = y \wedge (\forall (sub, vp) \in SubsVals\ vp_part. v_at\ vp = i \wedge (\forall z \in sub. v_check\ (v\ (x := z))\ \varphi\ vp)))$
 $(\forall_F x. \varphi, VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v_check\ (v\ (x := val))\ \varphi\ vp)$
 $(\mathbf{Y}\ I\ \varphi, VPrev\ vp) \Rightarrow$
 $\quad (let\ j = v_at\ vp; i = v_at\ (VPrev\ vp)\ in$
 $\quad i = j+1 \wedge v_check\ v\ \varphi\ vp)$
 $(\mathbf{Y}\ I\ \varphi, VPrevZ) \Rightarrow True$
 $(\mathbf{Y}\ I\ \varphi, VPrevOutL\ i) \Rightarrow$
 $\quad i > 0 \wedge \Delta\ \sigma\ i < left\ I$
 $(\mathbf{Y}\ I\ \varphi, VPrevOutR\ i) \Rightarrow$
 $\quad i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I$
 $(\mathbf{X}\ I\ \varphi, VNext\ vp) \Rightarrow$
 $\quad (let\ j = v_at\ vp; i = v_at\ (VNext\ vp)\ in$
 $\quad j = i+1 \wedge v_check\ v\ \varphi\ vp)$
 $(\mathbf{X}\ I\ \varphi, VNextOutL\ i) \Rightarrow$
 $\quad \Delta\ \sigma\ (i+1) < left\ I$
 $(\mathbf{X}\ I\ \varphi, VNextOutR\ i) \Rightarrow$
 $\quad enat\ (\Delta\ \sigma\ (i+1)) > right\ I$
 $(\mathbf{P}\ I\ \varphi, VOnceOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $(\mathbf{P}\ I\ \varphi, VOnce\ i\ li\ vps) \Rightarrow$
 $\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ v_at\ vps = [li\ ..< (LTP_p\ \sigma\ i\ I) + 1]$
 $\quad \wedge (\forall vp \in set\ vps. v_check\ v\ \varphi\ vp))$
 $(\mathbf{F}\ I\ \varphi, VEventually\ i\ hi\ vps) \Rightarrow$
 $\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
 $\quad \wedge map\ v_at\ vps = [(ETP_f\ \sigma\ i\ I) ..< hi + 1]$
 $\quad \wedge (\forall vp \in set\ vps. v_check\ v\ \varphi\ vp))$
 $(\mathbf{H}\ I\ \varphi, VHistorically\ i\ vp) \Rightarrow$
 $\quad (let\ j = v_at\ vp\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v_check\ v\ \varphi\ vp)$
 $(\mathbf{G}\ I\ \varphi, VAlways\ i\ vp) \Rightarrow$
 $\quad (let\ j = v_at\ vp$
 $\quad in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v_check\ v\ \varphi\ vp)$
 $(\varphi\ \mathbf{S}\ I\ \psi, VSinceOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $(\varphi\ \mathbf{S}\ I\ \psi, VSince\ i\ vp1\ vp2s) \Rightarrow$
 $\quad (let\ j = v_at\ vp1\ in$
 $\quad (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b \leq j) \wedge j \leq i$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ v_at\ vp2s = [j\ ..< (LTP_p\ \sigma\ i\ I) + 1] \wedge v_check\ v\ \varphi\ vp1$
 $\quad \wedge (\forall vp2 \in set\ vp2s. v_check\ v\ \psi\ vp2))$
 $(\varphi\ \mathbf{S}\ I\ \psi, VSinceInf\ i\ li\ vp2s) \Rightarrow$
 $\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ v_at\ vp2s = [li\ ..< (LTP_p\ \sigma\ i\ I) + 1]$
 $\quad \wedge (\forall vp2 \in set\ vp2s. v_check\ v\ \psi\ vp2))$
 $(\varphi\ \mathbf{U}\ I\ \psi, VUntil\ i\ vp2s\ vp1) \Rightarrow$
 $\quad (let\ j = v_at\ vp1\ in$
 $\quad (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP_f\ \sigma\ i\ b) \wedge i \leq j$
 $\quad \wedge map\ v_at\ vp2s = [ETP_f\ \sigma\ i\ I\ ..< j + 1] \wedge v_check\ v\ \varphi\ vp1$
 $\quad \wedge (\forall vp2 \in set\ vp2s. v_check\ v\ \psi\ vp2))$
 $(\varphi\ \mathbf{U}\ I\ \psi, VUntilInf\ i\ hi\ vp2s) \Rightarrow$
 $\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$

$$\wedge \text{map } v_at \text{ } vp2s = [ETP_f \sigma i I ..< hi + 1]$$

$$\wedge (\forall vp2 \in \text{set } vp2s. v_check v \psi vp2))$$

$$| (_ , _) \Rightarrow \text{False}$$

```

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*:

$$s_check v \varphi sp \implies SAT \sigma v (s_at sp) \varphi$$

$$v_check v \varphi vp \implies VIO \sigma v (v_at vp) \varphi$$

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_SubVals[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:  $k \in \text{set } ([j \dots < \text{Suc } (LTP\_p \sigma i I)])$  using j_def k_def by auto
  then obtain x where  $x \in \text{set } \text{sps } s\_at \ x = k$  using k_def map
    unfolding set_map set_eq_iff image_iff
    by metis
  then have SAT  $\sigma \ v \ k \ \varphi$  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  $\varphi$ )
  case (Always I  $\varphi$ )
  obtain n where n_def:  $\text{right } I = \text{enat } n$ 
    using SAlways
    by (auto simp: Always split: enat.splits)
  {fix k
    define j where j_def:  $j \equiv LTP \ \sigma \ (\tau \ \sigma \ i + n)$ 
    assume k_def:  $k \leq j \wedge k \geq i \wedge k \geq ETP \ \sigma \ (\tau \ \sigma \ i + \text{left } I)$ 
    from SAlways Always j_def have map:  $\text{set } (\text{map } s\_at \ \text{sps}) = \text{set } [(ETP\_f \ \sigma \ i \ I) \dots < \text{Suc } j]$ 
      by (auto simp: Let_def n_def)
    then have kset:  $k \in \text{set } [(ETP\_f \ \sigma \ i \ I) \dots < \text{Suc } j]$  using k_def j_def by auto
    then obtain x where  $x \in \text{set } \text{sps } s\_at \ x = k$  using k_def map
      unfolding set_map set_eq_iff image_iff
      by metis
    then have SAT  $\sigma \ v \ k \ \varphi$  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  $\varphi$ )
  case (Until  $\varphi \ I \ \psi$ )
  show ?thesis
    using SUNtil(3)
    unfolding Until
  proof (intro SAT_VIO.SUntil[of _ s_at spsi], goal_cases le mem SAT $\psi$  SAT $\varphi$ )
    case (SAT $\varphi$  k)
    then show ?case
      by (cases  $k \leq s\_at \ (hd \ \text{sps})$ )
        (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  $\varphi$ ) (auto simp add: Let_def SAT_VIO.SNext)
next
case (SPrev sp)

```

```

    then show ?case by (cases  $\varphi$ ) (auto simp add: Let_def SAT_VIO.SPrev)
next
  case VFF
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VFF)
next
  case VPred
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VPred)
next
  case VEq_Const
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VEq_Const)
next
  case VNeg
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VNeg)
next
  case VOr
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VOr)
next
  case VAndL
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VAndL)
next
  case VAndR
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VAndR)
next
  case VImp
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VImp)
next
  case VIffSV
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VIffSV)
next
  case VIffVS
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VIffVS)
next
  case (VExists x part)
  then show ?case
  proof (cases  $\varphi$ )
    case (Exists y  $\psi$ )
    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  $z \in D$  ( $D, ?vp \in \text{SubsVals part}$ )
      by blast
      with VExists(2-) Exists have v_check (v(y := z))  $\psi$  ?vp v_at ?vp = v_at (VExists x part)
      by auto
      then show VIO  $\sigma$  (v(y := z)) (v_at (VExists x part))  $\psi$ 
      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  $\varphi$ ) (auto intro: SAT_VIO.VForall)
next
  case VOnceOut
  then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VOnceOut)
next
  case (VOnce i li vps)
  then show ?case

```



```

proof (cases  $\varphi$ )
  case (Once I  $\varphi$ )
    {fix k
      define j where j_def: j  $\equiv$  case right I of  $\infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \text{ETP } \sigma (\tau \sigma i - n)$ 
      assume k_def: k  $\geq j \wedge k \leq i \wedge k \leq \text{LTP } \sigma (\tau \sigma i - \text{left } I)$ 
      from VOnce Once j_def have map: set (map v_at vps) = set [j ..< Suc (LTP_p  $\sigma$  i I)]
        by (auto simp: Let_def)
      then have kset: k  $\in$  set ([j ..< Suc (LTP_p  $\sigma$  i I)]) using j_def k_def by auto
      then obtain x where x: x  $\in$  set vps v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO  $\sigma$  v k  $\varphi$  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  $\varphi$ )
    case (Eventually I  $\varphi$ )
    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  $\equiv$  LTP  $\sigma$  ( $\tau \sigma$  i + n)
      assume k_def: k  $\leq j \wedge k \geq i \wedge k \geq \text{ETP } \sigma (\tau \sigma i + \text{left } I)$ 
      from VEventually Eventually j_def have map: set (map v_at vps) = set [(ETP_f  $\sigma$  i I) ..< Suc j]
        by (auto simp: Let_def n_def)
      then have kset: k  $\in$  set [(ETP_f  $\sigma$  i I) ..< Suc j]) using k_def j_def by auto
      then obtain x where x: x  $\in$  set vps v_at x = k using k_def map
        unfolding set_map set_eq_iff image_iff
        by metis
      then have VIO  $\sigma$  v k  $\varphi$  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  $\varphi$ )
    case (Historically I  $\varphi$ )
    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  $\varphi$ )

```

```

    case (Always I  $\varphi$ )
  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  $\varphi$ ) (auto intro: SAT_VIO.VNext)
next
case VNextOutR
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VNextOutR)
next
case VNextOutL
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VNextOutL)
next
case VPrev
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VPrev)
next
case VPrevOutR
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VPrevOutR)
next
case VPrevOutL
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VPrevOutL)
next
case VPrevZ
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VPrevZ)
next
case VSinceOut
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VSinceOut)
next
case (VSince i vp vps)
then show ?case
proof (cases  $\varphi$ )
  case (Since  $\varphi$  I  $\psi$ )
  {fix k
    assume k_def:  $k \geq v\_at\ vp \wedge k \leq i \wedge k \leq LTP\ \sigma\ (\tau\ \sigma\ i - left\ I)$ 
    from VSince Since have map:  $set\ (map\ v\_at\ vps) = set\ ([ (v\_at\ vp) ..< Suc\ (LTP\_p\ \sigma\ i\ I)])$ 
    by (auto simp: Let_def)
    then have kset:  $k \in set\ ([ (v\_at\ vp) ..< Suc\ (LTP\_p\ \sigma\ i\ I)])$  using k_def by auto
    then obtain x where  $x \in set\ vps\ v\_at\ x = k$  using k_def map kset
    unfolding set_map set_eq_iff image_iff
    by metis
    then have VIO  $\sigma\ v\ k\ \psi$  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  $\varphi$ )
  case (Until  $\varphi$  I  $\psi$ )
  {fix k

```

```

assume  $k\_def: k \leq v\_at\ vp \wedge k \geq i \wedge k \geq ETP\ \sigma\ (\tau\ \sigma\ i + left\ I)$ 
from  $VUntil\ Until$  have  $map: set\ (map\ v\_at\ vps) = set\ [(ETP\_f\ \sigma\ i\ I) \dots < Suc\ (v\_at\ vp)]$ 
  by  $(auto\ simp: Let\_def)$ 
then have  $kset: k \in set\ [(ETP\_f\ \sigma\ i\ I) \dots < Suc\ (v\_at\ vp)]$  using  $k\_def$  by  $auto$ 
then obtain  $x$  where  $x: x \in set\ vps\ v\_at\ x = k$  using  $k\_def\ map\ kset$ 
  unfolding  $set\_map\ set\_eq\_iff\ image\_iff$ 
  by  $metis$ 
then have  $VIO\ \sigma\ v\ k\ \psi$  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  $(cases\ \varphi)$ 
  case  $(Since\ \varphi\ I\ \psi)$ 
    {fix  $k$ 
      define  $j$  where  $j\_def: j \equiv case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ n \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - n)$ 
      assume  $k\_def: k \geq j \wedge k \leq i \wedge k \leq LTP\ \sigma\ (\tau\ \sigma\ i - left\ I)$ 
      from  $VSinceInf\ Since\ j\_def$  have  $map: set\ (map\ v\_at\ vps) = set\ [j \dots < Suc\ (LTP\_p\ \sigma\ i\ I)]$ 
        by  $(auto\ simp: Let\_def)$ 
      then have  $kset: k \in set\ [j \dots < Suc\ (LTP\_p\ \sigma\ i\ I)]$  using  $j\_def\ k\_def$  by  $auto$ 
      then obtain  $x$  where  $x: x \in set\ vps\ v\_at\ x = k$  using  $k\_def\ map$ 
        unfolding  $set\_map\ set\_eq\_iff\ image\_iff$ 
        by  $metis$ 
      then have  $VIO\ \sigma\ v\ k\ \psi$  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  $(cases\ \varphi)$ 
  case  $(Until\ \varphi\ I\ \psi)$ 
    obtain  $n$  where  $n\_def: right\ I = enat\ n$ 
      using  $VUntilInf$ 
      by  $(auto\ simp: Until\ split: enat.splits)$ 
    {fix  $k$ 
      define  $j$  where  $j\_def: j \equiv LTP\ \sigma\ (\tau\ \sigma\ i + n)$ 
      assume  $k\_def: k \leq j \wedge k \geq i \wedge k \geq ETP\ \sigma\ (\tau\ \sigma\ i + left\ I)$ 
      from  $VUntilInf\ Until\ j\_def$  have  $map: set\ (map\ v\_at\ vps) = set\ [(ETP\_f\ \sigma\ i\ I) \dots < Suc\ j]$ 
        by  $(auto\ simp: Let\_def\ n\_def)$ 
      then have  $kset: k \in set\ [(ETP\_f\ \sigma\ i\ I) \dots < Suc\ j]$  using  $k\_def\ j\_def$  by  $auto$ 
      then obtain  $x$  where  $x: x \in set\ vps\ v\_at\ x = k$  using  $k\_def\ map$ 
        unfolding  $set\_map\ set\_eq\_iff\ image\_iff$ 
        by  $metis$ 
      then have  $VIO\ \sigma\ v\ k\ \psi$  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 $v \longleftrightarrow (\forall x \in X. v \ x \in vs \ x)$

definition compatible_vals X vs = $\{v. \forall x \in X. v \ x \in vs \ x\}$

lemma compatible_alt:
 compatible X vs $v \longleftrightarrow v \in$ compatible_vals X vs
by (auto simp: compatible_def compatible_vals_def)

lemma compatible_empty_iff: compatible $\{\}$ vs $v \longleftrightarrow$ 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 $(X \cup Y)$ vs $v \longleftrightarrow$ compatible X vs $v \wedge$ compatible Y vs v
by (auto simp: compatible_def)

lemma compatible_vals_union_eq:
 compatible_vals $(X \cup Y)$ vs = compatible_vals X vs \cap compatible_vals Y vs
by (auto simp: compatible_vals_def)

lemma compatible_antimono:
 compatible X vs $v \implies Y \subseteq X \implies$ compatible Y vs v
by (auto simp: compatible_def)

lemma compatible_vals_antimono:
 $Y \subseteq X \implies$ compatible_vals X vs \subseteq compatible_vals Y vs
by (auto simp: compatible_vals_def)

lemma compatible_extensible:
 $(\forall x. vs \ x \neq \{\}) \implies$ compatible X vs $v \implies X \subseteq Y \implies \exists v'.$ compatible Y vs $v' \wedge (\forall x \in X. v \ x = v' \ x)$
using some_in_eq[of vs _] **by** (auto simp: override_on_def compatible_def
 intro: exI[**where** $x =$ override_on v ($\lambda x. \text{SOME } y. y \in vs \ x$) ($Y - X$)])

lemmas compatible_vals_extensible = compatible_extensible[unfolded compatible_alt]

primrec mk_values :: $((n, d) \text{ trm} \times 'a \text{ set}) \text{ list} \Rightarrow 'a \text{ list set}$
where mk_values $\square = \{\square\}$
 $|$ mk_values $(T \# Ts) =$ (case T of
 $(\mathbf{v} \ x, X) \Rightarrow$
 let terms = map fst Ts in
 if $\mathbf{v} \ x \in$ set terms then
 let fst_pos = hd (positions terms $(\mathbf{v} \ x)$) in $(\lambda xs. (xs ! \text{fst_pos}) \# xs)$ ' (mk_values Ts)
 else set_Cons X (mk_values Ts)
 $|$ $(\mathbf{c} \ a, X) \Rightarrow$ set_Cons X (mk_values Ts)

lemma mk_values_nempty:
 $\{\} \notin$ set (map snd tXs) \implies mk_values $tXs \neq \{\}$
by (induct tXs)
 (auto simp: set_Cons_def image_iff split: trm.splits if_splits)

lemma *mk_values_not_Nil*:

$\{\} \notin \text{set } (\text{map } \text{snd } tXs) \implies tXs \neq [] \implies vs \in \text{mk_values } tXs \implies vs \neq []$

by (*induct tXs*)

(*auto simp: set_Cons_def image_iff split: trm.splits if_splits*)

lemma *mk_values_nth_cong*: $\mathbf{v} \ x \in \text{set } (\text{map } \text{fst } tXs) \implies$

$n \in \text{set } (\text{positions } (\text{map } \text{fst } tXs) (\mathbf{v} \ x)) \implies$

$m \in \text{set } (\text{positions } (\text{map } \text{fst } tXs) (\mathbf{v} \ x)) \implies$

$vs \in \text{mk_values } tXs \implies$

$vs ! n = vs ! m$

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*

$\longleftrightarrow (\text{let } (\text{fintXs}, \text{inftXs}) = \text{partition } (\lambda tX. \text{finite } (\text{snd } tX)) \ tXs \text{ in}$

if inftXs = [] then {p} \times mk_values tXs \subseteq X

else let inf_dups = filter ($\lambda tX. (\text{fst } tX) \in \text{set } (\text{map } \text{fst } \text{fintXs})$) inftXs in

if inf_dups = [] then (if finite X then False else Code.abort STR "subset on infinite subset" ($\lambda _ . \{p\}$

\times mk_values tXs \subseteq X))

else if list_all ($\lambda tX. \text{Max } (\text{set } (\text{positions } tXs \ tX)) < \text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tX)))$)

inf_dups

then {p} \times mk_values tXs \subseteq X

else (if finite X then False else Code.abort STR "subset on infinite subset" ($\lambda _ . \{p\} \times \text{mk_values}$

tXs \subseteq X)))

lemma *mk_values_nemptyI*: $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies \text{mk_values } tXs \neq \{\}$

by (*induct tXs*)

(*auto simp: Let_def set_Cons_eq split: prod.splits trm.splits*)

lemma *infinite_mk_values1*: $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies tY \in \text{set } tXs \implies \forall Y. (\text{fst } tY, Y) \in \text{set } tXs \longrightarrow \text{infinite } Y \implies \text{infinite } (\text{mk_values } tXs)$

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 $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\}$
by (*simp add: Cons.prem(1)*)
moreover have $\forall Z. (\text{trm.Var } x, Z) \in \text{set } tXs \longrightarrow \text{infinite } Z$
using *Cons.prem(2,3) var_in*
by (*cases tY \in set tXs; clarsimp*)
(metis (no_types, lifting) Cons.hyps Cons.prem(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 (\lambda xs. 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 \neq \{\}*
using *Cons.prem*
by (*auto intro!: mk_values_emptyI*)
then show *?case*
using *Cons var_out infinite_set_ConsI(1)[OF \langle mk_values tXs \neq \{\} \rangle \langle infinite (snd tX) \rangle]*
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 \neq \{\}*
using *Cons.prem*
by (*auto intro!: mk_values_emptyI*)
then show *?case*
using *Cons const infinite_set_ConsI(1)[OF \langle mk_values tXs \neq \{\} \rangle \langle infinite (snd tX) \rangle]*
by auto

qed
qed *simp*

lemma *infinite_mk_values2*: $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies tY \in \text{set } tXs \implies \text{infinite } (\text{snd } tY) \implies \text{Max } (\text{set } (\text{positions } tXs \ tY)) \geq \text{Max } (\text{set } (\text{positions } (\text{map fst } tXs) (\text{fst } tY))) \implies \text{infinite } (\text{mk_values } tXs)$

proof (*induct tXs arbitrary: tY*)
case (*Cons tX tXs*)
hence *obs1: \forall tX \in set tXs. snd tX \neq \{\}*
by (*simp add: Cons.prem(1)*)
note *IH = Cons.hyps[OF obs1 _ \langle infinite (snd tY) \rangle]*

```

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.prem1 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.prem1
  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:  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies$ 
   $\text{mk\_values\_subset } p \ tXs \ X \iff \{p\} \times \text{mk\_values } tXs \subseteq X$ 
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 bx_simps
proof safe
  assume  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\}$  and finite X
  and filter1: filter ( $\lambda xy. \text{infinite } (\text{snd } xy) \wedge (\exists ab. (ab \in \text{set } tXs \wedge \text{finite } (\text{snd } ab)) \wedge \text{fst } xy = \text{fst } ab)$ )
  tXs = []
  and filter2: filter ( $\lambda x. \text{infinite } (\text{snd } x)$ ) tXs  $\neq []$ 
then obtain tY where  $tY \in \text{set } tXs$  and infinite (snd tY)
  by (meson filter_False)
moreover have  $\forall Y. (\text{fst } tY, Y) \in \text{set } tXs \implies \text{infinite } Y$ 
  using filter1 calculation
  by (auto simp: filter_empty_conv)
ultimately have infinite (mk_values tXs)
  using infinite_mk_values1[OF  $\langle \forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \rangle$ ]
  by auto
hence infinite ( $\{p\} \times \text{mk\_values } tXs$ )
  using finite_cartesian_productD2 by auto
thus  $\{p\} \times \text{mk\_values } tXs \subseteq X \implies \text{False}$ 
  using  $\langle \text{finite } X \rangle$ 
  by (simp add: finite_subset)
next
assume  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\}$ 
  and finite X
  and ex_dupl_inf:  $\neg \text{list\_all } (\lambda tX. \text{Max } (\text{set } (\text{positions } tXs \ tX)))$ 
   $< \text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tX)))$ 
   $(\text{filter } (\lambda xy. \text{infinite } (\text{snd } xy) \wedge (\exists ab. (ab \in \text{set } tXs \wedge \text{finite } (\text{snd } ab)) \wedge \text{fst } xy = \text{fst } ab)) \ tXs)$ 
  and filter: filter ( $\lambda x. \text{infinite } (\text{snd } x)$ ) tXs  $\neq []$ 
then obtain tY and Z where  $tY \in \text{set } tXs$ 
  and infinite (snd tY)
  and  $(\text{fst } tY, Z) \in \text{set } tXs$ 
  and finite Z
  and  $\text{Max } (\text{set } (\text{positions } tXs \ tY)) \geq \text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tY)))$ 
  by (auto simp: list_all_iff)
hence infinite (mk_values tXs)
  using infinite_mk_values2[OF  $\langle \forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \rangle \langle tY \in \text{set } tXs \rangle$ ]
  by auto
hence infinite ( $\{p\} \times \text{mk\_values } tXs$ )
  using finite_cartesian_productD2 by auto
thus  $\{p\} \times \text{mk\_values } tXs \subseteq X \implies \text{False}$ 
  using  $\langle \text{finite } X \rangle$ 
  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.prem
  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.prem 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.prem 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 $mk_values_subset_Compl\ r\ vs\ ts\ i = (\{r\} \times mk_values\ (vs\{\!\!\{ts\}\!\!\}) \subseteq -\ \Gamma\ \sigma\ i)$

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  $v \in compatible\_vals (\cup (fv\_trm\ ' set\ ts) - dom\ f)\ vs \wedge x\ y.\ f\ x = Some\ y \implies y \in vs\ x$ 
shows  $check\_values\ vs\ ts\ ((\lambda x.\ case\ f\ x\ of\ None \Rightarrow v\ x \mid Some\ y \Rightarrow y)\{\!\!\{ts\}\!\!\})\ (Some\ f) \neq None$ 
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  $v\_in[simp]: v\ x \in vs\ x$ 
by (auto simp: compatible_vals_def)
from Cons(2) have  $v \in compatible\_vals (\cup (fv\_trm\ ' set\ ts) - dom\ (f(x \mapsto v\ x)))\ vs$ 
by (auto simp: compatible_vals_def)
then have  $check\_values\ vs\ ts$ 
   $((\lambda z.\ case\ (f(x \mapsto v\ x))\ z\ of\ None \Rightarrow v\ z \mid Some\ y \Rightarrow y)\{\!\!\{ts\}\!\!\})$ 
   $(Some\ (f(x \mapsto v\ x))) \neq None$ 
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:
   $\forall v \in \text{compatible\_vals} \left( \bigcup (fv\_trm \text{ ' set } ts) - \text{dom } f \right) vs. us \neq (\lambda x. \text{case } f \text{ } x \text{ of } \text{None} \Rightarrow v \text{ } x \mid \text{Some } y \Rightarrow y) \llbracket ts \rrbracket \implies$ 
  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_Cmpl_code[code]:
  mk_values_subset_Cmpl r vs ts i =  $(\forall (q, us) \in \Gamma \sigma i. q \neq r \vee \text{check\_values } vs \text{ } ts \text{ } us \text{ } (\text{Some } \text{Map.empty}) = \text{None})$ 
  unfolding mk_values_subset_Cmpl_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  $\Rightarrow$  ('n, 'd) formula  $\Rightarrow$  ('n, 'd) sproof  $\Rightarrow$  bool
and v_check_exec :: ('n, 'd) envset  $\Rightarrow$  ('n, 'd) formula  $\Rightarrow$  ('n, 'd) vproof  $\Rightarrow$  bool where
  s_check_exec vs f p = (case (f, p) of
    ( $\top$ , STT i)  $\Rightarrow$  True
  | (r  $\dagger$  ts, SPred i s ts')  $\Rightarrow$ 
    (r = s  $\wedge$  ts = ts'  $\wedge$  mk_values_subset r (vs  $\llbracket ts \rrbracket$ ) ( $\Gamma \sigma i$ ))
  | (x  $\approx$  c, SEq_Const i x' c')  $\Rightarrow$ 
    (c = c'  $\wedge$  x = x'  $\wedge$  vs x = {c})
  | ( $\neg_F \varphi$ , SNeg vp)  $\Rightarrow$  v_check_exec vs  $\varphi$  vp
  | ( $\varphi \vee_F \psi$ , SOrL sp1)  $\Rightarrow$  s_check_exec vs  $\varphi$  sp1
  | ( $\varphi \vee_F \psi$ , SOrR sp2)  $\Rightarrow$  s_check_exec vs  $\psi$  sp2

```

$| (\varphi \wedge_F \psi, SAnd\ sp1\ sp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \longrightarrow_F \psi, SImpL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1$
 $| (\varphi \longrightarrow_F \psi, SImpR\ sp2) \Rightarrow s_check_exec\ vs\ \psi\ sp2$
 $| (\varphi \longleftrightarrow_F \psi, SIffSS\ sp1\ sp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \longleftrightarrow_F \psi, SIffVV\ vp1\ vp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\exists_F x. \varphi, SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s_check_exec\ (vs\ (x := \{val\}))\ \varphi\ sp)$
 $| (\forall_F x. \varphi, SForall\ y\ sp_part) \Rightarrow (let\ i = s_at\ (part_hd\ sp_part)$
 $\quad in\ x = y \wedge (\forall (sub, sp) \in SubsVals\ sp_part. s_at\ sp = i \wedge s_check_exec\ (vs\ (x := sub))\ \varphi\ sp))$
 $| (\mathbf{Y}\ I\ \varphi, SPrev\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp; i = s_at\ (SPrev\ sp)\ in$
 $\quad i = j+1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{X}\ I\ \varphi, SNext\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp; i = s_at\ (SNext\ sp)\ in$
 $\quad j = i+1 \wedge mem\ (\Delta\ \sigma\ j)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{P}\ I\ \varphi, SOnce\ i\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{F}\ I\ \varphi, SEventually\ i\ sp) \Rightarrow$
 $\quad (let\ j = s_at\ sp\ in$
 $\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{H}\ I\ \varphi, SHistoricallyOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{H}\ I\ \varphi, SHistorically\ i\ li\ sps) \Rightarrow$
 $\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - b))$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ s_at\ sps = [li\ ..< (LTP_p\ \sigma\ i\ I) + 1]$
 $\quad \wedge (\forall sp \in set\ sps. s_check_exec\ vs\ \varphi\ sp))$
 $| (\mathbf{G}\ I\ \varphi, SAlways\ i\ hi\ sps) \Rightarrow$
 $\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b)$
 $\quad \wedge right\ I \neq \infty$
 $\quad \wedge map\ s_at\ sps = [(ETP_f\ \sigma\ i\ I) ..< hi + 1]$
 $\quad \wedge (\forall sp \in set\ sps. s_check_exec\ vs\ \varphi\ sp))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, SSince\ sp2\ sp1s) \Rightarrow$
 $\quad (let\ i = s_at\ (SSince\ sp2\ sp1s); j = s_at\ sp2\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
 $\quad \wedge map\ s_at\ sp1s = [j+1\ ..< i+1]$
 $\quad \wedge s_check_exec\ vs\ \psi\ sp2$
 $\quad \wedge (\forall sp1 \in set\ sp1s. s_check_exec\ vs\ \varphi\ sp1))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, SUntil\ sp1s\ sp2) \Rightarrow$
 $\quad (let\ i = s_at\ (SUntil\ sp1s\ sp2); j = s_at\ sp2\ in$
 $\quad j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$
 $\quad \wedge map\ s_at\ sp1s = [i\ ..< j] \wedge s_check_exec\ vs\ \psi\ sp2$
 $\quad \wedge (\forall sp1 \in set\ sp1s. s_check_exec\ vs\ \varphi\ sp1))$
 $| (_ , _) \Rightarrow False$
 $| v_check_exec\ vs\ fp = (case\ (f, p)\ of$
 $\quad (\perp, VFF\ i) \Rightarrow True$
 $\quad (r \dagger ts, VPred\ i\ pred\ ts') \Rightarrow$
 $\quad (r = pred \wedge ts = ts' \wedge mk_values_subset_Compl\ r\ vs\ ts\ i)$
 $\quad (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow$
 $\quad (c = c' \wedge x = x' \wedge c \notin vs\ x)$
 $\quad (\neg_F\ \varphi, VNeg\ sp) \Rightarrow s_check_exec\ vs\ \varphi\ sp$
 $\quad (\varphi \vee_F \psi, VOr\ vp1\ vp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $\quad (\varphi \wedge_F \psi, VAndL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1$
 $\quad (\varphi \wedge_F \psi, VAndR\ vp2) \Rightarrow v_check_exec\ vs\ \psi\ vp2$
 $\quad (\varphi \longrightarrow_F \psi, VImp\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$

$| (\varphi \longleftrightarrow_F \psi, \text{ViffSV } sp1 \text{ } vp2) \Rightarrow s_check_exec \text{ vs } \varphi \text{ } sp1 \wedge v_check_exec \text{ vs } \psi \text{ } vp2 \wedge s_at \text{ } sp1 = v_at \text{ } vp2$
 $| (\varphi \longleftrightarrow_F \psi, \text{ViffVS } vp1 \text{ } sp2) \Rightarrow v_check_exec \text{ vs } \varphi \text{ } vp1 \wedge s_check_exec \text{ vs } \psi \text{ } sp2 \wedge v_at \text{ } vp1 = s_at \text{ } sp2$
 $| (\exists_{Fx}. \varphi, \text{VExists } y \text{ } vp_part) \Rightarrow (\text{let } i = v_at \text{ } (part_hd \text{ } vp_part)$
 $\quad \text{in } x = y \wedge (\forall (sub, vp) \in \text{SubsVals } vp_part. v_at \text{ } vp = i \wedge v_check_exec \text{ } (vs \text{ } (x := sub)) \varphi \text{ } vp))$
 $| (\forall_{Fx}. \varphi, \text{VForall } y \text{ } val \text{ } vp) \Rightarrow (x = y \wedge v_check_exec \text{ } (vs \text{ } (x := \{val\})) \varphi \text{ } vp)$
 $| (\mathbf{Y} \text{ } I \varphi, \text{VPrev } vp) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp; i = v_at \text{ } (\text{VPrev } vp) \text{ in}$
 $\quad i = j+1 \wedge v_check_exec \text{ vs } \varphi \text{ } vp)$
 $| (\mathbf{Y} \text{ } I \varphi, \text{VPrevZ}) \Rightarrow \text{True}$
 $| (\mathbf{Y} \text{ } I \varphi, \text{VPrevOutL } i) \Rightarrow$
 $\quad i > 0 \wedge \Delta \sigma \text{ } i < \text{left } I$
 $| (\mathbf{Y} \text{ } I \varphi, \text{VPrevOutR } i) \Rightarrow$
 $\quad i > 0 \wedge \text{enat } (\Delta \sigma \text{ } i) > \text{right } I$
 $| (\mathbf{X} \text{ } I \varphi, \text{VNext } vp) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp; i = v_at \text{ } (\text{VNext } vp) \text{ in}$
 $\quad j = i+1 \wedge v_check_exec \text{ vs } \varphi \text{ } vp)$
 $| (\mathbf{X} \text{ } I \varphi, \text{VNextOutL } i) \Rightarrow$
 $\quad \Delta \sigma \text{ } (i+1) < \text{left } I$
 $| (\mathbf{X} \text{ } I \varphi, \text{VNextOutR } i) \Rightarrow$
 $\quad \text{enat } (\Delta \sigma \text{ } (i+1)) > \text{right } I$
 $| (\mathbf{P} \text{ } I \varphi, \text{VOnceOut } i) \Rightarrow$
 $\quad \tau \sigma \text{ } i < \tau \sigma \text{ } 0 + \text{left } I$
 $| (\mathbf{P} \text{ } I \varphi, \text{VOnce } i \text{ } li \text{ } vps) \Rightarrow$
 $\quad (li = (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } b \Rightarrow \text{ETP_p } \sigma \text{ } i \text{ } b)$
 $\quad \wedge \tau \sigma \text{ } 0 + \text{left } I \leq \tau \sigma \text{ } i$
 $\quad \wedge \text{map } v_at \text{ } vps = [li \text{ } ..< (\text{LTP_p } \sigma \text{ } i \text{ } I) + 1]$
 $\quad \wedge (\forall vp \in \text{set } vps. v_check_exec \text{ vs } \varphi \text{ } vp))$
 $| (\mathbf{F} \text{ } I \varphi, \text{VEventually } i \text{ } hi \text{ } vps) \Rightarrow$
 $\quad (hi = (\text{case right } I \text{ of } \text{enat } b \Rightarrow \text{LTP_f } \sigma \text{ } i \text{ } b) \wedge \text{right } I \neq \infty$
 $\quad \wedge \text{map } v_at \text{ } vps = [(\text{ETP_f } \sigma \text{ } i \text{ } I) \text{ } ..< \text{hi} + 1]$
 $\quad \wedge (\forall vp \in \text{set } vps. v_check_exec \text{ vs } \varphi \text{ } vp))$
 $| (\mathbf{H} \text{ } I \varphi, \text{VHistorically } i \text{ } vp) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp \text{ in}$
 $\quad j \leq i \wedge \text{mem } (\tau \sigma \text{ } i - \tau \sigma \text{ } j) \text{ } I \wedge v_check_exec \text{ vs } \varphi \text{ } vp)$
 $| (\mathbf{G} \text{ } I \varphi, \text{VAlways } i \text{ } vp) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp$
 $\quad \text{in } j \geq i \wedge \text{mem } (\tau \sigma \text{ } j - \tau \sigma \text{ } i) \text{ } I \wedge v_check_exec \text{ vs } \varphi \text{ } vp)$
 $| (\varphi \text{ } \mathbf{S} \text{ } I \psi, \text{VSinceOut } i) \Rightarrow$
 $\quad \tau \sigma \text{ } i < \tau \sigma \text{ } 0 + \text{left } I$
 $| (\varphi \text{ } \mathbf{S} \text{ } I \psi, \text{VSince } i \text{ } vp1 \text{ } vp2s) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp1 \text{ in}$
 $\quad (\text{case right } I \text{ of } \infty \Rightarrow \text{True} \mid \text{enat } b \Rightarrow \text{ETP_p } \sigma \text{ } i \text{ } b \leq j) \wedge j \leq i$
 $\quad \wedge \tau \sigma \text{ } 0 + \text{left } I \leq \tau \sigma \text{ } i$
 $\quad \wedge \text{map } v_at \text{ } vp2s = [j \text{ } ..< (\text{LTP_p } \sigma \text{ } i \text{ } I) + 1] \wedge v_check_exec \text{ vs } \varphi \text{ } vp1$
 $\quad \wedge (\forall vp2 \in \text{set } vp2s. v_check_exec \text{ vs } \psi \text{ } vp2))$
 $| (\varphi \text{ } \mathbf{S} \text{ } I \psi, \text{VSinceInf } i \text{ } li \text{ } vp2s) \Rightarrow$
 $\quad (li = (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } b \Rightarrow \text{ETP_p } \sigma \text{ } i \text{ } b)$
 $\quad \wedge \tau \sigma \text{ } 0 + \text{left } I \leq \tau \sigma \text{ } i$
 $\quad \wedge \text{map } v_at \text{ } vp2s = [li \text{ } ..< (\text{LTP_p } \sigma \text{ } i \text{ } I) + 1]$
 $\quad \wedge (\forall vp2 \in \text{set } vp2s. v_check_exec \text{ vs } \psi \text{ } vp2))$
 $| (\varphi \text{ } \mathbf{U} \text{ } I \psi, \text{VUntil } i \text{ } vp2s \text{ } vp1) \Rightarrow$
 $\quad (\text{let } j = v_at \text{ } vp1 \text{ in}$
 $\quad (\text{case right } I \text{ of } \infty \Rightarrow \text{True} \mid \text{enat } b \Rightarrow j < \text{LTP_f } \sigma \text{ } i \text{ } b) \wedge i \leq j$
 $\quad \wedge \text{map } v_at \text{ } vp2s = [\text{ETP_f } \sigma \text{ } i \text{ } I \text{ } ..< j + 1] \wedge v_check_exec \text{ vs } \varphi \text{ } vp1$
 $\quad \wedge (\forall vp2 \in \text{set } vp2s. v_check_exec \text{ vs } \psi \text{ } vp2))$
 $| (\varphi \text{ } \mathbf{U} \text{ } I \psi, \text{VUntilInf } i \text{ } hi \text{ } vp2s) \Rightarrow$

```

    (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: formula.split sproof.split)
simps_of_case v_check_exec_simps[simp, code]: v_check_exec.simps[unfolded prod.case] (splits: formula.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  $\varphi$ )
{
  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  $\varphi1 \varphi2$ )
{
  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  $\varphi1 \varphi2$ )
{
  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  $\varphi1 \varphi2$ )
{
  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  $\varphi1 \varphi2$ )
{
  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  $x \varphi$ )

```



```

{
  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  $x \varphi$ )
{
  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  $I \varphi$ )
{
  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  $I \varphi$ )
{
  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  $I \varphi$ )
{
  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  $I \varphi$ )
{
  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  $\varphi$ )
{
  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  $\varphi$ )
{
  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  $\varphi 1$  I  $\varphi 2$ )
{
  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  $\varphi 1$  I  $\varphi 2$ )
{
  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]:
   $x \notin fv \varphi \implies s\_check (v(x := t)) \varphi sp = s\_check v \varphi sp$ 
  by (rule check_fv_cong) auto
lemma v_check_fun_upd_notin[simp]:
   $x \notin fv \varphi \implies v\_check (v(x := t)) \varphi sp = v\_check v \varphi sp$ 
  by (rule check_fv_cong) auto

```

```

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

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

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

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

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

lemma fun_upd_in_compatible_vals_notin:  $x \notin A \implies v \in \text{compatible\_vals } A \text{ vs} \implies v(x := t) \in$ 
 $\text{compatible\_vals } A \text{ vs}$ 
  unfolding compatible_vals_def
  by auto

lemma check_exec_check:
  assumes  $\forall x. \text{vs } x \neq \{\}$ 
  shows  $s\_check\_exec \text{ vs } \varphi \text{ sp} \longleftrightarrow (\forall v \in \text{compatible\_vals } (fv \varphi) \text{ vs. } s\_check \text{ v } \varphi \text{ sp})$ 
  and  $v\_check\_exec \text{ vs } \varphi \text{ vp} \longleftrightarrow (\forall v \in \text{compatible\_vals } (fv \varphi) \text{ vs. } v\_check \text{ v } \varphi \text{ vp})$ 
  using assms
proof (induct  $\varphi$  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:  $\forall tX \in \text{set } (vs \{ts\}). \text{snd } tX \neq \{\}$ 
  using  $\langle \forall x. vs \ x \neq \{\} \rangle$ 
  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  $\varphi$ )
{
  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  $\varphi1 \ \varphi2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi1 \cup \text{fv } \varphi2$ ] show ?case
  proof (cases sp)
  case (SOrL sp')
  from check_fv_cong(1)[of  $\varphi1 \ \_ \ sp'$ ] 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  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
proof (cases vp)
case (VOr vp1 vp2)
from check_fv_cong(2)[of  $\varphi 1$  _ _ vp1] check_fv_cong(2)[of  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
case  $\varphi 1$ 
then show ?case
by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
next
case  $\varphi 2$ 
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  $\varphi 1 \varphi 2$ )
{
case 1
with compatible_vals_nonemptyI[OF 1, of fv  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
proof (cases sp)
case (SAnd sp1 sp2)
from check_fv_cong(1)[of  $\varphi 1$  _ _ sp1] check_fv_cong(1)[of  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
case  $\varphi 1$ 
then show ?case
by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
next
case  $\varphi 2$ 
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  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
proof (cases vp)
case (VAndL vp')
from check_fv_cong(2)[of  $\varphi 1$  _ _ 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  $\varphi 2$  _ _ 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  $\varphi 1 \varphi 2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi 1 \cup \text{fv } \varphi 2$ ] show ?case
  proof (cases sp)
    case (SImpL vp')
    from check_fv_cong(2)[of  $\varphi 1 \_ \_ \text{vp}'$ ] 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  $\varphi 2 \_ \_ \text{sp}'$ ] 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  $\varphi 1 \cup \text{fv } \varphi 2$ ] show ?case
proof (cases vp)
  case (VImp sp1 vp2)
  from check_fv_cong(1)[of  $\varphi 1 \_ \_ \text{sp1}$ ] check_fv_cong(2)[of  $\varphi 2 \_ \_ \text{vp2}$ ] 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  $\varphi 1 \cup \text{fv } \varphi 2$ ]]
  proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
    case  $\varphi 1$ 
    then show ?case
    by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
  next
  case  $\varphi 2$ 
  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  $\varphi 1 \varphi 2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi 1 \cup \text{fv } \varphi 2$ ] show ?case
  proof (cases sp)
    case (SIffSS sp1 sp2)
    from check_fv_cong(1)[of  $\varphi 1 \_ \_ \text{sp1}$ ] check_fv_cong(1)[of  $\varphi 2 \_ \_ \text{sp2}$ ] 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  $\varphi 1 \cup \text{fv } \varphi 2$ ]]
    proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
      case  $\varphi 1$ 
      then show ?case
      by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
    next
      case  $\varphi 2$ 
      then show ?case
      by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    qed
  qed
}

```

```

next
  case (SIffVV vp1 vp2)
  from check_fv_cong(2)[of  $\varphi 1$  __ vp1] check_fv_cong(2)[of  $\varphi 2$  __ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
  proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
    case  $\varphi 1$ 
    then show ?case
  by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
next
  case  $\varphi 2$ 
  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  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
proof (cases vp)
  case (VIffSV sp1 vp2)
  from check_fv_cong(1)[of  $\varphi 1$  __ sp1] check_fv_cong(2)[of  $\varphi 2$  __ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
  proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
    case  $\varphi 1$ 
    then show ?case
  by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
next
  case  $\varphi 2$ 
  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  $\varphi 1$  __ vp1] check_fv_cong(1)[of  $\varphi 2$  __ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] refl, goal_cases  $\varphi 1 \varphi 2$ )
  case  $\varphi 1$ 
  then show ?case
  by (metis (mono_tags, lifting) 2 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
next
  case  $\varphi 2$ 
  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  $\varphi$ )
{
  case 1
  then have (vs(x := Z))  $y \neq \{\}$  if  $Z \neq \{\}$  for  $Z y$ 
  using that by auto
  with 1 have IH:

```

```

    s_check_exec (vs(x := {z}))  $\varphi$  sp = ( $\forall v \in \text{compatible\_vals}$  (fv  $\varphi$ ) (vs(x := {z})). s_check v  $\varphi$  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  $\varphi$  - {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))  $y \neq \{\}$  if  $Z \neq \{\}$  for  $Z$   $y$ 
  using that by auto
with 2 have IH:
   $Z \neq \{\} \implies v\_check\_exec$  (vs(x := Z))  $\varphi$  vp = ( $\forall v \in \text{compatible\_vals}$  (fv  $\varphi$ ) (vs(x := Z))). v_check
v  $\varphi$  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  $\varphi$  - {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  $\varphi$ )
{
  case 1
  then have (vs(x := Z))  $y \neq \{\}$  if  $Z \neq \{\}$  for  $Z$   $y$ 
    using that by auto
  with 1 have IH:
     $Z \neq \{\} \implies s\_check\_exec$  (vs(x := Z))  $\varphi$  sp = ( $\forall v \in \text{compatible\_vals}$  (fv  $\varphi$ ) (vs(x := Z))). s_check v
 $\varphi$  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  $\varphi$  - {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))  $y \neq \{\}$  if  $Z \neq \{\}$  for  $Z$   $y$ 
  using that by auto
with 2 have IH:
  v_check_exec (vs(x := {z}))  $\varphi$  vp = ( $\forall v \in \text{compatible\_vals}$  (fv  $\varphi$ ) (vs(x := {z}))). v_check v  $\varphi$  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  $\varphi - \{x\}$ ]
      by (cases vp) (auto simp: SubsVals_nonempty IH fun_upd_in_compatible_vals_notin_compatible_vals_fun_upd)
    }
  next
    case (Prev I  $\varphi$ )
    {
      case 1
      with Prev[of vs] show ?case
        using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
        by (cases sp) auto
      next
        case 2
        with Prev[of vs] show ?case
          using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
          by (cases vp) auto
    }
  next
    case (Next I  $\varphi$ )
    {
      case 1
      with Next[of vs] show ?case
        using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
        by (cases sp) (auto simp: Let_def)
      next
        case 2
        with Next[of vs] show ?case
          using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
          by (cases vp) auto
    }
  next
    case (Once I  $\varphi$ )
    {
      case 1
      with Once[of vs] show ?case
        using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
        by (cases sp) (auto simp: Let_def)
      next
        case 2
        with Once[of vs] show ?case
          using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
          by (cases vp) auto
    }
  next
    case (Historically I  $\varphi$ )
    {
      case 1
      with Historically[of vs] show ?case
        using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
        by (cases sp) auto
      next
        case 2
        with Historically[of vs] show ?case
          using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
          by (cases vp) (auto simp: Let_def)
    }
  next

```

```

case (Eventually I  $\varphi$ )
{
  case 1
  with Eventually[of vs] show ?case
  using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
  by (cases sp) (auto simp: Let_def)
next
  case 2
  with Eventually[of vs] show ?case
  using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
  by (cases vp) auto
}
next
case (Always I  $\varphi$ )
{
  case 1
  with Always[of vs] show ?case
  using compatible_vals_nonemptyI[OF 1, of fv  $\varphi$ ]
  by (cases sp) auto
next
  case 2
  with Always[of vs] show ?case
  using compatible_vals_nonemptyI[OF 2, of fv  $\varphi$ ]
  by (cases vp) (auto simp: Let_def)
}
next
case (Since  $\varphi1$  I  $\varphi2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi1 \cup$  fv  $\varphi2$ ] show ?case
  proof (cases sp)
    case (SSince sp' sps)
    from check_fv_cong(1)[of  $\varphi2$  _ _ 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  $\varphi1 \cup$  fv  $\varphi2$ ]]
  proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] ball_cong[of set sps, OF refl] refl, goal_cases  $\varphi2$   $\varphi1$ )
    case  $\varphi2$ 
    then show ?case
    by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
  next
    case ( $\varphi1$  sp)
    then show ?case using check_fv_cong(1)[of  $\varphi1$  _ _ 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  $\varphi1 \cup$  fv  $\varphi2$ ] show ?case
  proof (cases vp)
    case (VSince i vp' vps)
    from check_fv_cong(2)[of  $\varphi1$  _ _ 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  $\varphi1 \cup$  fv  $\varphi2$ ]]
  proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] ball_cong[of set vps, OF refl] refl, goal_cases  $\varphi1$   $\varphi2$ )

```

```

      case  $\varphi 1$ 
      then show ?case
      by (metis (mono_tags, lifting) 2 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    next
      case ( $\varphi 2$  vp)
      then show ?case using check_fv_cong(2)[of  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
    proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] ball_cong[of set vps, OF refl] refl, goal_cases  $\varphi 2$ )
      case ( $\varphi 2$  vp)
      then show ?case using check_fv_cong(2)[of  $\varphi 2$  _ _ 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  $\varphi 1$  I  $\varphi 2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
  proof (cases sp)
    case (SUntil sps sp')
    from check_fv_cong(1)[of  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
    proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] ball_cong[of set sps, OF refl] refl, goal_cases  $\varphi 2$   $\varphi 1$ )
      case  $\varphi 2$ 
      then show ?case
      by (metis (mono_tags, lifting) 1 IntE Un_upper2 compatible_vals_extensible compatible_vals_union_eq)
    next
      case ( $\varphi 1$  sp)
      then show ?case using check_fv_cong(1)[of  $\varphi 1$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ] show ?case
  proof (cases vp)
    case (VUntil i vps vp')
    from check_fv_cong(2)[of  $\varphi 1$  _ _ 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  $\varphi 1 \cup$  fv  $\varphi 2$ ]]
    proof (intro arg_cong2[of _ _ _ _ ( $\wedge$ )] ball_cong[of set vps, OF refl] refl, goal_cases  $\varphi 1$   $\varphi 2$ )
      case  $\varphi 1$ 
      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]: $s_check\ v\ \varphi\ sp = s_check_exec\ (\lambda x. \{v\ x\})\ \varphi\ sp$
by (subst check_exec_check)
(auto simp: compatible_vals_def elim: check_fv_cong[THEN iffD2, rotated])

lemma *v_check_code*[code]: $v_check\ v\ \varphi\ vp = v_check_exec\ (\lambda x. \{v\ x\})\ \varphi\ vp$
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  $\Rightarrow$  nat  $\Rightarrow$  nat option where
  LRTP  $\top$  i = Some i
| LRTP  $\perp$  i = Some i
| LRTP ( $\_ \dagger \_$ ) i = Some i
| LRTP ( $\_ \approx \_$ ) i = Some i
| LRTP ( $\neg_F \varphi$ ) i = LRTP  $\varphi$  i
| LRTP ( $\varphi \vee_F \psi$ ) i = max_opt (LRTP  $\varphi$  i) (LRTP  $\psi$  i)
| LRTP ( $\varphi \wedge_F \psi$ ) i = max_opt (LRTP  $\varphi$  i) (LRTP  $\psi$  i)
| LRTP ( $\varphi \rightarrow_F \psi$ ) i = max_opt (LRTP  $\varphi$  i) (LRTP  $\psi$  i)
| LRTP ( $\varphi \leftarrow_F \psi$ ) i = max_opt (LRTP  $\varphi$  i) (LRTP  $\psi$  i)
| LRTP ( $\exists_F \_ . \varphi$ ) i = LRTP  $\varphi$  i
| LRTP ( $\forall_F \_ . \varphi$ ) i = LRTP  $\varphi$  i
| LRTP (Y I  $\varphi$ ) i = LRTP  $\varphi$  (i-1)
| LRTP (X I  $\varphi$ ) i = LRTP  $\varphi$  (i+1)
| LRTP (P I  $\varphi$ ) i = LRTP  $\varphi$  (LTP_p_safe  $\sigma$  i I)
| LRTP (H I  $\varphi$ ) i = LRTP  $\varphi$  (LTP_p_safe  $\sigma$  i I)
| LRTP (F I  $\varphi$ ) i = (case right I of  $\infty \Rightarrow$  None | enat b  $\Rightarrow$  LRTP  $\varphi$  (LTP_f  $\sigma$  i b))
| LRTP (G I  $\varphi$ ) i = (case right I of  $\infty \Rightarrow$  None | enat b  $\Rightarrow$  LRTP  $\varphi$  (LTP_f  $\sigma$  i b))
| LRTP ( $\varphi$  S I  $\psi$ ) i = max_opt (LRTP  $\varphi$  i) (LRTP  $\psi$  (LTP_p_safe  $\sigma$  i I))
| LRTP ( $\varphi$  U I  $\psi$ ) i = (case right I of  $\infty \Rightarrow$  None | enat b  $\Rightarrow$  max_opt (LRTP  $\varphi$  ((LTP_f  $\sigma$  i b)-1))
(LRTP  $\psi$  (LTP_f  $\sigma$  i b)))

```

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

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

lemma *enat_trans[simp]*: $\text{enat } i \leq \text{enat } j \wedge \text{enat } j \leq \text{enat } k \implies \text{enat } i \leq \text{enat } k$
by *auto*

8.4 Active Domain

definition *AD* :: ('n, 'd) *formula* \Rightarrow *nat* \Rightarrow 'd *set*
where $AD \ \varphi \ i = \text{consts } \varphi \cup (\bigcup k \leq \text{the } (LRTP \ \varphi \ i). \bigcup (\text{set } ' \text{snd } ' \Gamma \ \sigma \ k))$

lemma *val_in_AD_iff*:
 $x \in \text{fv } \varphi \implies v \ x \in AD \ \varphi \ i \longleftrightarrow v \ x \in \text{consts } \varphi \vee$
 $(\exists r \ ts \ k. k \leq \text{the } (LRTP \ \varphi \ i) \wedge (r, v[[ts]]) \in \Gamma \ \sigma \ k \wedge x \in \bigcup (\text{set } (\text{map } \text{fv_trm } ts)))$
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 _ _ $\lambda x. _ \vee x]$ 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 ($\lambda d. \text{if } v \ x = d \text{ then } (v \ x) \text{ else } c \ d)$ 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*:
 $x \in \text{fv } \varphi \implies v \ x \notin AD \ \varphi \ i \longleftrightarrow v \ x \notin \text{consts } \varphi \wedge$
 $(\forall r \ ts \ k. k \leq \text{the } (LRTP \ \varphi \ i) \wedge x \in \bigcup (\text{set } (\text{map } \text{fv_trm } ts)) \longrightarrow (r, v[[ts]]) \notin \Gamma \ \sigma \ k)$
using *val_in_AD_iff* **by** *blast*

lemma *finite_values*: *finite* $(\bigcup (\text{set } ' \text{snd } ' \Gamma \ \sigma \ k))$
by (*transfer, auto simp add: sfinite_def*)

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

lemma *finite_AD [simp]*: *future_bounded* $\varphi \implies \text{finite } (AD \ \varphi \ i)$
using *finite_tps finite_values*
by (*simp add: AD_def enat_def*)

lemma *finite_AD_UNIV*:
assumes *future_bounded* φ **and** $AD \ \varphi \ i = (UNIV:: 'd \ \text{set})$
shows *finite* $(UNIV:: 'd \ \text{set})$
proof –

```

have finite (AD  $\varphi$   $i$ )
  using finite_AD[of  $\varphi$   $i$ , 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 ( $\neg_F \varphi$ )  $i$  = AD  $\varphi$   $i$ 
future_bounded ( $\varphi \vee_F \psi$ )  $\implies$  AD ( $\varphi \vee_F \psi$ )  $i$  = AD  $\varphi$   $i$   $\cup$  AD  $\psi$   $i$ 
future_bounded ( $\varphi \wedge_F \psi$ )  $\implies$  AD ( $\varphi \wedge_F \psi$ )  $i$  = AD  $\varphi$   $i$   $\cup$  AD  $\psi$   $i$ 
future_bounded ( $\varphi \longrightarrow_F \psi$ )  $\implies$  AD ( $\varphi \longrightarrow_F \psi$ )  $i$  = AD  $\varphi$   $i$   $\cup$  AD  $\psi$   $i$ 
future_bounded ( $\varphi \longleftrightarrow_F \psi$ )  $\implies$  AD ( $\varphi \longleftrightarrow_F \psi$ )  $i$  = AD  $\varphi$   $i$   $\cup$  AD  $\psi$   $i$ 
AD ( $\exists_F x. \varphi$ )  $i$  = AD  $\varphi$   $i$ 
AD ( $\forall_F x. \varphi$ )  $i$  = AD  $\varphi$   $i$ 
AD (Y  $I$   $\varphi$ )  $i$  = AD  $\varphi$  ( $i - 1$ )
AD (X  $I$   $\varphi$ )  $i$  = AD  $\varphi$  ( $i + 1$ )
future_bounded (F  $I$   $\varphi$ )  $\implies$  AD (F  $I$   $\varphi$ )  $i$  = AD  $\varphi$  (LTP_f  $\sigma$   $i$  (the_enat (right  $I$ )))
future_bounded (G  $I$   $\varphi$ )  $\implies$  AD (G  $I$   $\varphi$ )  $i$  = AD  $\varphi$  (LTP_f  $\sigma$   $i$  (the_enat (right  $I$ )))
AD (P  $I$   $\varphi$ )  $i$  = AD  $\varphi$  (LTP_p_safe  $\sigma$   $i$   $I$ )
AD (H  $I$   $\varphi$ )  $i$  = AD  $\varphi$  (LTP_p_safe  $\sigma$   $i$   $I$ )
future_bounded ( $\varphi$  S  $I$   $\psi$ )  $\implies$  AD ( $\varphi$  S  $I$   $\psi$ )  $i$  = AD  $\varphi$   $i$   $\cup$  AD  $\psi$  (LTP_p_safe  $\sigma$   $i$   $I$ )
future_bounded ( $\varphi$  U  $I$   $\psi$ )  $\implies$  AD ( $\varphi$  U  $I$   $\psi$ )  $i$  = AD  $\varphi$  (LTP_f  $\sigma$   $i$  (the_enat (right  $I$ )) - 1)  $\cup$  AD
 $\psi$  (LTP_f  $\sigma$   $i$  (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: $i \leq j \implies$ LTP_p_safe σ i $I \leq$ LTP_p_safe σ j I

```

unfolding LTP_p_safe_def
by (smt (verit, ccfv_threshold)  $\tau$ _mono bot_nat_0.extremum diff_le_mono order.trans  $i$ _LTP_tau
le_cases3 min.bounded_iff)

```

lemma LTP_f_mono:

```

assumes  $i \leq j$ 
shows LTP_f  $\sigma$   $i$   $b \leq$  LTP_f  $\sigma$   $j$   $b$ 
unfolding LTP_def
proof (rule Max_mono)
show finite { $i. \tau \sigma i \leq \tau \sigma j + b$ }
  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 $\varphi \implies i \leq j \implies$ the (LRTP φ i) \leq the (LRTP φ j)

```

proof (induct  $\varphi$  arbitrary:  $i$   $j$ )
case (Or  $\varphi$ 1  $\varphi$ 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  $\varphi$ 1  $\varphi$ 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  $\varphi$ 1  $\varphi$ 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  $\varphi 1 \varphi 2$ )
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  $\varphi 1 I \varphi 2$ )
from Since(1)[OF _ Since(4)] Since(2)[of LTP_p_safe  $\sigma i I$  LTP_p_safe  $\sigma j I$ ] Since(3-)
show ?case
  by (auto simp: max_opt_def not_none_fb_LRTP LTP_p_mono split: option.splits)
next
case (Until  $\varphi 1 I \varphi 2$ )
from Until(1)[of LTP_f  $\sigma i$  (the_enat (right I)) - 1 LTP_f  $\sigma j$  (the_enat (right I)) - 1]
  Until(2)[of LTP_f  $\sigma i$  (the_enat (right I)) LTP_f  $\sigma j$  (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 $\varphi \implies i \leq j \implies AD \varphi i \subseteq AD \varphi j$
by (auto 0 3 simp: AD_def Bex_def intro: LRTP_mono elim!: order_trans)

lemma LTP_p_safe_le[simp]: LTP_p_safe $\sigma i I \leq i$
by (auto simp: LTP_p_safe_def)

lemma check_AD_cong:

```

assumes future_bounded  $\varphi$ 
  and ( $\forall x \in fv \varphi. v x = v' x \vee (v x \notin AD \varphi i \wedge v' x \notin AD \varphi i)$ )
shows ( $s\_at\ sp = i \implies s\_check\ v\ \varphi\ sp \longleftrightarrow s\_check\ v'\ \varphi\ sp$ )
  ( $v\_at\ vp = i \implies v\_check\ v\ \varphi\ vp \longleftrightarrow v\_check\ v'\ \varphi\ vp$ )
using assms

```

proof (induction $v \varphi sp$ **and** $v \varphi vp$ 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 (SOlR  $sp'$ )
then show ?thesis
  using IH(2)[of _ _ _ _  $v'$ ] hyps
  by (cases f) auto

```

```

next
case (SOlR  $sp'$ )
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  $\varphi$ )
{ fix k
  assume k:  $k \leq i \tau \sigma i - \text{left } I \geq \tau \sigma k$ 
  then have  $\tau \sigma i - \text{left } I \geq \tau \sigma 0$ 
    by (meson  $\tau\_mono$   $le0$   $order\_trans$ )
  with k have  $k \leq LTP\_p\_safe \sigma i I$ 
  unfolding LTP_p_safe_def by (auto simp: i_LTP_tau)
}
}

```



```

    with Once hyps(2,3) have  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi k \wedge v' x \notin AD \varphi k$ 
      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  $\varphi$ )
  { fix sp :: ('n, 'd) sproof
    define l and u where l = s_at sp and u = LTP_p  $\sigma$  i I
    assume *: sp  $\in$  set sps  $\tau$   $\sigma$  0 + left I  $\leq$   $\tau$   $\sigma$  i
    then have u_def: u = LTP_p_safe  $\sigma$  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= $\lambda xs. nth xs j$ , OF eq] j len *(2)
      by (auto simp: nth_append)
    ultimately have l  $\leq$  u
      unfolding l_def by auto
    with Historically hyps(2,3) have  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi l \wedge v' x \notin AD \varphi l$ 
      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  $\varphi$ )
  { fix k
    assume  $\tau$   $\sigma$  k  $\leq$  the_enat (right I) +  $\tau$   $\sigma$  i
    then have k  $\leq$  LTP_f  $\sigma$  i (the_enat (right I))
      by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
    with Eventually hyps(2,3) have  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi k \wedge v' x \notin AD \varphi k$ 
      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  $\varphi$ )
  { fix sp :: ('n, 'd) sproof

```

```

define  $l$  and  $u$  where  $l = s\_at\ sp$  and  $u = LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I))$ 
assume  $*$ :  $sp \in set\ sps$ 
then obtain  $j$  where  $j: sp = sps ! j\ j < length\ sps$ 
  unfolding  $in\_set\_conv\_nth$  by  $auto$ 
assume  $eq$ :  $map\ s\_at\ sps = [ETP\_f\ \sigma\ i\ I ..< Suc\ u]$ 
then have  $length\ sps = Suc\ u - ETP\_f\ \sigma\ i\ I$ 
  by  $(auto\ dest!: arg\_cong[where\ f=length])$ 
with  $j\ eq$  have  $l \leq LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I))$ 
  by  $(auto\ simp: l\_def\ u\_def\ dest!: arg\_cong[where\ f=\lambda xs. nth\ xs\ j]$ 
     $simp\ del: upt.simps\ split: if\_splits)$ 
with  $Always\ hyps(2,3)$  have  $\forall x \in fv\ \varphi. v\ x = v'\ x \vee v\ x \notin AD\ \varphi\ l \wedge v'\ x \notin AD\ \varphi\ l$ 
  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  $(Since\ \varphi\ I\ \psi)$ 
  { fix  $sp :: ('n, 'd)\ sproof$ 
    define  $l$  where  $l = s\_at\ sp$ 
    assume  $*$ :  $sp \in set\ sps$ 
    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  $arg\_cong[where\ f=\lambda xs. nth\ xs\ j, OF\ eq]\ j\ len$ 
      by  $(auto\ simp: nth\_append)$ 
    ultimately have  $l \leq i$ 
      unfolding  $l\_def$  by  $auto$ 
    with  $Since\ hyps(2,3)$  have  $\forall x \in fv\ \varphi. v\ x = v'\ x \vee v\ x \notin AD\ \varphi\ l \wedge v'\ x \notin AD\ \varphi\ l$ 
      by  $(auto\ simp: dest!: bspec\ dest: AD\_mono[THEN\ set\_mp, rotated\ -1])$ 
  }
  moreover
  { fix  $k$ 
    assume  $k: k \leq i\ \tau\ \sigma\ i - left\ I \geq \tau\ \sigma\ k$ 
    then have  $\tau\ \sigma\ i - left\ I \geq \tau\ \sigma\ 0$ 
      by  $(meson\ \tau\_mono\ le0\ order\_trans)$ 
    with  $k$  have  $k \leq LTP\_p\_safe\ \sigma\ i\ I$ 
      unfolding  $LTP\_p\_safe\_def$  by  $(auto\ simp: i\_LTP\_tau)$ 
    with  $Since\ hyps(2,3)$  have  $\forall x \in fv\ \psi. v\ x = v'\ x \vee v\ x \notin AD\ \psi\ k \wedge v'\ x \notin AD\ \psi\ k$ 
      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  $\varphi$  I  $\psi$ )
  { fix sp :: ('n, 'd) sproof
    define l where l = s_at sp
    assume *: sp  $\in$  set sps
    from *(1) obtain j where j: sp = sps ! j j < length sps
      unfolding in_set_conv_nth by auto
    moreover
    assume  $\delta$   $\sigma$  (s_at sp') i  $\leq$  the_enat (right I)
    then have s_at sp'  $\leq$  LTP_f  $\sigma$  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= $\lambda$ xs. nth xs j, OF eq] j len
      by (auto simp: nth_append)
    ultimately have l  $\leq$  LTP_f  $\sigma$  i (the_enat (right I)) - 1
      unfolding l_def by auto
    with Until_hyps(2,3) have  $\forall x \in \text{fv } \varphi. v x = v' x \vee v x \notin \text{AD } \varphi l \wedge v' x \notin \text{AD } \varphi l$ 
      by (auto simp: dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
  }
  moreover
  { fix k
    assume  $\tau$   $\sigma$  k  $\leq$  the_enat (right I) +  $\tau$   $\sigma$  i
    then have k  $\leq$  LTP_f  $\sigma$  i (the_enat (right I))
      by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
    with Until_hyps(2,3) have  $\forall x \in \text{fv } \psi. v x = v' x \vee v x \notin \text{AD } \psi k \wedge v' x \notin \text{AD } \psi k$ 
      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 $\wedge$ )
  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  $\varphi$ )
  { fix vp :: ('n, 'd) vproof

```

```

define  $l$  and  $u$  where  $l = v\_at\ vp$  and  $u = LTP\_p\ \sigma\ i\ I$ 
assume  $*$ :  $vp \in set\ vps\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$ 
then have  $u\_def$ :  $u = LTP\_p\_safe\ \sigma\ i\ I$ 
  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  $arg\_cong[where\ f=\lambda xs. nth\ xs\ j, OF\ eq]\ j\ len\ *(2)$ 
  by (auto simp:  $nth\_append$ )
ultimately have  $l \leq u$ 
  unfolding  $l\_def$  by auto
with Once  $hypos(2,3)$  have  $\forall x \in fv\ \varphi. v\ x = v'\ x \vee v\ x \notin AD\ \varphi\ l \wedge v'\ x \notin AD\ \varphi\ l$ 
  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 ( $Historically\ I\ \varphi$ )
  { fix  $k$ 
    assume  $k$ :  $k \leq i\ \tau\ \sigma\ i - left\ I \geq \tau\ \sigma\ k$ 
    then have  $\tau\ \sigma\ i - left\ I \geq \tau\ \sigma\ 0$ 
      by (meson  $\tau\_mono\ le0\ order\_trans$ )
    with  $k$  have  $k \leq LTP\_p\_safe\ \sigma\ i\ I$ 
      unfolding  $LTP\_p\_safe\_def$  by (auto simp:  $i\_LTP\_tau$ )
    with  $Historically\ hyps(2,3)$  have  $\forall x \in fv\ \varphi. v\ x = v'\ x \vee v\ x \notin AD\ \varphi\ k \wedge v'\ x \notin AD\ \varphi\ k$ 
      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 ( $Eventually\ I\ \varphi$ )
  { fix  $vp :: ('n, 'd)\ vproof$ 
    define  $l$  and  $u$  where  $l = v\_at\ vp$  and  $u = LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I))$ 
    assume  $*$ :  $vp \in set\ vps$ 
    then obtain  $j$  where  $j$ :  $vp = vps\ !\ j\ j < length\ vps$ 
      unfolding  $in\_set\_conv\_nth$  by auto
    assume  $eq$ :  $map\ v\_at\ vps = [ETP\_f\ \sigma\ i\ I\ ..<\ Suc\ u]$ 
    then have  $length\ vps = Suc\ u - ETP\_f\ \sigma\ i\ I$ 
      by (auto dest!:  $arg\_cong[where\ f=length]$ )
    with  $j\ eq$  have  $l \leq LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I))$ 
      by (auto simp:  $l\_def\ u\_def\ dest!$ :  $arg\_cong[where\ f=\lambda xs. nth\ xs\ j]$ 
         $simp\ del: upt.simps\ split: if\_splits$ )
    with  $Eventually\ hyps(2,3)$  have  $\forall x \in fv\ \varphi. v\ x = v'\ x \vee v\ x \notin AD\ \varphi\ l \wedge v'\ x \notin AD\ \varphi\ l$ 
  }

```

```

    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  $\varphi$ )
  { fix k
    assume  $\tau \sigma k \leq \text{the\_enat (right I)} + \tau \sigma i$ 
    then have  $k \leq \text{LTP\_f } \sigma i (\text{the\_enat (right I)})$ 
      by (metis add.commute i_le_LTPi_add le_add_diff_inverse)
    with Always hyps(2,3) have  $\forall x \in \text{fv } \varphi. v x = v' x \vee v x \notin \text{AD } \varphi k \wedge v' x \notin \text{AD } \varphi k$ 
      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  $\varphi$  I  $\psi$ )
  { fix sp :: ('n, 'd) vproof
    define l and u where  $l = v\_at \text{ sp and } u = \text{LTP\_p } \sigma i I$ 
    assume *:  $sp \in \text{set vps } \tau \sigma 0 + \text{left I} \leq \tau \sigma i$ 
    then have u_def:  $u = \text{LTP\_p\_safe } \sigma i I$ 
      by (auto simp: LTP_p_safe_def u_def)
    from *(1) obtain j where  $j: sp = \text{vps ! } j \ j < \text{length vps}$ 
      unfolding in_set_conv_nth by auto
    moreover
    assume eq:  $\text{map } v\_at \text{ vps} = [v\_at \text{ vp' ..} < \text{Suc } u]$ 
    then have len:  $\text{length vps} = \text{Suc } u - v\_at \text{ vp'}$ 
      by (auto dest!: arg_cong[where f=length])
    moreover
    have  $v\_at (\text{vps ! } j) = v\_at \text{ vp' } + j$ 
      using arg_cong[where f= $\lambda x s. \text{nth } xs \ j$ , OF eq] j len
      by (auto simp: nth_append)
    ultimately have  $l \leq u$ 
      unfolding l_def by auto
    with Since hyps(2,3) have  $\forall x \in \text{fv } \psi. v x = v' x \vee v x \notin \text{AD } \psi l \wedge v' x \notin \text{AD } \psi l$ 
      by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
  }
  moreover
  { fix k
    assume  $k: k \leq i$ 
    with Since hyps(2,3) have  $\forall x \in \text{fv } \varphi. v x = v' x \vee v x \notin \text{AD } \varphi k \wedge v' x \notin \text{AD } \varphi k$ 
      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  $\varphi$  I  $\psi$ )
{ fix vp :: ('n, 'd) vproof
define l and u where l = v_at vp and u = LTP_p  $\sigma$  i I
assume *: vp  $\in$  set vps  $\tau$   $\sigma$  0 + left I  $\leq$   $\tau$   $\sigma$  i
then have u_def: u = LTP_p_safe  $\sigma$  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= $\lambda$ xs. nth xs j, OF eq] j len *(2)
by (auto simp: nth_append)
ultimately have l  $\leq$  u
unfolding l_def by auto
with Since hyps(2,3) have  $\forall x \in fv \psi. v x = v' x \vee v x \notin AD \psi l \wedge v' x \notin AD \psi l$ 
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  $\varphi$  I  $\psi$ )
{ fix sp :: ('n, 'd) vproof
define l and u where l = v_at sp and u = v_at vp'
assume *: sp  $\in$  set vps v_at vp'  $\leq$  LTP_f  $\sigma$  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  $\sigma$  i I ..< Suc u]
then have length vps = Suc u - ETP_f  $\sigma$  i I
by (auto dest!: arg_cong[where f=length])
with j eq *(2) have l  $\leq$  LTP_f  $\sigma$  i (the_enat (right I))
by (auto simp: l_def u_def dest!: arg_cong[where f= $\lambda$ xs. nth xs j]
simp del: upt.simps split: if_splits)
with Until hyps(2,3) have  $\forall x \in fv \psi. v x = v' x \vee v x \notin AD \psi l \wedge v' x \notin AD \psi l$ 
by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
}
moreover
{ fix k
assume k < LTP_f  $\sigma$  i (the_enat (right I))
then have k  $\leq$  LTP_f  $\sigma$  i (the_enat (right I)) - 1
by linarith
with Until hyps(2,3) have  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi k \wedge v' x \notin AD \varphi k$ 
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  $\varphi$  I  $\psi$ )
{ fix vp :: ('n, 'd) vproof
  define l and u where l = v_at vp and u = LTP_f  $\sigma$  i (the_enat (right I))
  assume *: vp  $\in$  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  $\sigma$  i I ..< Suc u]
  then have length vps = Suc u - ETP_f  $\sigma$  i I
    by (auto dest!: arg_cong[where f=length])
  with j eq have l  $\leq$  LTP_f  $\sigma$  i (the_enat (right I))
    by (auto simp: l_def u_def dest!: arg_cong[where f= $\lambda$ xs. nth xs j]
      simp del: upt.simps split: if_splits)
  with Until hyps(2,3) have  $\forall x \in fv \psi. v x = v' x \vee v x \notin AD \psi l \wedge v' x \notin AD \psi l$ 
    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: $distinct\ xs \implies part_hd\ (tabulate\ xs\ f\ z) = (case\ xs\ of\ [] \Rightarrow z \mid (x\ \# _) \Rightarrow (if\ set\ xs = UNIV\ then\ f\ x\ else\ z))$
 by (transfer, auto split: list.splits)

lemma s_at_tabulate:

```

assumes  $\forall z. s\_at\ (mypick\ z) = i$ 
and mypart = tabulate (sorted_list_of_set (AD  $\varphi$  i)) mypick (mypick (SOME z. z  $\notin$  AD  $\varphi$  i))
shows  $\forall (sub, vp) \in SubsVals\ mypart. s\_at\ vp = i$ 
using assms by (transfer, auto)

```

lemma v_at_tabulate:

```

assumes  $\forall z. v\_at\ (mypick\ z) = i$ 
and mypart = tabulate (sorted_list_of_set (AD  $\varphi$  i)) mypick (mypick (SOME z. z  $\notin$  AD  $\varphi$  i))
shows  $\forall (sub, vp) \in SubsVals\ mypart. v\_at\ vp = i$ 
using assms by (transfer, auto)

```

lemma s_check_tabulate:

```

assumes future_bounded  $\varphi$ 
and  $\forall z. s\_at\ (mypick\ z) = i$ 
and  $\forall z. s\_check\ (v(x:=z))\ \varphi\ (mypick\ z)$ 
and mypart = tabulate (sorted_list_of_set (AD  $\varphi$  i)) mypick (mypick (SOME z. z  $\notin$  AD  $\varphi$  i))
shows  $\forall (sub, vp) \in SubsVals\ mypart. \forall z \in sub. s\_check\ (v(x := z))\ \varphi\ vp$ 
using assms

```

```

proof (transfer fixing:  $\sigma\ \varphi\ mypick\ i\ v\ x, goal\_cases\ 1$ )
case (1 mypart)
{ fix z

```



```

assume  $s\_at\_assm$ :  $\forall z. s\_at (mypick\ z) = i$ 
and  $s\_check\_assm$ :  $\forall z. s\_check (v(x := z))\ \varphi (mypick\ z)$ 
and  $fb\_assm$ : future_bounded  $\varphi$ 
and  $z\_notin\_AD$ :  $z \notin (AD\ \varphi\ i)$ 
have  $s\_at\_mypick$ :  $s\_at (mypick (SOME\ z. z \notin local.AD\ \varphi\ i)) = i$ 
using  $s\_at\_assm$  by simp
have  $s\_check\_mypick$ :  $Checker.s\_check\ \sigma (v(x := SOME\ z. z \notin AD\ \varphi\ i))\ \varphi (mypick (SOME\ z. z \notin AD\ \varphi\ i))$ 
using  $s\_check\_assm$  by simp
have  $s\_check (v(x := z))\ \varphi (mypick (SOME\ z. z \notin AD\ \varphi\ i))$ 
using  $z\_notin\_AD$ 
by (subst check_AD_cong(1)[of  $\varphi\ v(x := z)\ v(x := (SOME\ z. z \notin Checker.AD\ \sigma\ \varphi\ i))\ i\ mypick (SOME\ z. z \notin AD\ \varphi\ i)$ , OF  $fb\_assm\ \_ s\_at\_mypick$ ])
  (auto simp add: someI[of  $\lambda z. z \notin AD\ \varphi\ i\ z$ ]  $s\_check\_mypick\ fb\_assm\ split: if\_splits$ ])
}
with 1 show ?case
by auto
qed

```

```

lemma  $v\_check\_tabulate$ :
assumes future_bounded  $\varphi$ 
and  $\forall z. v\_at (mypick\ z) = i$ 
and  $\forall z. v\_check (v(x:=z))\ \varphi (mypick\ z)$ 
and  $mypart = tabulate (sorted\_list\_of\_set (AD\ \varphi\ i))\ mypick (mypick (SOME\ z. z \notin AD\ \varphi\ i))$ 
shows  $\forall (sub, vp) \in SubsVals\ mypart. \forall z \in sub. v\_check (v(x := z))\ \varphi\ vp$ 
using assms
proof (transfer fixing:  $\sigma\ \varphi\ mypick\ i\ v\ x, goal\_cases\ 1$ )
case ( $1\ mypart$ )
{ fix  $z$ 
assume  $v\_at\_assm$ :  $\forall z. v\_at (mypick\ z) = i$ 
and  $v\_check\_assm$ :  $\forall z. v\_check (v(x := z))\ \varphi (mypick\ z)$ 
and  $fb\_assm$ : future_bounded  $\varphi$ 
and  $z\_notin\_AD$ :  $z \notin (AD\ \varphi\ i)$ 
have  $v\_at\_mypick$ :  $v\_at (mypick (SOME\ z. z \notin local.AD\ \varphi\ i)) = i$ 
using  $v\_at\_assm$  by simp
have  $v\_check\_mypick$ :  $Checker.v\_check\ \sigma (v(x := SOME\ z. z \notin AD\ \varphi\ i))\ \varphi (mypick (SOME\ z. z \notin AD\ \varphi\ i))$ 
using  $v\_check\_assm$  by simp
have  $v\_check (v(x := z))\ \varphi (mypick (SOME\ z. z \notin AD\ \varphi\ i))$ 
using  $z\_notin\_AD$ 
by (subst check_AD_cong(2)[of  $\varphi\ v(x := z)\ v(x := (SOME\ z. z \notin Checker.AD\ \sigma\ \varphi\ i))\ i\ mypick (SOME\ z. z \notin AD\ \varphi\ i)$ , OF  $fb\_assm\ \_ v\_at\_mypick$ ])
  (auto simp add: someI[of  $\lambda z. z \notin AD\ \varphi\ i\ z$ ]  $v\_check\_mypick\ fb\_assm\ split: if\_splits$ ])
}
with 1 show ?case
by auto
qed

```

```

lemma  $s\_at\_part\_hd\_tabulate$ :
assumes future_bounded  $\varphi$ 
and  $\forall z. s\_at (f\ z) = i$ 
and  $mypart = tabulate (sorted\_list\_of\_set (AD\ \varphi\ i))\ f (f (SOME\ z. z \notin AD\ \varphi\ i))$ 
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  $\varphi$ 
and  $\forall z. v\_at (f\ z) = i$ 

```

```

and mypart = tabulate (sorted_list_of_set (AD  $\varphi$  i)) f (f (SOME z. z  $\notin$  AD  $\varphi$  i))
shows v_at (part_hd mypart) = i
using assms by (simp add: part_hd_tabulate split: list.splits)

lemma check_completeness_aux:
  (SAT  $\sigma$  v i  $\varphi \longrightarrow$  future_bounded  $\varphi \longrightarrow$  ( $\exists$  sp. s_at sp = i  $\wedge$  s_check v  $\varphi$  sp))  $\wedge$ 
  (VIO  $\sigma$  v i  $\varphi \longrightarrow$  future_bounded  $\varphi \longrightarrow$  ( $\exists$  vp. v_at vp = i  $\wedge$  v_check v  $\varphi$  vp))
proof (induct v i  $\varphi$  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  $\varphi$ )
then show ?case
  by (auto intro: exI[of _ SNeg _])
next
case (VNeg v i  $\varphi$ )
then show ?case
  by (auto intro: exI[of _ VNeg _])
next
case (SOrL v i  $\varphi$   $\psi$ )
then show ?case
  by (auto intro: exI[of _ SOrL _])
next
case (SOrR v i  $\psi$   $\varphi$ )
then show ?case
  by (auto intro: exI[of _ SOrR _])
next
case (VOr v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ VOr _])
next
case (SAnd v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ SAnd _])
next
case (VAndL v i  $\varphi$   $\psi$ )
then show ?case

```

```

    by (auto intro: exI[of _ VAndL _])
next
case (VAndR v i  $\psi$   $\varphi$ )
then show ?case
  by (auto intro: exI[of _ VAndR _])
next
case (SImpL v i  $\varphi$   $\psi$ )
then show ?case
  by (auto intro: exI[of _ SImpL _])
next
case (SImpR v i  $\psi$   $\varphi$ )
then show ?case
  by (auto intro: exI[of _ SImpR _])
next
case (VImp v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ VImp _ _])
next
case (SIffSS v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ SIffSS _ _])
next
case (SIffVV v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ SIffVV _ _])
next
case (VIffSV v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ VIffSV _ _])
next
case (VIffVS v i  $\varphi$   $\psi$ )
then show ?case
  by (auto 0 3 intro: exI[of _ VIffVS _ _])
next
case (SExists v x i  $\varphi$ )
then show ?case
  by (auto 0 3 simp: fun_upd_def intro: exI[of _ SExists x _ _])
next
case (VExists v x i  $\varphi$ )
show ?case
proof
  assume future_bounded ( $\exists_F x. \varphi$ )
  then have fb: future_bounded  $\varphi$ 
    by simp
  obtain mypick where mypick_def:  $v\_at (mypick z) = i \wedge v\_check (v(x:=z)) \varphi (mypick z)$  for z
    using VExists fb by metis
  define mypart where mypart =  $tabulate (sorted\_list\_of\_set (AD \varphi i)) mypick (mypick (SOME z. z \notin (AD \varphi i)))$ 
  have mypick_at:  $\forall z. v\_at (mypick z) = i$ 
    by (simp add: mypick_def)
  have mypick_v_check:  $\forall z. v\_check (v(x:=z)) \varphi (mypick z)$ 
    by (simp add: mypick_def)
  have mypick_v_check2:  $\forall z. v\_check (v(x := (SOME z. z \notin AD \varphi i))) \varphi (mypick (SOME z. z \notin AD \varphi i))$ 
    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 ( $\exists_{Fx}. \varphi$ ) (VExists x mypart)
  using v_at_tabulate[of mypick i _  $\varphi$ , 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  $\exists vp. v\_at\ vp = i \wedge v\_check\ v\ (\exists_{Fx}. \varphi)\ vp$ 
  using v_at_myp v_check_myp by blast
qed
next
case (SForall v x i  $\varphi$ )
show ?case
proof
  assume future_bounded ( $\forall_{Fx}. \varphi$ )
  then have fb: future_bounded  $\varphi$ 
    by simp
  obtain mypick where mypick_def: s_at (mypick z) = i  $\wedge$  s_check (v(x:=z))  $\varphi$  (mypick z) for z
    using SForall fb by metis
  define mypart where mypart = tabulate (sorted_list_of_set (AD  $\varphi$  i)) mypick (mypick (SOME z. z
 $\notin$  (AD  $\varphi$  i)))
  have mypick_at:  $\forall z. s\_at\ (mypick\ z) = i$ 
    by (simp add: mypick_def)
  have mypick_s_check:  $\forall z. s\_check\ (v(x:=z))\ \varphi\ (mypick\ z)$ 
    by (simp add: mypick_def)
  have mypick_s_check2:  $\forall z. s\_check\ (v(x := (SOME\ z.\ z \notin AD\ \varphi\ i)))\ \varphi\ (mypick\ (SOME\ z.\ z \notin AD\ \varphi\ i))$ 
    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 ( $\forall_{Fx}. \varphi$ ) (SForall x mypart)
    using s_at_tabulate[of mypick i _  $\varphi$ , 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  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (\forall_{Fx}. \varphi)\ sp$ 
    using s_at_myp s_check_myp by blast
qed
next
case (VForall v x i  $\varphi$ )
then show ?case
  by (auto 0 3 simp: fun_upd_def intro: exI[of _ VForall x _])
next
case (SPrev i I v  $\varphi$ )
then show ?case
  by (force intro: exI[of _ SPrev _])
next
case (VPrev i v  $\varphi$  I)
then show ?case
  by (force intro: exI[of _ VPrev _])
next
case (VPrevZ i v I  $\varphi$ )
then show ?case
  by (auto intro!: exI[of _ VPrevZ])
next
case (VPrevOutL i I v  $\varphi$ )
then show ?case
  by (auto intro!: exI[of _ VPrevOutL i])
next
case (VPrevOutR i I v  $\varphi$ )
then show ?case

```

```

    by (auto intro!: exI[of _ VPrevOutR i])
next
case (SNext i I v  $\varphi$ )
then show ?case
  by (force simp: Let_def intro: exI[of _ SNext _])
next
case (VNext v i  $\varphi$  I)
then show ?case
  by (force simp: Let_def intro: exI[of _ VNext _])
next
case (VNextOutL i I v  $\varphi$ )
then show ?case
  by (auto intro!: exI[of _ VNextOutL i])
next
case (VNextOutR i I v  $\varphi$ )
then show ?case
  by (auto intro!: exI[of _ VNextOutR i])
next
case (SOnce j i I v  $\varphi$ )
then show ?case
  by (auto simp: Let_def intro: exI[of _ SOnce i _])
next
case (VOnceOut i I v  $\varphi$ )
then show ?case
  by (auto intro!: exI[of _ VOnceOut i])
next
case (VOnce j I i v  $\varphi$ )
show ?case
proof
  assume future_bounded (P I  $\varphi$ )
  then have fb: future_bounded  $\varphi$ 
    by simp
  obtain mypick where mypick_def:  $\forall k \in \{j .. LTP\_p \sigma i I\}. v\_at (mypick k) = k \wedge v\_check v \varphi$ 
    (mypick k)
  using VOnce fb by metis
  then obtain vps where vps_def:  $map (v\_at) vps = [j ..< Suc (LTP\_p \sigma i I)] \wedge (\forall vp \in set vps. v\_check v \varphi vp)$ 
  by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([j ..< Suc (LTP\_p \sigma i I)])])
  then have  $v\_at (VOnce i j vps) = i \wedge v\_check v (P I \varphi) (VOnce i j vps)$ 
    using VOnce by auto
  then show  $\exists vp. v\_at vp = i \wedge v\_check v (P I \varphi) vp$ 
    by blast
qed
next
case (SEventually j i I v  $\varphi$ )
then show ?case
  by (auto simp: Let_def intro: exI[of _ SEventually i _])
next
case (VEventually I i v  $\varphi$ )
show ?case
proof
  assume fb_eventually: future_bounded (F I  $\varphi$ )
  then have fb: future_bounded  $\varphi$ 
    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:  $j = LTP \sigma (\tau \sigma i + b)$ 

```

```

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 \sigma 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 \varphi)
show ?case
proof
  assume fb_historically: future_bounded (H I \varphi)
  then have fb: future_bounded \varphi
  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 \sigma 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 \varphi)
then show ?case
  by (auto intro!: exI[of _ SHistoricallyOut i])
next
case (VHistorically j i I v \varphi)
then show ?case
  by (auto simp: Let_def intro: exI[of _ VHistorically i _])
next
case (SAlways I i v \varphi)
show ?case
proof
  assume fb_always: future_bounded (G I \varphi)
  then have fb: future_bounded \varphi
  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 \sigma i I ..< Suc j]])])

```

```

then have  $s\_at (SAlways\ i\ j\ sps) = i \wedge s\_check\ v\ (G\ I\ \varphi) (SAlways\ i\ j\ sps)$ 
  using  $SAlways\ b\_def\ j\_def$  by  $simp$ 
then show  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (G\ I\ \varphi)\ sp$ 
  by  $blast$ 
qed
next
case  $(VAlways\ j\ i\ I\ v\ \varphi)$ 
then show  $?case$ 
  by  $(auto\ simp: Let\_def\ intro: exI[of\_ VAlways\ i\_])$ 
next
case  $(SSince\ j\ i\ I\ v\ \psi\ \varphi)$ 
show  $?case$ 
proof
  assume  $fb\_since: future\_bounded\ (\varphi\ S\ I\ \psi)$ 
  then have  $fb: future\_bounded\ \varphi\ future\_bounded\ \psi$ 
    by  $simp\_all$ 
  obtain  $sp2$  where  $sp2\_def: s\_at\ sp2 = j \wedge s\_check\ v\ \psi\ sp2$ 
    using  $SSince\ fb\_since$  by  $auto$ 
  {
    assume  $Suc\ j > i$ 
    then have  $s\_at (SSince\ sp2\ []) = i \wedge s\_check\ v\ (\varphi\ S\ I\ \psi) (SSince\ sp2\ [])$ 
      using  $sp2\_def\ SSince$  by  $auto$ 
    then have  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (\varphi\ S\ I\ \psi)\ sp$ 
      by  $blast$ 
  }
  moreover
  {
    assume  $sucj\_leq\_i: Suc\ j \leq i$ 
    obtain  $mypick$  where  $mypick\_def: \forall k \in \{Suc\ j ..< Suc\ i\}. s\_at (mypick\ k) = k \wedge s\_check\ v\ \varphi$ 
       $(mypick\ k)$ 
    using  $SSince\ fb\_since$  by  $atomize\_elim\ (rule\ bchoice, simp)$ 
    then obtain  $sp1s$  where  $sp1s\_def: map\ (s\_at)\ sp1s = [Suc\ j ..< Suc\ i] \wedge (\forall sp \in set\ sp1s. s\_check$ 
       $v\ \varphi\ sp)$ 
    by  $atomize\_elim\ (auto\ intro!: trans[OF\ list.map\_cong\ list.map\_id]\ exI[of\_ map\ mypick\ ([Suc\ j$ 
       $..< Suc\ i]])$ 
    then have  $sp1s \neq []$ 
      using  $sucj\_leq\_i$  by  $auto$ 
    then have  $s\_at (SSince\ sp2\ sp1s) = i \wedge s\_check\ v\ (\varphi\ S\ I\ \psi) (SSince\ sp2\ sp1s)$ 
      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  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (\varphi\ S\ I\ \psi)\ sp$ 
      by  $blast$ 
  }
  ultimately show  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (\varphi\ S\ I\ \psi)\ sp$ 
    using  $not\_less$  by  $blast$ 
qed
next
case  $(VSinceOut\ i\ I\ v\ \varphi\ \psi)$ 
then show  $?case$ 
  by  $(auto\ intro!: exI[of\_ VSinceOut\ i])$ 
next
case  $(VSince\ I\ i\ j\ v\ \varphi\ \psi)$ 
show  $?case$ 
proof
  assume  $fb\_since: future\_bounded\ (\varphi\ S\ I\ \psi)$ 
  then have  $fb: future\_bounded\ \varphi\ future\_bounded\ \psi$ 

```

```

    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  $s\_at$  ( $SUntil$   $sp1s$   $sp2$ ) =  $i \wedge s\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ ) ( $SUntil$   $sp1s$   $sp2$ )
  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  $\exists sp. s\_at$   $sp = i \wedge s\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ )  $sp$ 
  by blast
}
ultimately show  $\exists sp. s\_at$   $sp = i \wedge s\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ )  $sp$ 
using not_less by blast
qed
next
case ( $VUntil$   $I$   $j$   $i$   $v$   $\varphi$   $\psi$ )
show ? $case$ 
proof
  assume  $fb\_until$ :  $future\_bounded$  ( $\varphi \mathbf{U} I \psi$ )
  then have  $fb$ :  $future\_bounded$   $\varphi$   $future\_bounded$   $\psi$ 
    by simp_all
  obtain  $vp1$  where  $vp1\_def$ :  $v\_at$   $vp1 = j \wedge v\_check$   $v$   $\varphi$   $vp1$ 
    using  $VUntil$   $fb\_until$  by auto
  obtain  $mypick$  where  $mypick\_def$ :  $\forall k \in \{ETP\_f \sigma i I .. j\}. v\_at$  ( $mypick$   $k$ ) =  $k \wedge v\_check$   $v$   $\psi$ 
    ( $mypick$   $k$ )
    using  $VUntil$   $fb\_until$  by  $atomize\_elim$  ( $rule$   $bchoice$ ,  $simp$ )
  then obtain  $vp2s$  where  $vp2s\_def$ :  $map$  ( $v\_at$ )  $vp2s = [ETP\_f \sigma i I ..< Suc$   $j] \wedge (\forall vp \in set$   $vp2s.$ 
 $v\_check$   $v$   $\psi$   $vp)$ 
    by  $atomize\_elim$  ( $auto$   $intro$ !:  $trans$ [ $OF$   $list.map\_cong$   $list.map\_id$ ]  $exI$ [ $of\_map$   $mypick$  ([ $ETP\_f$ 
 $\sigma i I ..< Suc$   $j$ ])])])
  then have  $v\_at$  ( $VUntil$   $i$   $vp2s$   $vp1$ ) =  $i \wedge v\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ ) ( $VUntil$   $i$   $vp2s$   $vp1$ )
    using  $VUntil$   $fb\_until$   $vp1\_def$  by simp
  then show  $\exists vp. v\_at$   $vp = i \wedge v\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ )  $vp$ 
    by blast
qed
next
case ( $VUntilInf$   $I$   $i$   $v$   $\psi$   $\varphi$ )
show ? $case$ 
proof
  assume  $fb\_until$ :  $future\_bounded$  ( $\varphi \mathbf{U} I \psi$ )
  then have  $fb$ :  $future\_bounded$   $\varphi$   $future\_bounded$   $\psi$ 
    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  $j\_def$ :  $j = LTP$   $\sigma$  ( $\tau$   $\sigma$   $i$  +  $b$ )
  obtain  $mypick$  where  $mypick\_def$ :  $\forall k \in \{ETP\_f \sigma i I .. j\}. v\_at$  ( $mypick$   $k$ ) =  $k \wedge v\_check$   $v$   $\psi$ 
    ( $mypick$   $k$ )
    using  $VUntilInf$   $fb\_until$  unfolding  $b\_def$   $j\_def$  by  $atomize\_elim$  ( $rule$   $bchoice$ ,  $simp$ )
  then obtain  $vp2s$  where  $vp2s\_def$ :  $map$  ( $v\_at$ )  $vp2s = [ETP\_f \sigma i I ..< Suc$   $j] \wedge (\forall vp \in set$   $vp2s.$ 
 $v\_check$   $v$   $\psi$   $vp)$ 
    by  $atomize\_elim$  ( $auto$   $intro$ !:  $trans$ [ $OF$   $list.map\_cong$   $list.map\_id$ ]  $exI$ [ $of\_map$   $mypick$  ([ $ETP\_f$ 
 $\sigma i I ..< Suc$   $j$ ])])])
  then have  $v\_at$  ( $VUntilInf$   $i$   $j$   $vp2s$ ) =  $i \wedge v\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ ) ( $VUntilInf$   $i$   $j$   $vp2s$ )
    using  $VUntilInf$   $b\_def$   $j\_def$  by simp
  then show  $\exists vp. v\_at$   $vp = i \wedge v\_check$   $v$  ( $\varphi \mathbf{U} I \psi$ )  $vp$ 
    by blast
qed
qed
lemmas  $check\_completeness =$ 
   $conjunct1$ [ $OF$   $check\_completeness\_aux$ ,  $rule\_format$ ]
   $conjunct2$ [ $OF$   $check\_completeness\_aux$ ,  $rule\_format$ ]

```

definition $p_check\ v\ \varphi\ p = (case\ p\ of\ Inl\ sp \Rightarrow s_check\ v\ \varphi\ sp \mid Inr\ vp \Rightarrow v_check\ v\ \varphi\ vp)$
definition $p_check_exec\ vs\ \varphi\ p = (case\ p\ of\ Inl\ sp \Rightarrow s_check_exec\ vs\ \varphi\ sp \mid Inr\ vp \Rightarrow v_check_exec\ vs\ \varphi\ vp)$

definition $valid :: ('n, 'd)\ envset \Rightarrow nat \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ proof \Rightarrow bool$ **where**
 $valid\ vs\ i\ \varphi\ p =$
 $(case\ p\ of$
 $\quad Inl\ p \Rightarrow s_check_exec\ vs\ \varphi\ p \wedge s_at\ p = i$
 $\quad \mid Inr\ p \Rightarrow v_check_exec\ vs\ \varphi\ p \wedge v_at\ p = i)$

end

8.7 Lifting the Checker to PDTs

fun $check_one$ **where**
 $check_one\ \sigma\ v\ \varphi\ (Leaf\ p) = p_check\ \sigma\ v\ \varphi\ p$
 $\mid check_one\ \sigma\ v\ \varphi\ (Node\ x\ part) = check_one\ \sigma\ v\ \varphi\ (lookup_part\ part\ (v\ x))$

fun $check_all_aux$ **where**
 $check_all_aux\ \sigma\ vs\ \varphi\ (Leaf\ p) = p_check_exec\ \sigma\ vs\ \varphi\ p$
 $\mid check_all_aux\ \sigma\ vs\ \varphi\ (Node\ x\ part) = (\forall (D, e) \in set\ (subsvals\ part). check_all_aux\ \sigma\ (vs(x := D))\ \varphi\ e)$

fun $collect_paths_aux$ **where**
 $collect_paths_aux\ DS\ \sigma\ vs\ \varphi\ (Leaf\ p) = (if\ p_check_exec\ \sigma\ vs\ \varphi\ p\ then\ \{\}\ else\ rev\ 'DS)$
 $\mid collect_paths_aux\ DS\ \sigma\ vs\ \varphi\ (Node\ x\ part) = (\bigcup (D, e) \in set\ (subsvals\ part). collect_paths_aux\ (Cons\ D\ 'DS)\ \sigma\ (vs(x := D))\ \varphi\ e)$

lemma $check_one_cong: \forall x \in fv\ \varphi \cup vars\ e. v\ x = v'\ x \implies check_one\ \sigma\ v\ \varphi\ e = check_one\ \sigma\ v'\ \varphi\ e$
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 $check_all_aux_check_one: \forall x. vs\ x \neq \{\} \implies distinct_paths\ e \implies (\forall x \in vars\ e. vs\ x = UNIV)$
 \implies

$check_all_aux\ \sigma\ vs\ \varphi\ e \longleftrightarrow (\forall v \in compatible_vals\ (fv\ \varphi)\ vs. check_one\ \sigma\ v\ \varphi\ e)$

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_subvals[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)
  qed (use Cons(2-) pick in ⟨simp_all add: vars_order.intros⟩)
qed

```

lemma *check_all_completeness:*

fixes $\varphi :: ('n, 'd :: \{\text{default}, \text{linorder}\}) \text{ formula}$

assumes *future_bounded* φ

shows $\exists e. \text{pdt_at } i \ e \wedge \text{check_all } \sigma \ \varphi \ e$

proof –

obtain *vs* **where** *vs[simp]: distinct vs set vs = fv φ*

by (*meson finite_distinct_list finite_fv*)

have *s*: *s_check* $\sigma \ v \ \varphi \ sp$

if *vars_order vs e*

and $\forall v. (\forall sp. \text{eval_pdt } v \ e = \text{Inl } sp \rightarrow (\exists x. x \notin \text{fv } \varphi \wedge v \ x \neq \text{undefined}) \vee \text{s_check } \sigma \ v \ \varphi \ sp) \wedge$
 $(\forall vp. \text{eval_pdt } v \ e = \text{Inr } vp \rightarrow (\exists x. x \notin \text{fv } \varphi \wedge v \ x \neq \text{undefined}) \vee \text{v_check } \sigma \ v \ \varphi \ vp)$

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* $\sigma \ v \ \varphi \ vp$

if *vars_order vs e*

and $\forall v. (\forall sp. \text{eval_pdt } v \ e = \text{Inl } sp \rightarrow (\exists x. x \notin \text{fv } \varphi \wedge v \ x \neq \text{undefined}) \vee \text{s_check } \sigma \ v \ \varphi \ sp) \wedge$
 $(\forall vp. \text{eval_pdt } v \ e = \text{Inr } vp \rightarrow (\exists x. x \notin \text{fv } \varphi \wedge v \ x \neq \text{undefined}) \vee \text{v_check } \sigma \ v \ \varphi \ vp)$

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  $\varphi$  i  $\lambda$ _. undefined  $\sigma$ ] assms
    unfolding check_all_alt p_check_def
    by (auto elim!: exI [where  $P = \lambda x. \_ x \wedge \_ x$ , OF conjI] simp: vars_order_distinct_paths split:
sum.splits intro: s v)
qed

```

```

lemma check_all_soundness_aux: check_all  $\sigma$   $\varphi$  e  $\implies$  p = eval_pdt v e  $\implies$  isl p  $\longleftrightarrow$  sat  $\sigma$  v (p_at
p)  $\varphi$ 

```

```

  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  $\sigma$   $\varphi$  e  $\implies$  pdt_at i e  $\implies$  isl (eval_pdt v e)  $\longleftrightarrow$  sat  $\sigma$  v i  $\varphi$ 
  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  $\rightarrow$  (OCaml) string

```

```

  | constant HOL.equal :: string8  $\Rightarrow$  string8  $\Rightarrow$  bool  $\rightarrow$  (OCaml) Stdlib.(=)

```

```

  | constant ( $\leq$ ) :: string8  $\Rightarrow$  string8  $\Rightarrow$  bool  $\rightarrow$  (OCaml) Stdlib.(<=)

```

```

  | constant (<) :: string8  $\Rightarrow$  string8  $\Rightarrow$  bool  $\rightarrow$  (OCaml) Stdlib.<

```

```

  | constant empty_string :: string8  $\rightarrow$  (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 $b \neq 0 \implies \text{div_to_zero } a \ b * b + \text{mod_to_zero } a \ b = a$

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  $\implies$  infinite (UNIV :: 'a set)
  and finite: universe = Some xs  $\implies$  distinct xs  $\wedge$  set xs = UNIV
begin

lemma finite_coset: finite (List.coset (xs :: 'a list)) = (case universe of None  $\implies$  False | _  $\implies$  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 (λzs. 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)  $CARD('b) = Suc\ 0\ a \neq b$  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  $\forall a. the\ (map\_of\ (zip\ xs\ ys)\ a) = the\ (map\_of\ (zip\ xs\ zs)\ a)$ 
    for xs :: 'a list and ys zs :: 'b list
    using that by (metis mapfst_zip map_of_eqI map_of_zip_inject map_of_zip_is_None option.expand)
  have 3:  $\exists zs. length\ zs = length\ xs \wedge set\ zs \subseteq set\ ys \wedge (\forall x. f\ x = the\ (map\_of\ (zip\ xs\ zs)\ x))$ 
    if  $\forall x. x \in set\ xs \forall x. x \in set\ ys$ 
    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* \Rightarrow ('n, 'd) *Formula.formula* \Rightarrow *nat* \Rightarrow *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* $_ \varphi$) *j* = *progress* σ φ *j*
| *progress* σ (*Formula.Forall* $_ \varphi$) *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*. $\forall k. k < j \wedge k \leq$ (*progress* σ φ *j*) \longrightarrow ($\tau \sigma k - \tau \sigma i$) \leq *right I*}
| *progress* σ (*Formula.Always* *I* φ) *j* =
| *Inf* {*i*. $\forall k. k < j \wedge k \leq$ (*progress* σ φ *j*) \longrightarrow ($\tau \sigma k - \tau \sigma i$) \leq *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*. $\forall k. k < j \wedge k \leq$ *min* (*progress* σ φ *j*) (*progress* σ ψ *j*) \longrightarrow ($\tau \sigma k - \tau \sigma i$) \leq *right I*}

lemma *Inf_Min*:
fixes *P* :: *nat* \Rightarrow *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* ($\lambda i. \text{enat } (\delta \sigma m i) \leq \text{right } I$) {..*j*}))

proof -

define *P* **where** *P* \equiv ($\lambda i. \forall k. k < j \wedge k \leq$ (*progress* σ φ *j*) \longrightarrow *enat* ($\delta \sigma k i$) \leq *right I*)

have *P_j*: *P j*

by (*auto simp: P_def enat_0*)

have *all_wit*: ($\forall k \in$ {..*m*}. *enat* ($\delta \sigma k i$) \leq *right I*) \longleftrightarrow (*enat* ($\delta \sigma (m - 1) i$) \leq *right I*) **for** *i m*

proof (*cases m*)

case (*Suc ma*)

have *k* < *Suc ma* \implies $\delta \sigma k i \leq \delta \sigma ma i$ **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*: $\text{progress } \sigma \text{ (Formula.Always } I \varphi) j =$
 $(\text{let } m = \min j \text{ (Suc (progress } \sigma \varphi j)) - 1 \text{ in Min (Set.filter } (\lambda i. \text{enat } (\delta \sigma m i) \leq \text{right } I) \{..j\}))$

```

proof -
  define P where P  $\equiv (\lambda i. \forall k. k < j \wedge k \leq (\text{progress } \sigma \varphi j) \longrightarrow \text{enat } (\delta \sigma k i) \leq \text{right } I)$ 
  have P_j: P j
    by (auto simp: P_def enat_0)
  have all_wit:  $(\forall k \in \{..<m\}. \text{enat } (\delta \sigma k i) \leq \text{right } I) \longleftrightarrow (\text{enat } (\delta \sigma (m - 1) i) \leq \text{right } I)$  for i m
  proof (cases m)
    case (Suc ma)
    have  $k < \text{Suc } ma \implies \delta \sigma k i \leq \delta \sigma ma i$  for k
    using  $\tau\_mono$ 
    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*: $\text{progress } \sigma \text{ (Formula.Until } \varphi I \psi) j =$
 $(\text{let } m = \min j \text{ (Suc (min (progress } \sigma \varphi j) (\text{progress } \sigma \psi j))) - 1 \text{ in Min (Set.filter } (\lambda i. \text{enat } (\delta \sigma m i) \leq \text{right } I) \{..j\}))$

```

proof -
  define P where P  $\equiv (\lambda i. \forall k. k < j \wedge k \leq \min (\text{progress } \sigma \varphi j) (\text{progress } \sigma \psi j) \longrightarrow \text{enat } (\delta \sigma k i) \leq \text{right } I)$ 
  have P_j: P j
    by (auto simp: P_def enat_0)
  have all_wit:  $(\forall k \in \{..<m\}. \text{enat } (\delta \sigma k i) \leq \text{right } I) \longleftrightarrow (\text{enat } (\delta \sigma (m - 1) i) \leq \text{right } I)$  for i m
  proof (cases m)
    case (Suc ma)
    have  $k < \text{Suc } ma \implies \delta \sigma k i \leq \delta \sigma ma i$  for k
    using  $\tau\_mono$ 
    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*: $(x \#\# s) !! n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } m \Rightarrow s !! m)$
by (cases n) auto

lemma *extend_is_stream*:

assumes *sorted* (map snd list)
and $\bigwedge x. x \in \text{set list} \Rightarrow \text{snd } x \leq m$
and $\bigwedge x. x \in \text{set list} \Rightarrow \text{finite } (\text{fst } x)$
shows *sorted* (smap snd (list @- smap ($\lambda n. (\{\}, n + m)$) nats)) \wedge
sincreasing (smap snd (list @- smap ($\lambda n. (\{\}, n + m)$) nats)) \wedge
sfinite (smap fst (list @- smap ($\lambda n. (\{\}, n + m)$) nats))

proof -

have *A*: $\forall x \in \text{set list}. n \leq \text{snd } x \Rightarrow n \leq m \Rightarrow$
 $n \leq (\text{map snd list @- smap } (\lambda x. x + m) \text{ nats}) !! i \text{ for } n \ i$
and *list* :: ('a set \times 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 *sorted* (smap snd (list @- smap ($\lambda n. (\{\}, n + m)$) nats))

using *assms*

by (induction list) (auto simp: stream.map_comp o_def sorted_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: increasing_def) presburger

next

case (Cons a as)

have *IH*: $\bigwedge x. \exists i. x < \text{smap snd } (as @- \text{smap } (\lambda n. (\{\}, n + m)) \text{ nats}) !! i$

using *Cons*

by (simp add: increasing_def)

show ?case

using *IH*

by (simp add: increasing_def)

(metis snth_Stream)

qed

moreover have *sfinite* (smap fst (list @- smap ($\lambda n. (\{\}, n + m)$) nats))

using *assms*(\mathcal{B})

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* = $\{(n, m, t) :: (\text{nat} \times \text{nat} \times (\text{nat}, 'a \text{ set} \times \text{nat}) \text{ mapping}) \mid$

$n\ m\ t.$ $\text{Mapping.keys } t = \{..\lt n\} \wedge$
 $\text{sorted } (\text{map } (\text{snd} \circ (\text{the} \circ \text{Mapping.lookup } t)) [0..\lt n]) \wedge$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } n' \Rightarrow (\text{case } \text{Mapping.lookup } t\ n' \text{ of } \text{Some } (X', t') \Rightarrow t' \leq m \mid \text{None} \Rightarrow \text{False}))$
 \wedge
 $(\forall n' < n. \text{case } \text{Mapping.lookup } t\ n' \text{ of } \text{Some } (X', t') \Rightarrow \text{finite } X' \mid \text{None} \Rightarrow \text{False})$
by (rule $\text{exI}[\text{of } _ (0, 0, \text{Mapping.empty})]$) *auto*

setup_lifting *type_definition_trace_mapping*

lemma $\text{lookup_bulkload_Some}: i < \text{length } \text{list} \Longrightarrow$
 $\text{Mapping.lookup } (\text{Mapping.bulkload } \text{list})\ i = \text{Some } (\text{list } !\ i)$
by *transfer auto*

lift_definition $\text{trace_mapping_of_list} :: ('a \text{ set} \times \text{nat}) \text{ list} \Rightarrow 'a \text{ trace_mapping}$ **is**
 $\lambda xs. \text{if } \text{sorted } (\text{map } \text{snd } xs) \wedge (\forall x \in \text{set } xs. \text{finite } (\text{fst } x)) \text{ then } (\text{if } xs = [] \text{ then } (0, 0, \text{Mapping.empty})$
 $\text{else } (\text{length } xs, \text{snd } (\text{last } xs), \text{Mapping.bulkload } xs))$
 $\text{else } (0, 0, \text{Mapping.empty})$
by (auto *simp*: $\text{lookup_bulkload_Some}$ $\text{sorted_iff_nth_Suc}$ last_conv_nth
 $\text{list_all_iff_in_set_conv_nth}$ Ball_def Bex_def $\text{image_iff_split: nat.splits}$)

lift_definition $\text{trace_mapping_nth} :: 'a \text{ trace_mapping} \Rightarrow \text{nat} \Rightarrow ('a \text{ set} \times \text{nat})$ **is**
 $\lambda(n, m, t)\ i. \text{if } i < n \text{ then the } (\text{Mapping.lookup } t\ i) \text{ else } (\{\}, (i - n) + m)$.

lift_definition $\text{Trace_Mapping} :: 'a \text{ trace_mapping} \Rightarrow 'a \text{ Trace.trace}$ **is**
 $\lambda(n, m, t). \text{map } (\text{the} \circ \text{Mapping.lookup } t) [0..\lt n] @- \text{smap } (\lambda n. (\{\} :: 'a \text{ set}, n + m)) \text{nats}$

proof (*goal_cases*)

case (1 *prod*)
obtain $n\ m\ t$ **where** $\text{prod_def}: \text{prod} = (n, m, t)$
by (*cases prod*) *auto*
have $\text{props}: \text{Mapping.keys } t = \{..\lt n\}$
 $\text{sorted } (\text{map } (\text{snd} \circ (\text{the} \circ \text{Mapping.lookup } t)) [0..\lt n])$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } n' \Rightarrow (\text{case } \text{Mapping.lookup } t\ n' \text{ of } \text{Some } (X', t') \Rightarrow t' \leq m \mid \text{None} \Rightarrow$
 $\text{False}))$
 $(\forall n' < n. \text{case } \text{Mapping.lookup } t\ n' \text{ of } \text{Some } (X', t') \Rightarrow \text{finite } X' \mid \text{None} \Rightarrow \text{False})$
using 1 **by** (auto *simp add*: prod_def)
have $\text{aux}: x \in \text{set } (\text{map } (\text{the} \circ \text{Mapping.lookup } t) [0..\lt n]) \Longrightarrow \text{snd } x \leq m$ **for** x
using $\text{props}(2,3)$ less_Suc_eq_le
by (*fastforce simp*: $\text{sorted_iff_nth_mono}$ $\text{split: nat.splits option.splits}$)
have $\text{aux2}: x \in \text{set } (\text{map } (\text{the} \circ \text{Mapping.lookup } t) [0..\lt n]) \Longrightarrow \text{finite } (\text{fst } x)$ **for** x
using $\text{props}(1,4)$
by (auto $\text{split: nat.splits option.splits}$)
show ?*case*
unfolding prod_def prod.case
by (rule $\text{extend_is_stream}[\text{where } ?m=m]$) (use props aux aux2 **in** $\langle \text{auto simp: prod_def} \rangle$)
qed

code_datatype Trace_Mapping

definition $\text{trace_of_list } xs = \text{Trace_Mapping } (\text{trace_mapping_of_list } xs)$

lemma $\Gamma_rbt_code[\text{code}]: \Gamma (\text{Trace_Mapping } t)\ i = \text{fst } (\text{trace_mapping_nth } t\ i)$
by *transfer (auto split: prod.splits)*

lemma $\tau_rbt_code[\text{code}]: \tau (\text{Trace_Mapping } t)\ i = \text{snd } (\text{trace_mapping_nth } t\ i)$
by *transfer (auto split: prod.splits)*

lemma $\text{trace_mapping_of_list_sound}: \text{sorted } (\text{map } \text{snd } xs) \wedge (\forall x \in \text{set } xs. \text{finite } (\text{fst } x)) \Longrightarrow i < \text{length } xs \Longrightarrow$

$xs ! i = (\Gamma (\text{trace_of_list } xs) i, \tau (\text{trace_of_list } xs) i)$
unfolding *trace_of_list_def*
by *transfer (auto simp: lookup_bulkload_Some)*

10.4 Auxiliary results

definition $\text{sum_proofs } f \text{ } xs = \text{sum_list } (\text{map } f \text{ } xs)$

lemma $\text{sum_proofs_empty}[simp]: \text{sum_proofs } f \text{ } [] = 0$
by *(auto simp: sum_proof_def)*

lemma $\text{sum_proofs_fundef_cong}[fundef_cong]: (\bigwedge x. x \in \text{set } xs \implies f x = f' x) \implies$
 $\text{sum_proofs } f \text{ } xs = \text{sum_proofs } f' \text{ } xs$
by *(induction xs) (auto simp: sum_proof_def)*

lemma sum_proofs_Cons :
fixes $f :: 'a \Rightarrow \text{nat}$
shows $\text{sum_proofs } f \text{ } (p \# qs) = f p + \text{sum_proofs } f \text{ } qs$
by *(auto simp: sum_proof_def split: list.splits)*

lemma sum_proofs_app :
fixes $f :: 'a \Rightarrow \text{nat}$
shows $\text{sum_proofs } f \text{ } (qs @ [p]) = f p + \text{sum_proofs } f \text{ } qs$
by *(auto simp: sum_proof_def split: list.splits)*

context
fixes $w :: 'n \Rightarrow \text{nat}$
begin

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

```

| 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 $p_pred :: ('n, 'd) \text{proof} \Rightarrow \text{nat}$ **where**
 $p_pred = \text{case_sum } s_pred \ v_pred$

end

10.5 v_check_exec setup

lemma $ETP_minus_le_iff$: $ETP \ \sigma \ (\tau \ \sigma \ i - n) \leq j \longleftrightarrow \delta \ \sigma \ i \ j \leq n$
 by (simp add: add commute $i_ETP_tau \ le_diff_conv$)

lemma $ETP_minus_gt_iff$: $j < ETP \ \sigma \ (\tau \ \sigma \ i - n) \longleftrightarrow \delta \ \sigma \ i \ j > n$
 by (meson $ETP_minus_le_iff \ leD \ le_less_linear$)

lemma $nat_le_iff_less$:

fixes $n :: \text{nat}$
shows $(j \leq n) \longleftrightarrow (j = 0 \vee j - 1 < n)$
 by auto

lemma $ETP_minus_eq_iff$: $j = ETP \ \sigma \ (\tau \ \sigma \ i - n) \longleftrightarrow ((j = 0 \vee n < \delta \ \sigma \ i \ (j - 1)) \wedge \delta \ \sigma \ i \ j \leq n)$
unfolding eq_iff [of $j \ ETP \ \sigma \ (\tau \ \sigma \ i - n)$] $nat_le_iff_less$ [of j] $ETP_minus_le_iff \ ETP_minus_gt_iff$
 by auto

lemma $LTP_minus_ge_iff$: $\tau \ \sigma \ 0 + n \leq \tau \ \sigma \ i \implies j \leq LTP \ \sigma \ (\tau \ \sigma \ i - n) \longleftrightarrow$
 $(\text{case } n \ \text{of } 0 \Rightarrow \delta \ \sigma \ j \ i = 0 \mid _ \Rightarrow j \leq i \wedge \delta \ \sigma \ i \ j \geq n)$
using τ_mono [of $i \ j \ \sigma$]
 by (fastforce simp add: $i_LTP_tau \ le_diff_conv2 \ Suc_le_eq \ split$: $nat.splits$)

lemma $LTP_plus_ge_iff$: $j \leq LTP \ \sigma \ (\tau \ \sigma \ i + n) \longleftrightarrow \delta \ \sigma \ j \ i \leq n$
 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* $\tau_mono[OF \text{that}, \text{of } \sigma]$
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 $LTP \ \sigma \ (\tau \ \sigma \ i - n) < j \longleftrightarrow (\text{if } \neg j \leq i \wedge n = 0 \text{ then } \delta \ \sigma \ j \ i > 0 \text{ else } \delta \ \sigma \ i \ j < n)$
proof (*cases* $j \leq i$)
case *True*
then show *?thesis*
using *assms* $i_le_LTPi_minus[of \ \sigma \ n \ i]$ $LTP_minus_lt_if[of \ j \ i \ \sigma \ n]$
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* $0 \Rightarrow i \leq j \wedge \delta \ \sigma \ j \ i = 0 \wedge \delta \ \sigma \ (Suc \ j) \ j > 0$
 $|_ \Rightarrow j \leq i \wedge n \leq \delta \ \sigma \ i \ j \wedge \delta \ \sigma \ i \ (Suc \ j) < n$
proof (*cases* n)
case 0

show *?thesis*
using *assms 0 i_LTP_tau*[of $\sigma \tau \sigma i$ *LTP* $\sigma (\tau \sigma i)$]
i_LTP_tau[of $\sigma \tau \sigma i$ *Suc* (*LTP* $\sigma (\tau \sigma i)$)] *i_LTP_tau*[of $\sigma \tau \sigma i j$]
less_τD[of σ (*LTP* $\sigma (\tau \sigma i)$) *Suc j*]
by (*auto simp: i_le_LTPi not_le elim!: antisym dest!*
order_antisym_conv[of $\tau \sigma i \tau \sigma j$, *THEN iffD1*, *rotated*]
order_antisym_conv[of $\tau \sigma i \tau \sigma$ (*LTP* $\sigma (\tau \sigma i)$), *THEN iffD1*, *rotated*])
next
case (*Suc n'*)
show *?thesis*
using *assms*
by (*simp add: Suc eq_iff*[of j *LTP* $\sigma (\tau \sigma i - \text{Suc } n')$] *less_Suc_eq_le*[of *LTP* $\sigma (\tau \sigma i - \text{Suc } n')$ j ,
symmetric] *LTP_minus_ge_iff LTP_minus_lt_iff*)
qed

lemma *LTP_plus_eq_iff*:
shows $j = \text{LTP } \sigma (\tau \sigma i + n) \longleftrightarrow (\delta \sigma j i \leq n \wedge \delta \sigma (\text{Suc } j) i > n)$
unfolding *eq_iff*[of j *LTP* $\sigma (\tau \sigma i + n)$]
by (*meson LTP_plus_ge_iff linorder_not_less not_less_eq_eq*)

lemma *LTP_p_def*: $\tau \sigma 0 + \text{left } I \leq \tau \sigma i \implies \text{LTP}_p \sigma i I = (\text{case left } I \text{ of } 0 \Rightarrow i \mid _ \Rightarrow \text{LTP } \sigma (\tau \sigma i - \text{left } I))$
using *i_le_LTPi* **by** (*auto simp: min_def i_LTP_tau split: nat.splits*)

definition *check_upt_LTP_p* $\sigma I li xs i \longleftrightarrow (\text{case } xs \text{ of } [] \Rightarrow$
(case left } I \text{ of } 0 \Rightarrow i < li \mid \text{Suc } n \Rightarrow
(if } \neg li \leq i \wedge \text{left } I = 0 \text{ then } 0 < \delta \sigma li i \text{ else } \delta \sigma i li < \text{left } I))
 $\mid _ \Rightarrow xs = [li..<li + \text{length } xs] \wedge$
(case left } I \text{ of } 0 \Rightarrow li + \text{length } xs - 1 = i \mid \text{Suc } n \Rightarrow
(li + \text{length } xs - 1 \leq i \wedge \text{left } I \leq \delta \sigma i (li + \text{length } xs - 1) \wedge \delta \sigma i (li + \text{length } xs) < \text{left } I)))

lemma *check_upt_l_cong*:
assumes $\bigwedge j. j \leq \max i li \implies \tau \sigma j = \tau \sigma' j$
shows $\text{check_upt_LTP}_p \sigma I li xs i = \text{check_upt_LTP}_p \sigma' I li xs i$
proof –
have $li + \text{length } ys \leq i \implies \text{Suc } n \leq \delta \sigma' i (li + \text{length } ys) \implies$
 $(\text{Suc } (li + \text{length } ys)) \leq i$ **for** $ys :: \text{nat list}$ **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 $\tau \sigma 0 + \text{left } I \leq \tau \sigma i$
shows $xs = [li..<\text{Suc } (\text{LTP}_p \sigma i I)] \longleftrightarrow \text{check_upt_LTP}_p \sigma I li xs i$
proof –
have $li + \text{length } xs = \text{Suc } (\text{LTP}_p \sigma i I) \longleftrightarrow li + \text{length } xs - 1 = \text{LTP}_p \sigma i I$ **if** $xs \neq []$
using *that*
by (*cases xs*) *auto*
then have $xs = [li..<\text{Suc } (\text{LTP}_p \sigma i I)] \longleftrightarrow (xs = [] \wedge \text{LTP}_p \sigma i I < li) \vee$
 $(xs \neq [] \wedge xs = [li..<li + \text{length } xs] \wedge li + \text{length } xs - 1 = \text{LTP}_p \sigma i I)$
by *auto*
moreover have $\dots \longleftrightarrow (xs = [] \wedge$
 $(\text{case left } I \text{ of } 0 \Rightarrow i < li \mid \text{Suc } n \Rightarrow$
 $(\text{if } \neg li \leq i \wedge \text{left } I = 0 \text{ then } 0 < \delta \sigma li i \text{ else } \delta \sigma i li < \text{left } I))) \vee$
 $(xs \neq [] \wedge xs = [li..<li + \text{length } xs] \wedge$
 $(\text{case left } I \text{ of } 0 \Rightarrow li + \text{length } xs - 1 = i \mid \text{Suc } n \Rightarrow$

$(\text{case left } I \text{ of } 0 \Rightarrow i \leq li + \text{length } xs - 1 \wedge$
 $\delta \sigma (li + \text{length } xs - 1) i = 0 \wedge 0 < \delta \sigma (\text{Suc } (li + \text{length } xs - 1)) (li + \text{length } xs - 1)$
 $| \text{Suc } n \Rightarrow li + \text{length } xs - 1 \leq i \wedge$
 $\text{left } I \leq \delta \sigma i (li + \text{length } xs - 1) \wedge \delta \sigma i (\text{Suc } (li + \text{length } xs - 1)) < \text{left } I))$
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 ... $\longleftrightarrow (\text{case } xs \text{ of } [] \Rightarrow$
 $(\text{case left } I \text{ of } 0 \Rightarrow i < li | \text{Suc } n \Rightarrow$
 $(\text{if } \neg li \leq i \wedge \text{left } I = 0 \text{ then } 0 < \delta \sigma li i \text{ else } \delta \sigma i li < \text{left } I))$
 $| _ \Rightarrow xs = [li..<li + \text{length } xs] \wedge$
 $(\text{case left } I \text{ of } 0 \Rightarrow li + \text{length } xs - 1 = i | \text{Suc } n \Rightarrow$
 $(li + \text{length } xs - 1 \leq i \wedge \text{left } I \leq \delta \sigma i (li + \text{length } xs - 1) \wedge \delta \sigma i (li + \text{length } xs) < \text{left } I))$
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*]: $v_check_exec \sigma vs (\text{Formula.Once } I \varphi) vp = (\text{case } vp \text{ of}$
 $VOnce \ i \ li \ vps \Rightarrow$
 $(\text{case right } I \text{ of } \infty \Rightarrow li = 0 | \text{enat } b \Rightarrow ((li = 0 \vee b < \delta \sigma i (li - 1)) \wedge \delta \sigma i li \leq b))$
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$
 $\wedge \text{check_upt_LTP_p } \sigma \ I \ li \ (\text{map } v_at \ vps) \ i \wedge \text{Ball } (\text{set } vps) (v_check_exec \sigma vs \varphi)$
 $| VOnceOut \ i \Rightarrow \tau \sigma i < \tau \sigma 0 + \text{left } I$
 $| _ \Rightarrow \text{False})$
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*]: $s_check_exec \sigma vs (\text{Formula.Historically } I \varphi) vp = (\text{case}$
 $vp \text{ of}$
 $SHistorically \ i \ li \ vps \Rightarrow$
 $(\text{case right } I \text{ of } \infty \Rightarrow li = 0 | \text{enat } b \Rightarrow ((li = 0 \vee b < \delta \sigma i (li - 1)) \wedge \delta \sigma i li \leq b))$
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$
 $\wedge \text{check_upt_LTP_p } \sigma \ I \ li \ (\text{map } s_at \ vps) \ i \wedge \text{Ball } (\text{set } vps) (s_check_exec \sigma vs \varphi)$
 $| SHistoricallyOut \ i \Rightarrow \tau \sigma i < \tau \sigma 0 + \text{left } I$
 $| _ \Rightarrow \text{False})$
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*]: $v_check_exec \sigma vs (\text{Formula.Since } \varphi \ I \ \psi) vp = (\text{case } vp \text{ of}$
 $VSince \ i \ vp1 \ vp2s \Rightarrow$
 $\text{let } j = v_at \ vp1 \ \text{in}$
 $(\text{case right } I \text{ of } \infty \Rightarrow \text{True} | \text{enat } b \Rightarrow \delta \sigma i j \leq b) \wedge j \leq i$
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$
 $\wedge \text{check_upt_LTP_p } \sigma \ I \ j \ (\text{map } v_at \ vp2s) \ i$
 $\wedge v_check_exec \sigma vs \varphi \ vp1 \wedge \text{Ball } (\text{set } vp2s) (v_check_exec \sigma vs \psi)$
 $| VSinceInf \ i \ li \ vp2s \Rightarrow$
 $(\text{case right } I \text{ of } \infty \Rightarrow li = 0 | \text{enat } b \Rightarrow ((li = 0 \vee b < \delta \sigma i (li - 1)) \wedge \delta \sigma i li \leq b)) \wedge$
 $\tau \sigma 0 + \text{left } I \leq \tau \sigma i \wedge$
 $\text{check_upt_LTP_p } \sigma \ I \ li \ (\text{map } v_at \ vp2s) \ i \wedge \text{Ball } (\text{set } vp2s) (v_check_exec \sigma vs \psi)$
 $| VSinceOut \ i \Rightarrow \tau \sigma i < \tau \sigma 0 + \text{left } I$
 $| _ \Rightarrow \text{False})$
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*: $\text{max } i \ (ETP \ \sigma \ (\tau \sigma \ i \ + \ a)) \leq j \longleftrightarrow i \leq j \wedge \delta \sigma j \ i \geq a$

by (metis add.commute max.bounded_iff τ _mono i_ETP_tau le_diff_conv2)

lemma ETP_f_ge_iff: $j \leq \max i (ETP \sigma (\tau \sigma i + n)) \iff (case\ n\ of\ 0 \Rightarrow j \leq i \mid Suc\ n' \Rightarrow (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta \sigma j' i < n))$

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: $\max i (ETP \sigma (\tau \sigma i + n)) = ETP \sigma (\tau \sigma i + n)$

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 $j \leq \max i (ETP \sigma (\tau \sigma i + n)) \iff (\forall ia. \tau \sigma i + n \leq \tau \sigma ia \longrightarrow j \leq ia)$

unfolding max

unfolding ETP_def

by (meson LeastI_ex Least_le order.trans ex_le_ τ)

moreover have $\dots \iff (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \neg \tau \sigma i + n \leq \tau \sigma j')$

by (auto split: nat.splits) (meson i_ETP_tau le_trans not_less_eq_eq)

moreover have $\dots \iff (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta \sigma j' i < n)$

by (auto simp: Suc_split: nat.splits)

ultimately show ?thesis

by (auto simp: Suc)

qed

definition check_upt_ETP_f $\sigma\ I\ i\ xs\ hi \iff (let\ j = Suc\ hi - length\ xs\ in$
 $(case\ xs\ of\ [] \Rightarrow (case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta \sigma hi\ i < left\ I)$
 $\mid _ \Rightarrow (xs = [j..<Suc\ hi] \wedge$
 $(case\ left\ I\ of\ 0 \Rightarrow j \leq i \mid Suc\ n' \Rightarrow$
 $(case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta \sigma j' i < left\ I))) \wedge$
 $i \leq j \wedge left\ I \leq \delta \sigma j\ i))$

lemma check_upt_lu_cong:

assumes $\bigwedge j. \min i\ hi \leq j \wedge j \leq \max i\ hi \implies \tau \sigma j = \tau \sigma' j$

shows $check_upt_ETP_f\ \sigma\ I\ i\ xs\ hi = check_upt_ETP_f\ \sigma'\ I\ i\ xs\ hi$

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: $xs = [ETP_f\ \sigma\ i\ I..<Suc\ hi] \iff check_upt_ETP_f\ \sigma\ I\ i\ xs\ hi$

proof -

have F1: $(case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta \sigma hi\ i < left\ I) =$

$(Suc\ hi \leq ETP_f\ \sigma\ i\ I)$

unfolding ETP_f_ge_iff[where ?j=Suc hi and ?n=left I]

by (auto split: nat.splits)

have $xs = [ETP_f\ \sigma\ i\ I..<Suc\ hi] \iff (let\ j = Suc\ hi - length\ xs\ in$

$(xs = [] \wedge (case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta \sigma hi\ i < left\ I)) \vee$

$(xs \neq [] \wedge xs = [j..<Suc\ hi] \wedge$

$(case\ left\ I\ of\ 0 \Rightarrow j \leq i \mid Suc\ n' \Rightarrow$

$(case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta \sigma j' i < left\ I))) \wedge$

$i \leq j \wedge left\ I \leq \delta \sigma j\ i)$

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 $\sigma\ i\ I$, symmetric]

by auto

moreover have $\dots \iff (let\ j = Suc\ hi - length\ xs\ in$

```

(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*: $\neg t \leq \tau \sigma i \implies i \leq (\text{LEAST } i. t \leq \tau \sigma i)$
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  $\sigma$  t = ETP_rec  $\sigma$  t 0
  using ETP_rec_sound[of  $\sigma$  t 0]
  by (auto simp: ETP_def)

lemma LTP_aux:
  assumes  $\tau \sigma (Suc\ i) \leq t$ 
  shows  $i \leq Max\ \{i.\ \tau \sigma\ i \leq t\}$ 
proof -
  have finite  $\{i.\ \tau \sigma\ i \leq t\}$ 
  by (smt (verit, del_insts)  $\tau$ _mono finite_nat_set_iff_bounded_le i_LTP_tau le0 le_trans mem_Collect_eq)

  moreover have  $i \in \{i.\ \tau \sigma\ i \leq t\}$ 
  using le_trans[OF  $\tau$ _mono[of i Suc i  $\sigma$ ] assms]
  by auto
  ultimately show ?thesis
  by (rule Max_ge)
qed

function (sequential) LTP_rec where
  LTP_rec  $\sigma$  t i = (if  $\tau \sigma (Suc\ i) \leq t$  then LTP_rec  $\sigma$  t (i + 1) else i)
  by pat_completeness auto
termination
  using LTP_aux
  by (relation measure  $(\lambda(\sigma, t, i).\ Suc\ (LTP\ \sigma\ t) - i)$  (fastforce simp: LTP_def)+)

lemma LTP_rec_sound: LTP_rec  $\sigma$  t j = Max ( $\{i.\ i \geq j \wedge (\tau \sigma\ i) \leq t\} \cup \{j\})$ 
proof (induction  $\sigma$  t j rule: LTP_rec.induct)
  case (1  $\sigma$  t j)
  have fin: finite  $\{i.\ j \leq i \wedge \tau \sigma\ i \leq t\}$ 
  by (smt (verit, del_insts)  $\tau$ _mono finite_nat_set_iff_bounded_le i_LTP_tau le0 le_trans mem_Collect_eq)
  show ?case
proof (cases  $\tau \sigma (Suc\ j) \leq t$ )
  case True
  have diffI:  $\{i.\ Suc\ j \leq i \wedge \tau \sigma\ i \leq t\} = \{i.\ j \leq i \wedge \tau \sigma\ i \leq t\} - \{j\}$ 
  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 $t < \tau \sigma$ 0
then *Code.abort* (STR "LTP: undefined") ($\lambda_.$ *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) \subseteq set ys) = (case universe of None \Rightarrow False
| Some $zs \Rightarrow \forall z \in$ set $zs. z \in$ set $xs \vee z \in$ 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 \Rightarrow False
| Some $zs \Rightarrow \forall z \in$ set $zs. z \in$ 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 \Rightarrow (' n , ' d) formula \Rightarrow (' n , ' d) expl \Rightarrow ' d
set list set option **where**
collect_paths σ φ $e =$ (if (*distinct_paths* $e \wedge$ *check_all_aux* σ ($\lambda_.$ UNIV) φ e) then None else Some
(*collect_paths_aux* {[]} σ ($\lambda_.$ UNIV) φ e))

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

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

definition *trace_of_list_specialized* :: ((name \times event_data list) set \times nat) list \Rightarrow (name, event_data)
trace **where**
trace_of_list_specialized $xs =$ *trace_of_list* xs

definition *specialized_set* :: (name \times event_data list) list \Rightarrow (name \times event_data list) set **where**
specialized_set = set

definition *ed_set* :: event_data list \Rightarrow event_data set **where**
ed_set = set

definition *sum_nat* :: nat \Rightarrow nat \Rightarrow nat **where**
sum_nat m $n = m + n$

definition *sub_nat* :: nat \Rightarrow nat \Rightarrow 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  $\bigwedge A1 A2. A1 \in \text{set} (\text{map fst} (\text{inP1 } P1 f v1 \text{ part2})) \implies$ 
   $A2 \in \text{set} (\text{map fst} (\text{inP1 } P1 f v1 \text{ part2})) \implies A1 \neq A2 \implies \text{disjnt } A1 A2$ 
  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 {}  $\notin \text{set} (\text{map fst} (\text{inP1 } P1 f v1 \text{ part2}))$ 
  using assms
  by (auto simp: map_filter_def split: if_splits)
show distinct (map fst ((P1, v1) # part1))  $\implies$  distinct (map fst part2)  $\implies$ 
  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
   $\equiv \text{List.map\_filter } (\lambda(P2, v2). \text{if } P2 - P1 \neq \{\} \text{ then Some}(P2 - P1, v2) \text{ else None}) \text{ part2}$ 
  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))  $\implies$  distinct (map fst (part2))  $\implies$ 
    distinct (map fst (notinP2 P1 f v1 part2))
proof (rule partition_onI)
  show  $\bigcup (\text{set} (\text{map fst} (\text{notinP2 } P1 f v1 \text{ part2}))) = X - P1$ 
  proof -
    have  $\exists P2. ((\exists x \in P2. x \notin P1) \longrightarrow (\exists v2. (P2, v2) \in \text{set part2})) \wedge (\exists x \in P2. x \notin P1) \wedge x \in P2$ 
      if  $\bigcup (\text{fst ' set part2}) = P1 \cup \bigcup (\text{fst ' set part1})$   $x \notin P1 (P1', v1) \in \text{set part1 } x \in P1' \text{ for } x P1' v1$ 
      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  $\bigwedge A1 A2. A1 \in \text{set} (\text{map fst} (\text{notinP2 } P1 f v1 \text{ part2})) \implies$ 
     $A2 \in \text{set} (\text{map fst} (\text{notinP2 } P1 f v1 \text{ part2})) \implies A1 \neq A2 \implies \text{disjnt } A1 A2$ 
    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 {}  $\notin \text{set} (\text{map fst} (\text{notinP2 } P1 f v1 \text{ part2}))$ 
    using assms
    by (auto simp: map_filter_def split: if_splits)
  show distinct (map fst ((P1, v1) # part1))  $\implies$  distinct (map fst part2)  $\implies$ 
    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 im-
      age_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  $\bigcup (\text{set} (\text{map fst part1})) = X - P1$ 
  using partition_onD1[OF assms(1)] partition_onD2[OF assms(1)] assms(2)
  by (auto simp: disjoint_def image_iff)
  show  $\bigwedge A1 A2. A1 \in \text{set} (\text{map fst part1}) \implies A2 \in \text{set} (\text{map fst part1}) \implies A1 \neq A2 \implies \text{disjnt } A1$ 
   $A2$ 
  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.prem1
    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.prem1 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.prem1 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.prem1 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))) \wedge 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))))
 \wedge 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))) \wedge 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))] (λ pt3
f'. f' pt3])
by simp
qed auto

lift_definition merge_part2 :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('d, 'a) part \Rightarrow ('d, 'a) part \Rightarrow ('d, 'a) part **is**
merge_part2_raw
by (rule wf_part_list_merge_part2_raw)

lift_definition merge_part3 :: ('a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('d, 'a) part \Rightarrow ('d, 'a) part \Rightarrow ('d, 'a) part \Rightarrow
('d, 'a) part **is** merge_part3_raw
by (rule wf_part_list_merge_part3_raw)

definition proof_app :: ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof (**infixl** \oplus 65) **where**

p \oplus q = (case (p, q) of
(Inl (SHistorically i li sps), Inl q) \Rightarrow Inl (SHistorically (i+1) li (sps @ [q]))
| (Inl (SAlways i hi sps), Inl q) \Rightarrow Inl (SAlways (i-1) hi (q # sps))
| (Inl (SSince sp2 sp1s), Inl q) \Rightarrow Inl (SSince sp2 (sp1s @ [q]))
| (Inl (SUntil sp1s sp2), Inl q) \Rightarrow Inl (SUntil (q # sp1s) sp2)
| (Inr (VSince i vp1 vp2s), Inr q) \Rightarrow Inr (VSince (i+1) vp1 (vp2s @ [q]))
| (Inr (VOnce i li vps), Inr q) \Rightarrow Inr (VOnce (i+1) li (vps @ [q]))
| (Inr (VEventually i hi vps), Inr q) \Rightarrow Inr (VEventually (i-1) hi (q # vps))
| (Inr (VSinceInf i li vp2s), Inr q) \Rightarrow Inr (VSinceInf (i+1) li (vp2s @ [q]))
| (Inr (VUntil i vp2s vp1), Inr q) \Rightarrow Inr (VUntil (i-1) (q # vp2s) vp1)
| (Inr (VUntilInf i hi vp2s), Inr q) \Rightarrow Inr (VUntilInf (i-1) hi (q # vp2s)))

definition proof_incr :: ('n, 'd) proof \Rightarrow ('n, 'd) proof **where**

proof_incr p = (case p of
Inl (SOnce i sp) \Rightarrow Inl (SOnce (i+1) sp)
| Inl (SEventually i sp) \Rightarrow Inl (SEventually (i-1) sp)
| Inl (SHistorically i li sps) \Rightarrow Inl (SHistorically (i+1) li sps)
| Inl (SAlways i hi sps) \Rightarrow Inl (SAlways (i-1) hi sps)
| Inr (VSince i vp1 vp2s) \Rightarrow Inr (VSince (i+1) vp1 vp2s)
| Inr (VOnce i li vps) \Rightarrow Inr (VOnce (i+1) li vps)
| Inr (VEventually i hi vps) \Rightarrow Inr (VEventually (i-1) hi vps)
| Inr (VHistorically i vp) \Rightarrow Inr (VHistorically (i+1) vp)
| Inr (VAlways i vp) \Rightarrow Inr (VAlways (i-1) vp)
| Inr (VSinceInf i li vp2s) \Rightarrow Inr (VSinceInf (i+1) li vp2s)
| Inr (VUntil i vp2s vp1) \Rightarrow Inr (VUntil (i-1) vp2s vp1)
| Inr (VUntilInf i hi vp2s) \Rightarrow Inr (VUntilInf (i-1) hi vp2s))

definition min_list_wrt :: (('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow bool) \Rightarrow ('n, 'd) proof list \Rightarrow ('n, 'd) proof
where

min_list_wrt r xs = hd [x \leftarrow xs. \forall y \in set xs. r x y]

definition do_neg :: ('n, 'd) proof \Rightarrow ('n, 'd) proof list **where**

do_neg p = (case p of
Inl sp \Rightarrow [Inr (VNeg sp)]

| $Inr\ vp \Rightarrow [Inl\ (SNeg\ vp)]$)

definition $do_or :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_or\ p1\ p2 = (case\ (p1, p2)\ of$
 $(Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SOrL\ sp1), Inl\ (SOrR\ sp2)]$
 $| (Inl\ sp1, Inr\ _) \Rightarrow [Inl\ (SOrL\ sp1)]$
 $| (Inr\ _ , Inl\ sp2) \Rightarrow [Inl\ (SOrR\ sp2)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inr\ (VOr\ vp1\ vp2)])$

definition $do_and :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_and\ p1\ p2 = (case\ (p1, p2)\ of$
 $(Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SAnd\ sp1\ sp2)]$
 $| (Inl\ _ , Inr\ vp2) \Rightarrow [Inr\ (VAndR\ vp2)]$
 $| (Inr\ vp1, Inl\ _) \Rightarrow [Inr\ (VAndL\ vp1)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inr\ (VAndL\ vp1), Inr\ (VAndR\ vp2)])$

definition $do_imp :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_imp\ p1\ p2 = (case\ (p1, p2)\ of$
 $(Inl\ _ , Inl\ sp2) \Rightarrow [Inl\ (SImpR\ sp2)]$
 $| (Inl\ sp1, Inr\ vp2) \Rightarrow [Inr\ (VImp\ sp1\ vp2)]$
 $| (Inr\ vp1, Inl\ sp2) \Rightarrow [Inl\ (SImpL\ vp1), Inl\ (SImpR\ sp2)]$
 $| (Inr\ vp1, Inr\ _) \Rightarrow [Inl\ (SImpL\ vp1)])$

definition $do_iff :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_iff\ p1\ p2 = (case\ (p1, p2)\ of$
 $(Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SIffSS\ sp1\ sp2)]$
 $| (Inl\ sp1, Inr\ vp2) \Rightarrow [Inr\ (VIffSV\ sp1\ vp2)]$
 $| (Inr\ vp1, Inl\ sp2) \Rightarrow [Inr\ (VIffVS\ vp1\ sp2)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inl\ (SIffVV\ vp1\ vp2)])$

definition $do_exists :: 'n \Rightarrow ('n, 'd::\{default, linorder\})\ proof + ('d, ('n, 'd)\ proof)\ part \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_exists\ x\ p_part = (case\ p_part\ of$
 $Inl\ p \Rightarrow (case\ p\ of$
 $Inl\ sp \Rightarrow [Inl\ (SEexists\ x\ default\ sp)]$
 $| Inr\ vp \Rightarrow [Inr\ (VExists\ x\ (trivial_part\ vp)])])$
 $| Inr\ part \Rightarrow (if\ (\exists\ x \in Vals\ part.\ isl\ x)\ then$
 $map\ (\lambda(D,p).\ map_sum\ (SEexists\ x\ (Min\ D))\ id\ p)\ (filter\ (\lambda(_, p).\ isl\ p)\ (subvals\ part))$
 $else$
 $[Inr\ (VExists\ x\ (map_part\ projr\ part))])$

definition $do_forall :: 'n \Rightarrow ('n, 'd::\{default, linorder\})\ proof + ('d, ('n, 'd)\ proof)\ part \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_forall\ x\ p_part = (case\ p_part\ of$
 $Inl\ p \Rightarrow (case\ p\ of$
 $Inl\ sp \Rightarrow [Inl\ (SForall\ x\ (trivial_part\ sp))]$
 $| Inr\ vp \Rightarrow [Inr\ (VForall\ x\ default\ vp)])$
 $| Inr\ part \Rightarrow (if\ (\forall\ x \in Vals\ part.\ isl\ x)\ then$
 $[Inl\ (SForall\ x\ (map_part\ projl\ part))]$
 $else$
 $map\ (\lambda(D,p).\ map_sum\ id\ (VForall\ x\ (Min\ D))\ p)\ (filter\ (\lambda(_, p).\ \neg isl\ p)\ (subvals\ part))$

definition $do_prev :: nat \Rightarrow \mathcal{I} \Rightarrow nat \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$

$do_prev\ i\ I\ t\ p = (case\ (p, t < left\ I)\ of$
 $(Inl\ _ , True) \Rightarrow [Inr\ (VPrevOutL\ i)]$
 $| (Inl\ sp, False) \Rightarrow (if\ mem\ t\ I\ then\ [Inl\ (SPrev\ sp)]\ else\ [Inr\ (VPrevOutR\ i)])$
 $| (Inr\ vp, True) \Rightarrow [Inr\ (VPrev\ vp), Inr\ (VPrevOutL\ i)]$
 $| (Inr\ vp, False) \Rightarrow (if\ mem\ t\ I\ then\ [Inr\ (VPrev\ vp)]\ else\ [Inr\ (VPrev\ vp), Inr\ (VPrevOutR\ i)])$

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

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

definition *do_once* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof list}$ **where**
do_once *i a p p'* = (case (*p*, *a* = 0, *p'*) of
 (*Inl* *sp*, *True*, *Inr* _) \Rightarrow [*Inl* (*SOnce* *i sp*)]
 | (*Inl* *sp*, *True*, *Inl* (*SOnce* _ *sp'*)) \Rightarrow [*Inl* (*SOnce* *i sp'*), *Inl* (*SOnce* *i sp*)]
 | (*Inl* _, *False*, *Inl* (*SOnce* _ *sp'*)) \Rightarrow [*Inl* (*SOnce* *i sp'*)]
 | (*Inl* _, *False*, *Inr* (*VOnce* _ *li vps'*)) \Rightarrow [*Inr* (*VOnce* *i li vps'*)]
 | (*Inr* _, *True*, *Inl* (*SOnce* _ *sp'*)) \Rightarrow [*Inl* (*SOnce* *i sp'*)]
 | (*Inr* *vp*, *True*, *Inr* *vp'*) \Rightarrow [(*Inr* *vp'*) \oplus (*Inr* *vp*)]
 | (*Inr* _, *False*, *Inl* (*SOnce* _ *sp'*)) \Rightarrow [*Inl* (*SOnce* *i sp'*)]
 | (*Inr* _, *False*, *Inr* (*VOnce* _ *li vps'*)) \Rightarrow [*Inr* (*VOnce* *i li vps'*)]

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

definition *do_eventually* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof list}$ **where**
do_eventually *i a p p'* = (case (*p*, *a* = 0, *p'*) of
 (*Inl* *sp*, *True*, *Inr* _) \Rightarrow [*Inl* (*SEventually* *i sp*)]
 | (*Inl* *sp*, *True*, *Inl* (*SEventually* _ *sp'*)) \Rightarrow [*Inl* (*SEventually* *i sp'*), *Inl* (*SEventually* *i sp*)]
 | (*Inl* _, *False*, *Inl* (*SEventually* _ *sp'*)) \Rightarrow [*Inl* (*SEventually* *i sp'*)]
 | (*Inl* _, *False*, *Inr* (*VEventually* _ *hi vps'*)) \Rightarrow [*Inr* (*VEventually* *i hi vps'*)]
 | (*Inr* _, *True*, *Inl* (*SEventually* _ *sp'*)) \Rightarrow [*Inl* (*SEventually* *i sp'*)]
 | (*Inr* *vp*, *True*, *Inr* *vp'*) \Rightarrow [(*Inr* *vp'*) \oplus (*Inr* *vp*)]
 | (*Inr* _, *False*, *Inl* (*SEventually* _ *sp'*)) \Rightarrow [*Inl* (*SEventually* *i sp'*)]
 | (*Inr* _, *False*, *Inr* (*VEventually* _ *hi vps'*)) \Rightarrow [*Inr* (*VEventually* *i hi vps'*)]

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

definition *do_historically* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof list}$ **where**
do_historically *i a p p'* = (case (*p*, *a* = 0, *p'*) of
 (*Inl* _, *True*, *Inr* (*VHistorically* _ *vp'*)) \Rightarrow [*Inr* (*VHistorically* *i vp'*)]
 | (*Inl* *sp*, *True*, *Inl* *sp'*) \Rightarrow [(*Inl* *sp'*) \oplus (*Inl* *sp*)]
 | (*Inl* _, *False*, *Inl* (*SHistorically* _ *li sps'*)) \Rightarrow [*Inl* (*SHistorically* *i li sps'*)]
 | (*Inl* _, *False*, *Inr* (*VHistorically* _ *vp'*)) \Rightarrow [*Inr* (*VHistorically* *i vp'*)]
 | (*Inr* *vp*, *True*, *Inl* _) \Rightarrow [*Inr* (*VHistorically* *i vp*)]
 | (*Inr* *vp*, *True*, *Inr* (*VHistorically* _ *vp'*)) \Rightarrow [*Inr* (*VHistorically* *i vp*), *Inr* (*VHistorically* *i vp'*)]
 | (*Inr* _, *False*, *Inl* (*SHistorically* _ *li sps'*)) \Rightarrow [*Inl* (*SHistorically* *i li sps'*)]
 | (*Inr* _, *False*, *Inr* (*VHistorically* _ *vp'*)) \Rightarrow [*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 $\sigma :: ('n, 'd :: \{\text{default}, \text{linorder}\}) \text{ trace and}$

$\text{cmp} :: ('n, 'd) \text{ proof} \Rightarrow ('n, 'd) \text{ proof} \Rightarrow \text{bool}$

begin

function (sequential) $\text{eval} :: 'n \text{ list} \Rightarrow \text{nat} \Rightarrow ('n, 'd) \text{ Formula.formula} \Rightarrow ('n, 'd) \text{ expl}$ **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] (\c. 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 (\x. x \in Formula.fv (Formula.Pred r ts)) vs) (List.map_filter (match ts) (sorted_list_of_set
(snd '{rd \in \Gamma \sigma i. fst rd = r})))
| eval vs i (Formula.Neg \varphi) = apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_neg p)) (eval vs i \varphi)
| eval vs i (Formula.Or \varphi \psi) = apply_pdt2 vs (\lambda p1 p2. min_list_wrt cmp (do_or p1 p2)) (eval vs i \varphi)
(eval vs i \psi)
| eval vs i (Formula.And \varphi \psi) = apply_pdt2 vs (\lambda p1 p2. min_list_wrt cmp (do_and p1 p2)) (eval vs i
\varphi) (eval vs i \psi)
| eval vs i (Formula.Imp \varphi \psi) = apply_pdt2 vs (\lambda p1 p2. min_list_wrt cmp (do_imp p1 p2)) (eval vs i
\varphi) (eval vs i \psi)
| eval vs i (Formula.Iff \varphi \psi) = apply_pdt2 vs (\lambda p1 p2. min_list_wrt cmp (do_iff p1 p2)) (eval vs i \varphi)
(eval vs i \psi)
| eval vs i (Formula.Exists x \varphi) = hide_pdt (vs @ [x]) (\lambda p. min_list_wrt cmp (do_exists x p)) (eval (vs
@ [x]) i \varphi)
| eval vs i (Formula.Forall x \varphi) = hide_pdt (vs @ [x]) (\lambda p. min_list_wrt cmp (do_forall x p)) (eval (vs
@ [x]) i \varphi)
| eval vs i (Formula.Prev I \varphi) = (if i = 0 then Leaf (Inr VPrevZ)
else apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_prev i I (\Delta \sigma i) p)) (eval vs
(i-1) \varphi))
| eval vs i (Formula.Next I \varphi) = apply_pdt1 vs (\lambda l. min_list_wrt cmp (do_next i I (\Delta \sigma (i+1)) l)) (eval
vs (i+1) \varphi)
| eval vs i (Formula.Once I \varphi) =
  (if \tau \sigma i < \tau \sigma 0 + left I then Leaf (Inr (VOnceOut i))
  else (let expl = eval vs i \varphi in
    (if i = 0 then
      apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_once_base 0 0 p)) expl
    else (if right I \ge enat (\Delta \sigma i) then
      apply_pdt2 vs (\lambda p p'. min_list_wrt cmp (do_once i (left I) p p')) expl
      (eval vs (i-1) (Formula.Once (subtract (\Delta \sigma i) I) \varphi))
    else apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_once_base i (left I) p)) expl))))
| eval vs i (Formula.Historically I \varphi) =
  (if \tau \sigma i < \tau \sigma 0 + left I then Leaf (Inl (SHistoricallyOut i))
  else (let expl = eval vs i \varphi in
    (if i = 0 then
      apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_historically_base 0 0 p)) expl
    else (if right I \ge enat (\Delta \sigma i) then
      apply_pdt2 vs (\lambda p p'. min_list_wrt cmp (do_historically i (left I) p p')) expl
      (eval vs (i-1) (Formula.Historically (subtract (\Delta \sigma i) I) \varphi))
    else apply_pdt1 vs (\lambda p. min_list_wrt cmp (do_historically_base i (left I) p)) expl))))

```



```

| eval vs i (Formula.Eventually I  $\varphi$ ) =
  (let expl = eval vs i  $\varphi$  in
   (if right I =  $\infty$  then undefined
    else (if right I  $\geq$  enat ( $\Delta$   $\sigma$  (i+1)) then
      apply_pdt2 vs ( $\lambda p p'$ . min_list_wrt cmp (do_eventually i (left I) p p')) expl
        (eval vs (i+1) (Formula.Eventually (subtract ( $\Delta$   $\sigma$  (i+1)) I)  $\varphi$ ))
      else apply_pdt1 vs ( $\lambda p$ . min_list_wrt cmp (do_eventually_base i (left I) p)) expl)))
| eval vs i (Formula.Always I  $\varphi$ ) =
  (let expl = eval vs i  $\varphi$  in
   (if right I =  $\infty$  then undefined
    else (if right I  $\geq$  enat ( $\Delta$   $\sigma$  (i+1)) then
      apply_pdt2 vs ( $\lambda p p'$ . min_list_wrt cmp (do_always i (left I) p p')) expl
        (eval vs (i+1) (Formula.Always (subtract ( $\Delta$   $\sigma$  (i+1)) I)  $\varphi$ ))
      else apply_pdt1 vs ( $\lambda p$ . min_list_wrt cmp (do_always_base i (left I) p)) expl)))
| eval vs i (Formula.Since  $\varphi$  I  $\psi$ ) =
  (if  $\tau \sigma i < \tau \sigma 0 + \text{left } I$  then Leaf (Inr (VSinceOut i))
   else (let expl1 = eval vs i  $\varphi$  in
         let expl2 = eval vs i  $\psi$  in
         (if i = 0 then
           apply_pdt2 vs ( $\lambda p1 p2$ . min_list_wrt cmp (do_since_base 0 0 p1 p2)) expl1 expl2
         else (if right I  $\geq$  enat ( $\Delta$   $\sigma$  i) then
           apply_pdt3 vs ( $\lambda p1 p2 p'$ . min_list_wrt cmp (do_since i (left I) p1 p2 p')) expl1 expl2
             (eval vs (i-1) (Formula.Since  $\varphi$  (subtract ( $\Delta$   $\sigma$  i) I)  $\psi$ ))
           else apply_pdt2 vs ( $\lambda p1 p2$ . min_list_wrt cmp (do_since_base i (left I) p1 p2)) expl1
             expl2))))))
| eval vs i (Formula.Until  $\varphi$  I  $\psi$ ) =
  (let expl1 = eval vs i  $\varphi$  in
   let expl2 = eval vs i  $\psi$  in
   (if right I =  $\infty$  then undefined
    else (if right I  $\geq$  enat ( $\Delta$   $\sigma$  (i+1)) then
      apply_pdt3 vs ( $\lambda p1 p2 p'$ . min_list_wrt cmp (do_until i (left I) p1 p2 p')) expl1 expl2
        (eval vs (i+1) (Formula.Until  $\varphi$  (subtract ( $\Delta$   $\sigma$  (i+1)) I)  $\psi$ ))
      else apply_pdt2 vs ( $\lambda p1 p2$ . 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  $\sigma$  (case right I of  $\infty \Rightarrow 0$  | enat b  $\Rightarrow (\tau \sigma i + b)$ ) - i
| dist i (Formula.Always I _) = LTP  $\sigma$  (case right I of  $\infty \Rightarrow 0$  | enat b  $\Rightarrow (\tau \sigma i + b)$ ) - i
| dist i (Formula.Since _ _ _) = i
| dist i (Formula.Until _ I _) = LTP  $\sigma$  (case right I of  $\infty \Rightarrow 0$  | enat b  $\Rightarrow (\tau \sigma i + b)$ ) - i
| dist _ _ = undefined

```

lemma i_less_LTP: $\tau \sigma (\text{Suc } i) \leq b + \tau \sigma i \implies i < \text{LTP } \sigma (b + \tau \sigma i)$

by (metis Suc_le_lessD i_le_LTPi_add le_iff_add)

termination eval

```

by (relation measures [ $\lambda(\_, \_, \varphi)$ . size  $\varphi$ ,  $\lambda(\_, i, \varphi)$ . dist i  $\varphi$ ]
  (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 (λi. eval (trace_of_list π) (λp 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.