

A Verified Proof Checker for Metric First-Order Temporal Logic

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

February 6, 2026

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	Regular expressions	10
3	Metric First-Order Temporal Logic	12
3.1	Syntax	12
3.2	Semantics	14
4	Valued Partitions	18
4.1	<i>size</i> setup	19
4.2	Functions on Valued Partitions	19
5	Partitioned Decision Trees	22
6	Proof System	29
6.1	Soundness and Completeness	31
7	Proof Objects	41
8	Auxiliary Lemmas	43
9	Proof Checker	60
9.1	Checker Soundness	63
9.2	Executable Variant of the Checker	80
9.3	Latest Relevant Time-Point	97
9.4	Active Domain	98

9.5	Congruence Modulo Active Domain	99
9.6	Checker Completeness	113
9.7	Lifting the Checker to PDTs	123
10	Type of Events	127
10.1	Code Adaptation for 8-bit strings	127
10.2	Event Parameters	128
11	Code Generation	129
11.1	Type Class Instances	129
11.2	Progress	131
11.3	Trace	133
11.4	Auxiliary results	135
11.5	<i>v_check_exec</i> setup	137
11.6	ETP/LTP setup	142
11.7	Exported functions	144
12	Unverified Explanation-Producing Monitoring Algorithm	145
13	Examples	155
13.1	Infinite Domain	156
13.2	Finite Domain	156

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*: $\text{sorted } s \longleftrightarrow (\forall i. s !! i \leq s !! \text{Suc } i)$
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*: $(\bigwedge x. \exists i. x < s !! i) \implies \text{sincreasing } s$
by (*simp add: increasing_def*)

lemma *sincreasing_grD*:

fixes $x :: 'a :: \text{semilattice_sup}$

assumes *sincreasing* s

shows $\exists j > i. x < s !! j$

proof –

let $?A = \text{insert } x \{s !! n \mid n. n \leq i\}$

from *assms* **obtain** j **where** $*$: $\text{Sup_fin } ?A < s !! j$

by (*auto simp: increasing_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 $s !! j \leq \text{Sup_fin } ?A$

by (*auto intro: Sup_fin.coboundedI*)

with $*$ **show** *False* **by** *simp*

qed

ultimately show *?thesis* **by** *blast*

qed

lemma *sincreasing_siterate_nat*[*simp*]:

fixes $n :: \text{nat}$

assumes $(\bigwedge n. n < f n)$

shows *sincreasing* (*siterate* $f n$)

unfolding *increasing_def* **proof**

fix x

show $\exists i. x < \text{siterate } f n !! i$

proof (*induction* x)

case 0

have $0 < \text{siterate } f n !! 1$

using *order.strict_trans1*[*OF le0 assms*] **by** *simp*

then show *?case* ..

next

case (*Suc* x)

then obtain i **where** $x < \text{siterate } f n !! i$..

then have *Suc* $x < \text{siterate } f n !! \text{Suc } i$

using *order.strict_trans1*[*OF _ assms*] **by** (*simp del: snth.simps*)

then show *?case* ..

qed

qed

lemma *sincreasing_stl*: *sincreasing* $s \implies \text{sincreasing } (\text{stl } s)$ **for** $s :: 'a :: \text{semilattice_sup}$ *stream*
by (*auto 0 4 simp: gr0_conv_Suc intro!: increasingI dest: increasing_grD*[of $s 0$])

definition *sfinite* $s = (\forall i. \text{finite } (s !! i))$

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

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

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

setup_lifting type_definition_trace

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

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

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

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

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

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

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

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

abbreviation Δ s i ≡ τ s i - τ s (i - 1)

1.2 Finite Trace Prefixes

typedef 'a prefix = {p :: ('a set × nat) list. sorted (map snd p)}

by (auto intro!: exI[of _ []])

setup_lifting type_definition_prefix

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

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

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

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

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

lift_definition psnoc :: 'a prefix ⇒ 'a set × nat ⇒ 'a prefix is
λp x. if (case p of [] ⇒ 0 | _ ⇒ snd (last p)) ≤ snd x then p @ [x] else []
proof (goal_cases sorted_psnoc)
case (sorted_psnoc p x)

```

then show ?case
  by (induction p) (auto split: if_splits list.splits)
qed

instantiation prefix :: (type) order begin

lift_definition less_eq_prefix :: 'a prefix  $\Rightarrow$  '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  $\lambda p s. \text{stake } (\text{length } p) s = p$  .

lemma prefix_of_pnil[simp]: prefix_of pnul  $\sigma$ 
  by transfer auto

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

lemma plen_mono:  $\pi \leq \pi' \Longrightarrow \text{plen } \pi \leq \text{plen } \pi'$ 
  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]: pnul  $\leq$   $\pi$ 
  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  $\sigma$ ) = 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]*: $\text{prefix_of } (\text{take_prefix } i \ \sigma) \ \sigma$
by transfer auto

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

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

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

lemma *prefix_of_imp_linear*: $\text{prefix_of } \pi \ \sigma \Longrightarrow \text{prefix_of } \pi' \ \sigma \Longrightarrow \pi \leq \pi' \vee \pi' \leq \pi$
proof transfer

fix $\pi \ \pi'$ and $\sigma :: ('a \text{ set} \times \text{nat}) \text{ stream}$

assume *assms*: $\text{stake } (\text{length } \pi) \ \sigma = \pi \ \text{stake } (\text{length } \pi') \ \sigma = \pi'$

show $(\exists r. \pi' = \pi \ @ \ r) \vee (\exists r. \pi = \pi' \ @ \ r)$

proof (cases *length* π *length* π' rule: le_cases)

case *le*

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

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

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

lemma *sincreasing_sdrop*:

fixes $s :: ('a :: \text{semilattice_sup}) \text{ stream}$

assumes *sincreasing* s

shows *sincreasing* (*sdrop* $n \ s$)

proof (rule *sincreasingI*)

fix x

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

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

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

by (*simp add: sdrop_snth*)

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

qed

lemma *sorted_shift*:

$\text{sorted } (xs \ @ \ s) = (\text{sorted } xs \ \wedge \ \text{sorted } s \ \wedge \ (\forall x \in \text{set } xs. \forall y \in \text{set } s. x \leq y))$

proof *safe*

```

assume *: sorted (xs @- s)
then show sorted xs
  by (auto simp: sorted_iff_mono shift_snth sorted_iff_nth_mono split: if_splits)
from sorted_sdrop[OF *, of length xs] show sorted 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 sorted_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 (sorted xs s)
  with <sorted s> show ?case
  by (subst (asm) sorted.simps) (auto 0 4 simp: neq_Nil_conv shd_sset intro: exI[of _ _ # _])
qed
qed

```

```

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

```

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

```

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

```

1.3 Earliest and Latest Time-Points

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

lemma i_ETP_tau: i ≥ ETP σ n ↔ τ σ i ≥ n
proof

```

```

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

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

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

```

```

    using subset_eq_atLeast0_lessThan_finite[of A Suc j]
    by simp
  moreover have  $i \in A$  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

```

lemma *etp_0[simp]*: $ETP \sigma 0 = 0$
using *i_ETP_tau* **by** *auto*

2 Regular expressions

context begin

qualified datatype (*atms*: 'a) *regex* = *Skip nat* | *Test 'a*
| *Plus 'a regex 'a regex* | *Times 'a regex 'a regex* | *Star 'a regex*

lemma *finite_atms[simp]*: *finite (atms r)*
by (*induct r*) *auto*

definition *Wild* = *Skip 1*

lemma *size_regex_estimation[termination_simp]*: $x \in \text{atms } r \implies y < f x \implies y < \text{size_regex } f r$
by (*induct r*) *auto*

lemma *size_regex_estimation'[termination_simp]*: $x \in \text{atms } r \implies y \leq f x \implies y \leq \text{size_regex } f r$
by (*induct r*) *auto*

qualified definition *TimesL* *r S* = *Times r ' S*

qualified definition *TimesR* *R s* = ($\lambda r. \text{Times } r s$) ' *R*

qualified primrec *collect where*

collect f (Skip n) = {}
| *collect f (Test φ)* = *f φ*
| *collect f (Plus r s)* = *collect f r* \cup *collect f s*
| *collect f (Times r s)* = *collect f r* \cup *collect f s*
| *collect f (Star r)* = *collect f r*

lemma *collect_cong[fundef_cong]*:
 $r = r' \implies (\bigwedge z. z \in \text{atms } r \implies f z = f' z) \implies \text{collect } f r = \text{collect } f' r'$
by (*induct r arbitrary: r'*) *auto*

lemma *finite_collect[simp]*: $(\bigwedge z. z \in \text{atms } r \implies \text{finite } (f z)) \implies \text{finite } (\text{collect } f r)$
by (*induct r*) *auto*

lemma *collect_commute*:

$(\bigwedge z. z \in \text{atms } r \implies x \in f z \iff g x \in f' z) \implies x \in \text{collect } f r \iff g x \in \text{collect } f' r$
by (*induct r*) *auto*

lemma *collect_alt*: $\text{collect } f r = (\bigcup z \in \text{atms } r. f z)$
by (*induct r*) *auto*

qualified definition *ncollect where*

ncollect f r = *Max (insert 0 (Suc ' collect f r))*

lemma *insert_Un*: $\text{insert } x (A \cup B) = \text{insert } x A \cup \text{insert } x B$
by *auto*

lemma *ncollect_simps[simp]*:
assumes [*simp*]: $(\bigwedge z. z \in \text{atms } r \implies \text{finite } (f z)) (\bigwedge z. z \in \text{atms } s \implies \text{finite } (f z))$
shows
ncollect f (Skip n) = 0
ncollect f (Test φ) = *Max (insert 0 (Suc ' f φ))*

$ncollect\ f\ (Plus\ r\ s) = \max\ (ncollect\ f\ r)\ (ncollect\ f\ s)$
 $ncollect\ f\ (Times\ r\ s) = \max\ (ncollect\ f\ r)\ (ncollect\ f\ s)$
 $ncollect\ f\ (Star\ r) = ncollect\ f\ r$
unfolding $ncollect_def$
by (*auto simp add: image_Un Max_Un insert_Un simp del: Un_insert_right Un_insert_left*)

abbreviation $min_regex_default\ f\ r\ j \equiv (if\ atms\ r = \{\}\ then\ j\ else\ Min\ ((\lambda z. f\ z\ j)\ 'atms\ r))$

qualified primrec $match :: (nat \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ regex \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

$match\ test\ (Skip\ n) = (\lambda i\ j. j = i + n)$
 $| match\ test\ (Test\ \varphi) = (\lambda i\ j. i = j \wedge test\ i\ \varphi)$
 $| match\ test\ (Plus\ r\ s) = match\ test\ r \sqcup match\ test\ s$
 $| match\ test\ (Times\ r\ s) = match\ test\ r\ OO\ match\ test\ s$
 $| match\ test\ (Star\ r) = (match\ test\ r)^{**}$

lemma $match_cong[fundef_cong]$:

$r = r' \implies (\bigwedge i\ z. z \in atms\ r \implies t\ i\ z = t'\ i\ z) \implies match\ t\ r = match\ t'\ r'$
by (*induct r arbitrary: r'*) *auto*

lemma $match_le$: $match\ test\ r\ i\ j \implies i \leq j$

proof (*induction r arbitrary: i j*)

case ($Times\ r\ s$)

then show *?case using order.trans by fastforce*

next

case ($Star\ r$)

from $Star.prem$ s **show** *?case*

unfolding $match.simps$ **by** (*induct i j rule: rtranclp.induct*) (*force dest: Star.IH*)**+**

qed *auto*

lemma $match_rtrancl_le$: $(match\ test\ r)^{**}\ i\ j \implies i \leq j$

by (*metis match.simps(5) match_le*)

lemma $match_map_regex$: $match\ t\ (map_regex\ f\ r) = match\ (\lambda k\ z. t\ k\ (f\ z))\ r$

by (*induct r*) *auto*

lemma $match_mono_strong$:

$(\bigwedge k\ z. k \in \{i..j+1\} \implies z \in atms\ r \implies t\ k\ z \implies t'\ k\ z) \implies match\ t\ r\ i\ j \implies match\ t'\ r\ i\ j$

proof (*induction r arbitrary: i j*)

case ($Times\ r\ s$)

from $Times.prem$ s **show** *?case*

by (*auto 0 4 simp: relcompp_apply intro: le_less_trans match_le less_Suc_eq_le*
dest: Times.IH[rotated -1] match_le)

next

case ($Star\ r$)

from $Star(3)$ **show** *?case unfolding match.simps*

proof –

assume $*$: $(match\ t\ r)^{**}\ i\ j$

then have $i \leq j$ **unfolding** $match.simps(5)[symmetric]$

by (*rule match_le*)

with $*$ **show** $(match\ t'\ r)^{**}\ i\ j$ **using** $Star.prem$ s

proof (*induction i j rule: rtranclp.induct*)

case ($rtrancl_into_rtrancl\ a\ b\ c$)

from $rtrancl_into_rtrancl(1,2,4,5)$ **show** *?case*

by (*intro rtranclp.rtrancl_into_rtrancl[OF rtrancl_into_rtrancl.IH]*)

(auto dest!: Star.IH[rotated -1])

dest: match_le match_rtranclp_le simp: less_Suc_eq_le

qed *simp*

qed

qed auto

lemma match_cong_strong:

$(\bigwedge k z. k \in \{i ..< j + 1\} \implies z \in \text{atms } r \implies t k z = t' k z) \implies \text{match } t r i j = \text{match } t' r i j$
using match_mono_strong[of i j r t t'] match_mono_strong[of i j r t' t] by blast

end

3 Metric First-Order Temporal Logic

3.1 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 \Rightarrow 'a

type_synonym ('n, 'a) envset = 'n \Rightarrow 'a set

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

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 ($\langle \top \rangle$)
| FF ($\langle \perp \rangle$)
| Eq_Const 'n 'a ($\langle _ \approx _ \rangle$ [85, 85] 85)
| Pred 'n ('n, 'a) trm list ($\langle _ \dagger _ \rangle$ [85, 85] 85)
| Neg ('n, 'a) formula ($\langle \neg_F _ \rangle$ [82] 82)
| Or ('n, 'a) formula ('n, 'a) formula (infixr $\langle \vee_F \rangle$ 80)
| And ('n, 'a) formula ('n, 'a) formula (infixr $\langle \wedge_F \rangle$ 80)
| Imp ('n, 'a) formula ('n, 'a) formula (infixr $\langle \longrightarrow_F \rangle$ 79)
| Iff ('n, 'a) formula ('n, 'a) formula (infixr $\langle \longleftrightarrow_F \rangle$ 79)
| Exists 'n ('n, 'a) formula ($\langle \exists_{F_} _ \rangle$ [70, 70] 70)
| Forall 'n ('n, 'a) formula ($\langle \forall_{F_} _ \rangle$ [70, 70] 70)
| Prev \mathcal{I} ('n, 'a) formula ($\langle \mathbf{Y} _ _ \rangle$ [1000, 65] 65)
| Next \mathcal{I} ('n, 'a) formula ($\langle \mathbf{X} _ _ \rangle$ [1000, 65] 65)
| Once \mathcal{I} ('n, 'a) formula ($\langle \mathbf{P} _ _ \rangle$ [1000, 65] 65)
| Historically \mathcal{I} ('n, 'a) formula ($\langle \mathbf{H} _ _ \rangle$ [1000, 65] 65)
| Eventually \mathcal{I} ('n, 'a) formula ($\langle \mathbf{F} _ _ \rangle$ [1000, 65] 65)
| Always \mathcal{I} ('n, 'a) formula ($\langle \mathbf{G} _ _ \rangle$ [1000, 65] 65)
| Since ('n, 'a) formula \mathcal{I} ('n, 'a) formula ($\langle _ \mathbf{S} _ _ \rangle$ [60, 1000, 60] 60)
| Until ('n, 'a) formula \mathcal{I} ('n, 'a) formula ($\langle _ \mathbf{U} _ _ \rangle$ [60, 1000, 60] 60)
| MatchP \mathcal{I} ('n, 'a) formula Regex.regex ($\langle \triangleleft _ _ \rangle$ [1000, 60] 60)
| MatchF \mathcal{I} ('n, 'a) formula Regex.regex ($\langle \triangleright _ _ \rangle$ [1000, 60] 60)

fun fv :: ('n, 'a) formula \Rightarrow 'n set where

fv (r \dagger ts) = \bigcup (fv_trm ' set ts)

| fv \top = {}

| fv \perp = {}

| fv (x \approx c) = {x}

| fv ($\neg_F \varphi$) = fv φ

| fv ($\varphi \vee_F \psi$) = fv $\varphi \cup$ fv ψ

| fv ($\varphi \wedge_F \psi$) = fv $\varphi \cup$ fv ψ

| fv ($\varphi \longrightarrow_F \psi$) = fv $\varphi \cup$ fv ψ

| fv ($\varphi \longleftrightarrow_F \psi$) = fv $\varphi \cup$ fv ψ

```

|  $fv (\exists_F x. \varphi) = fv \varphi - \{x\}$ 
|  $fv (\forall_F x. \varphi) = fv \varphi - \{x\}$ 
|  $fv (\mathbf{Y} I \varphi) = fv \varphi$ 
|  $fv (\mathbf{X} I \varphi) = fv \varphi$ 
|  $fv (\mathbf{P} I \varphi) = fv \varphi$ 
|  $fv (\mathbf{H} I \varphi) = fv \varphi$ 
|  $fv (\mathbf{F} I \varphi) = fv \varphi$ 
|  $fv (\mathbf{G} I \varphi) = fv \varphi$ 
|  $fv (\varphi \mathbf{S} I \psi) = fv \varphi \cup fv \psi$ 
|  $fv (\varphi \mathbf{U} I \psi) = fv \varphi \cup fv \psi$ 
|  $fv (\triangleleft I r) = Regex.collect fv r$ 
|  $fv (\triangleright I r) = Regex.collect fv r$ 

```

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

consts ($r \dagger ts$) = {} — terms may also contain constants, but these only filter out values from the trace and do not introduce new values of interest (i.e., do not extend the active domain)

```

| consts  $\top = \{\}$ 
| consts  $\perp = \{\}$ 
| 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 ( $\varphi \longleftrightarrow_F \psi$ ) = 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  $\psi$ 
| consts ( $\triangleleft I r$ ) = Regex.collect consts  $r$ 
| consts ( $\triangleright I r$ ) = Regex.collect consts  $r$ 

```

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

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

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

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

fun *future_bounded* :: ('n, 'a) formula \Rightarrow bool **where**

```

| future_bounded  $\top = \text{True}$ 
| future_bounded  $\perp = \text{True}$ 
| 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  $\psi$ )

```

$| \text{future_bounded } (\varphi \longleftrightarrow_F \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\exists_{F_} \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\forall_{F_} \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{Y} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{X} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{P} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{H} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{F} I \varphi) = (\text{future_bounded } \varphi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\mathbf{G} I \varphi) = (\text{future_bounded } \varphi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\varphi \mathbf{S} I \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\varphi \mathbf{U} I \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\triangleleft I r) = \text{Regex.pred_regex future_bounded } r$
 $| \text{future_bounded } (\triangleright I r) = (\text{Regex.pred_regex future_bounded } r \wedge \text{right } I \neq \infty)$

3.2 Semantics

primrec $\text{eval_trm} :: ('n, 'a) \text{env} \Rightarrow ('n, 'a) \text{trm} \Rightarrow 'a \langle _ _ \rangle$ [70,89] 89) **where**

$\text{eval_trm } v (\mathbf{v} x) = v x$
 $| \text{eval_trm } v (\mathbf{c} x) = x$

lemma $\text{eval_trm_fv_cong}: \forall x \in \text{fv_trm } t. v x = v' x \Longrightarrow v \llbracket t \rrbracket = v' \llbracket t \rrbracket$

by (induction t) simp_all

definition $\text{eval_trms} :: ('n, 'a) \text{env} \Rightarrow ('n, 'a) \text{trm list} \Rightarrow 'a \text{ list } \langle _ _ \rangle$ [70,89] 89) **where**

$\text{eval_trms } v ts = \text{map } (\text{eval_trm } v) ts$

lemma eval_trms_fv_cong :

$\forall t \in \text{set } ts. \forall x \in \text{fv_trm } t. v x = v' x \Longrightarrow v \llbracket ts \rrbracket = v' \llbracket ts \rrbracket$

using $\text{eval_trm_fv_cong}[\text{of_ } v v']$

by (auto simp: eval_trms_def)

primrec $\text{eval_trm_set} :: ('n, 'a) \text{envset} \Rightarrow ('n, 'a) \text{trm} \Rightarrow ('n, 'a) \text{trm} \times 'a \text{ set} \langle _ _ \rangle$ [70,89] 89)

where

$\text{eval_trm_set } vs (\mathbf{v} x) = (\mathbf{v} x, vs x)$
 $| \text{eval_trm_set } vs (\mathbf{c} x) = (\mathbf{c} x, \{x\})$

definition $\text{eval_trms_set} :: ('n, 'a) \text{envset} \Rightarrow ('n, 'a) \text{trm list} \Rightarrow ((n, 'a) \text{trm} \times 'a \text{ set}) \text{ list } \langle _ _ \rangle$ [70,89] 89)

where $\text{eval_trms_set } vs ts = \text{map } (\text{eval_trm_set } vs) ts$

lemma $\text{eval_trms_set_Nil}: vs \llbracket [] \rrbracket = []$

by (simp add: eval_trms_set_def)

lemma $\text{eval_trms_set_Cons}$:

$vs \llbracket (t \# ts) \rrbracket = vs \llbracket t \rrbracket \# vs \llbracket ts \rrbracket$

by (simp add: eval_trms_set_def)

fun $\text{sat} :: ('n, 'a) \text{trace} \Rightarrow ('n, 'a) \text{env} \Rightarrow \text{nat} \Rightarrow ('n, 'a) \text{formula} \Rightarrow \text{bool} \langle _ _ _ \rangle \models _ \rangle$ [56, 56, 56, 56] 55) **where**

$\langle \sigma, v, i \rangle \models \top = \text{True}$
 $| \langle \sigma, v, i \rangle \models \perp = \text{False}$
 $| \langle \sigma, v, i \rangle \models r \dagger ts = ((r, v \llbracket ts \rrbracket) \in \Gamma \sigma i)$
 $| \langle \sigma, v, i \rangle \models x \approx c = (v x = c)$
 $| \langle \sigma, v, i \rangle \models \neg_F \varphi = (\neg \langle \sigma, v, i \rangle \models \varphi)$
 $| \langle \sigma, v, i \rangle \models \varphi \vee_F \psi = (\langle \sigma, v, i \rangle \models \varphi \vee \langle \sigma, v, i \rangle \models \psi)$
 $| \langle \sigma, v, i \rangle \models \varphi \wedge_F \psi = (\langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i \rangle \models \psi)$
 $| \langle \sigma, v, i \rangle \models \varphi \longrightarrow_F \psi = (\langle \sigma, v, i \rangle \models \varphi \longrightarrow \langle \sigma, v, i \rangle \models \psi)$

$\langle \sigma, v, i \rangle \models \varphi \longleftrightarrow_F \psi = (\langle \sigma, v, i \rangle \models \varphi \longleftrightarrow \langle \sigma, v, i \rangle \models \psi)$
 $\langle \sigma, v, i \rangle \models \exists_F x. \varphi = (\exists z. \langle \sigma, v(x := z), i \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \forall_F x. \varphi = (\forall z. \langle \sigma, v(x := z), i \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{Y} I \varphi = (\text{case } i \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } j \Rightarrow \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{X} I \varphi = (\text{mem } (\tau \sigma (\text{Suc } i) - \tau \sigma i) I \wedge \langle \sigma, v, \text{Suc } i \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{P} I \varphi = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{H} I \varphi = (\forall j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \longrightarrow \langle \sigma, v, j \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{F} I \varphi = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \mathbf{G} I \varphi = (\forall j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \longrightarrow \langle \sigma, v, j \rangle \models \varphi)$
 $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{j < .. i\}. \langle \sigma, v, k \rangle \models \varphi))$
 $\langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{i .. < j\}. \langle \sigma, v, k \rangle \models \varphi))$
 $\langle \sigma, v, i \rangle \models (\triangleleft I r) = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \text{Regex.match } (\lambda k \varphi. \langle \sigma, v, k \rangle \models \varphi) r j i)$
 $\langle \sigma, v, i \rangle \models (\triangleright I r) = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \text{Regex.match } (\lambda k \varphi. \langle \sigma, v, k \rangle \models \varphi) r i j)$

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

proof (*induct* φ *arbitrary*: $v v' i$)

case (*Pred* n ts)

thus *?case*

by (*simp cong*: *map_cong eval_trms_fv_cong*[*rule_format*, *OF Pred*[*simplified*, *rule_format*]]
split: *option.splits*)

next

case (*Exists* t φ)

then show *?case unfolding sat.simps*

by (*intro iff_exI*) (*simp add*: *nth_Cons'*)

next

case (*Forall* t φ)

then show *?case unfolding sat.simps*

by (*intro iff_allI*) (*simp add*: *nth_Cons'*)

qed (*auto 10 0 simp*: *Let_def collect_alt split*: *nat.splits intro!*: *iff_exI eval_trm_fv_cong elim!*: *match_cong_strong*[*THEN iffD1*, *rotated*])

lemma *sat_Until_rec*: $\langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi \longleftrightarrow$

$(\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \psi \vee$

$\Delta \sigma (i + 1) \leq \text{right } I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (\text{subtract } (\Delta \sigma (i + 1)) I) \psi)$

(*is ?L* \longleftrightarrow *?R*)

proof (*rule iffI*; (*elim disjE conjE*)?)

assume *?L*

then obtain j **where** $j: i \leq j \text{ mem } (\tau \sigma j - \tau \sigma i) I \langle \sigma, v, j \rangle \models \psi \forall k \in \{i .. < j\}. \langle \sigma, v, k \rangle \models \varphi$

by *auto*

then show *?R*

proof (*cases* $i = j$)

case *False*

with $j(1,2)$ **have** $\Delta \sigma (i + 1) \leq \text{right } I$

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** $\langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (\text{subtract } (\Delta \sigma (i + 1)) I) \psi$

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 $\Delta: \Delta \sigma (i + 1) \leq \text{right } I$ **and now**: $\langle \sigma, v, i \rangle \models \varphi$ **and**

next: $\langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (\text{subtract } (\Delta \sigma (i + 1)) I) \psi$

from *next* **obtain** j **where** $j: i + 1 \leq j \text{ mem } (\tau \sigma j - \tau \sigma (i + 1)) (\text{subtract } (\Delta \sigma (i + 1)) I)$

$\langle \sigma, v, j \rangle \models \psi \forall k \in \{i + 1 .. < j\}. \langle \sigma, v, k \rangle \models \varphi$

by *auto*

from $\Delta j(1,2)$ **have** $\text{mem } (\tau \sigma j - \tau \sigma i) I$

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*: $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi \longleftrightarrow$

$\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \psi \vee$

$(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (\text{subtract } (\Delta \sigma i) I) \psi)$

(is ?L \longleftrightarrow ?R)

proof (rule iffI; (elim disjE conjE)?)

assume ?L

then obtain *j* **where** $j: j \leq i \text{ mem } (\tau \sigma i - \tau \sigma j) I \langle \sigma, v, j \rangle \models \psi \forall k \in \{j <.. i\}. \langle \sigma, v, k \rangle \models \varphi$

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** $\Delta \sigma i \leq \text{right } I$

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** $\langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (\text{subtract } (\Delta \sigma i) I) \psi$

by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])

ultimately show ?thesis **by** auto

qed simp

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: $\langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (\text{subtract } (\Delta \sigma i) I) \psi$

from *prev* **obtain** *j* **where** $j: j \leq i - 1 \text{ mem } (\tau \sigma (i - 1) - \tau \sigma j) ((\text{subtract } (\Delta \sigma i) I))$

$\langle \sigma, v, j \rangle \models \psi \forall k \in \{j <.. i - 1\}. \langle \sigma, v, k \rangle \models \varphi$

by auto

from Δ $i j(1,2)$ **have** $\text{mem } (\tau \sigma i - \tau \sigma j) I$

by (cases right I) (auto simp: le_diff_conv2)

with now $i j(1,3,4)$ **show** ?L **by** (auto simp: le_Suc_eq gr0_conv_Suc intro!: exI[of _ j])

qed auto

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

by auto

lemma *sat_Since_point*: $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi \implies$

$(\bigwedge j. j \leq i \implies \text{mem } (\tau \sigma i - \tau \sigma j) I \implies \langle \sigma, v, i \rangle \models \varphi \mathbf{S} (\text{point } (\tau \sigma i - \tau \sigma j)) \psi \implies P) \implies P$

by (auto intro: diff_le_self)

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

by auto

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

by auto

lemma *sat_Once_rec*: $\langle \sigma, v, i \rangle \models \mathbf{P} I \varphi \longleftrightarrow$

$\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$

$(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \langle \sigma, v, i - 1 \rangle \models \mathbf{P} (\text{subtract } (\Delta \sigma i) I) \varphi)$

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*: $\langle \sigma, v, i \rangle \models \mathbf{F} I \varphi = \langle \sigma, v, i \rangle \models \top \mathbf{U} I \varphi$
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_syntax*
begin

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

notation *Var* ($\langle \mathbf{v} \rangle$)
and *Const* ($\langle \mathbf{c} \rangle$)

For subscripts type “backslash” followed by “sub”

notation *TT* ($\langle \top \rangle$)
and *FF* ($\langle \perp \rangle$)
and *Pred* ($\langle _ \dagger _ \rangle$ [85, 85] 85)
and *Eq_Const* ($\langle _ \approx _ \rangle$ [85, 85] 85)
and *Neg* ($\langle \neg_F _ \rangle$ [82] 82)
and *And* (**infixr** $\langle \wedge_F \rangle$ 80)
and *Or* (**infixr** $\langle \vee_F \rangle$ 80)
and *Imp* (**infixr** $\langle \longrightarrow_F \rangle$ 79)
and *Iff* (**infixr** $\langle \longleftrightarrow_F \rangle$ 79)
and *Exists* ($\langle \exists_F _ _ \rangle$ [70, 70] 70)
and *Forall* ($\langle \forall_F _ _ \rangle$ [70, 70] 70)
and *Prev* ($\langle \mathbf{Y} _ _ \rangle$ [1000, 65] 65)
and *Next* ($\langle \mathbf{X} _ _ \rangle$ [1000, 65] 65)
and *Once* ($\langle \mathbf{P} _ _ \rangle$ [1000, 65] 65)
and *Eventually* ($\langle \mathbf{F} _ _ \rangle$ [1000, 65] 65)
and *Historically* ($\langle \mathbf{H} _ _ \rangle$ [1000, 65] 65)
and *Always* ($\langle \mathbf{G} _ _ \rangle$ [1000, 65] 65)
and *Since* ($\langle _ \mathbf{S} _ _ \rangle$ [60, 1000, 60] 60)
and *Until* ($\langle _ \mathbf{U} _ _ \rangle$ [60, 1000, 60] 60)

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

end

unbundle no MFOTL_syntax

4 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 $\text{disjoint } (\text{fst } ' \text{ set } xs) \wedge \text{distinct } (\text{map } \text{fst } xs)$

proof –

from *assms*(1) have $\text{disjoint } (\text{fst } ' \text{ set } xs)$

by (*metis disjoint_def in_set_conv_nth pairwise_imageI*)

moreover have $\text{distinct } (\text{map } \text{fst } xs)$

using *assms*

by (*smt (verit) distinct_conv_nth image_iff inf.idem*
 $\text{length_map } \text{nth_map } \text{nth_mem}$)

ultimately show *?thesis*

by *blast*

qed

lemma part_list_set_eq_aux2:

assumes

$\text{disjoint } (\text{fst } ' \text{ set } xs)$

$\text{distinct } (\text{map } \text{fst } xs)$

$i < \text{length } xs$

$j < \text{length } xs$

$i \neq j$

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:

$(\bigcup X \in \text{fst } ' \text{ set } xs. X) = \text{UNIV}$

$\wedge (\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j$

$\longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\}) \wedge \{\} \notin \text{fst } ' \text{ set } xs$

$\longleftrightarrow \text{partition_on } \text{UNIV } (\text{set } (\text{map } \text{fst } xs)) \wedge \text{distinct } (\text{map } \text{fst } xs)$

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)

4.1 size setup

lift_definition subs :: ('d, 'a) part \Rightarrow 'd set list is map fst .

lift_definition Subs :: ('d, 'a) part \Rightarrow 'd set set is set o map fst .

lift_definition vals :: ('d, 'a) part \Rightarrow 'a list is map snd .

lift_definition SubsVals :: ('d, 'a) part \Rightarrow ('d set \times 'a) set is set .

lift_definition subsvals :: ('d, 'a) part \Rightarrow ('d set \times 'a) list is id .

lift_definition size_part :: ('d \Rightarrow nat) \Rightarrow ('a \Rightarrow nat) \Rightarrow ('d, 'a) part \Rightarrow nat is $\lambda f g. \text{size_list } (\lambda(x, y). \text{sum } f x + g y)$.

instantiation part :: (type, type) size begin

definition size_part where

size_part_overloaded_def: size_part = Partition.size_part ($\lambda_. 0$) ($\lambda_. 0$)

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 \Longrightarrow size_part f g \circ map_part h = size_part f (g \circ h)

by (rule ext, transfer)

(auto simp: fun_eq_iff map_prod_def o_def split_beta case_prod_beta[abs_def])

setup \langle

BNF_LFP_Size.register_size_global **type_name** \langle part \rangle **const_name** \langle size_part \rangle

@{thm size_part_overloaded_def} @ {thms size_part_def size_part_overloaded_simps}

@ {thms part_size_o_map}

\rangle

lemma is_measure_size_part[measure_function]: is_measure f \Longrightarrow is_measure g \Longrightarrow is_measure (size_part f g)

by (rule is_measure_trivial)

lemma size_part_estimation[termination_simp]: $x \in \text{Vals } xs \Longrightarrow y < g x \Longrightarrow y < \text{size_part } f g xs$

by transfer (auto simp: termination_simp)

lemma size_part_estimation'[termination_simp]: $x \in \text{Vals } xs \Longrightarrow y \leq g x \Longrightarrow y \leq \text{size_part } f g xs$

by transfer (auto simp: termination_simp)

lemma size_part_pointwise[termination_simp]: $(\bigwedge x. x \in \text{Vals } xs \Longrightarrow f x \leq g x) \Longrightarrow \text{size_part } h f xs \leq \text{size_part } h g xs$

by transfer (force simp: image_iff intro!: size_list_pointwise)

4.2 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*]: *rel_fun* (*pcr_part* (=) (=)) (=) (*set* \circ *map_snd*) *Vals*
by (*force_simp*: *Vals_def* *rel_fun_def* *pcr_part_def* *cr_part_def* *rel_set_eq* *prod.rel_eq* *list.rel_eq*)

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

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

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

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

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

lemma *lookup_part_tabulate*[*simp*]: *lookup_part* (*tabulate xs f z*) *x* =
if *distinct xs* \wedge $x \in$ *set xs* *then f x* *else z*
by (*transfer_fixing*: *x xs f z*)
(*auto_simp*: *find_dropWhile* *dropWhile_eq* *Cons_conv* *map_eq* *append_conv* *split*: *list.splits*)

lemma *part_hd_Vals*[*simp*]: *part_hd part* \in *Vals part*
by *transfer* (*auto_simp*: *partition_on_def* *image_iff* *intro!*: *hd_in_set*)

lemma *lookup_part_Vals*[*simp*]: *lookup_part part d* \in *Vals 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 *set xs* *d* \in *D*

unfolding *partition_on_def*

by *fastforce*

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

unfolding *set_eq_iff*

by (*auto_simp*: *find_Some_iff* *in_set_conv_nth* *split_beta*)

with D show *?case*

by (*force_simp*: *image_iff*)

qed

lemma *lookup_part_Subsvals*: $\exists D$. $d \in D \wedge (D, \text{lookup_part part } d) \in \text{Subsvals part}$

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*) \in *set xs* *d* \in *D*

unfolding *partition_on_def*

by *fastforce*

with unique have *find* ($\lambda(D, _)$. $d \in D$) *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*: $(D, e) \in \text{set (subvals part)} \implies d \in D \implies \text{lookup_part part } d = e$
proof (*transfer fixing: D e d, goal_cases*)
case (1 part)
then show ?case
proof (*cases find* $(\lambda(D, _). d \in D)$ 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]*: $\text{size (lookup_part part } d) < \text{Suc (size_part } (\lambda_ . 0) \text{ size part)}$
unfolding *less_Suc_eq_le*
by (*rule size_part_estimation'[OF _ order_refl]*) *simp*

lemma *subvals_part_estimation[termination_simp]*: $(D, e) \in \text{set (subvals part)} \implies \text{size } e < \text{Suc (size_part } (\lambda_ . 0) \text{ size part)}$
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]*: $\text{size (part_hd part)} < \text{Suc (size_part } (\lambda_ . 0) \text{ size part)}$
unfolding *less_Suc_eq_le*
by (*rule size_part_estimation'[OF _ order_refl]*) *simp*

lemma *size_last_estimation[termination_simp]*: $xs \neq [] \implies \text{size (last } xs) < \text{size_list size } xs$
by (*induct xs*) *auto*

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

lemma *snd_lookup[simp]*: $\text{snd (lookup part } d) = \text{lookup_part part } d$
by *transfer auto*

lemma *distinct_disjoint_uniq*: $\text{distinct } xs \implies \text{disjoint (set } xs) \implies i < j \implies j < \text{length } xs \implies d \in xs ! i \implies d \in xs ! j \implies \text{False}$
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*:
 $\text{partition_on UNIV (set (map fst part))} \implies \text{distinct (map fst part)} \implies \exists y. \text{find } (\lambda(D, _). d \in D) \text{ part} = \text{Some } y$
unfolding *partition_on_def set_eq_iff*
by (*auto simp: find_Some_iff in_set_conv_nth Ball_def image_iff Bex_def split_beta dest: distinct_disjoint_uniq dest!: spec[of _ d] intro!: exI[where P= $\lambda x. \exists y z. P x y z \wedge Q x y z$ for P Q, OF exI, OF exI, OF conjI]*)

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

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

lift_definition *trivial_part* :: *'pt ⇒ ('d, 'pt) part is λpt. [(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*

5 Partitioned Decision Trees

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

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

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

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

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

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

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

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

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

context begin

qualified inductive

SAT :: $(\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ Regex.regex} \Rightarrow \text{bool}$
for *sat* **where**
STest: $i = j \implies \text{sat } i \ x \implies \text{SAT } sat \ i \ j \ (\text{Regex.Test } x)$
| *SSkip*: $j = i + n \implies \text{SAT } sat \ i \ j \ (\text{Regex.Skip } n)$
| *SPlusL*: $\text{SAT } sat \ i \ j \ r \implies \text{SAT } sat \ i \ j \ (\text{Regex.Plus } r \ s)$
| *SPlusR*: $\text{SAT } sat \ i \ j \ s \implies \text{SAT } sat \ i \ j \ (\text{Regex.Plus } r \ s)$
| *STimes*: $\text{SAT } sat \ i \ k \ r \implies \text{SAT } sat \ k \ j \ s \implies \text{SAT } sat \ i \ j \ (\text{Regex.Times } r \ s)$
| *SStar_eps*: $i = j \implies \text{SAT } sat \ i \ j \ (\text{Regex.Star } r)$
| *SStar*: $i < j \implies (\exists \text{zs. } xs = i \ \# \ \text{zs} \ @ \ [j]) \implies$
 $\forall k \in \{0 \dots \text{length } xs - 1\}. xs ! \ k < xs ! \ (\text{Suc } k) \implies$
 $\forall k \in \{0 \dots \text{length } xs - 1\}. \text{SAT } sat \ (xs ! \ k) \ (xs ! \ (\text{Suc } k)) \ r \implies$
 $\text{SAT } sat \ i \ j \ (\text{Regex.Star } r)$

lemma *SAT_mono*[*mono*]:

assumes $X \leq Y$
shows $\text{SAT } X \leq \text{SAT } Y$
unfolding *le_fun_def le_bool_def*

proof *safe*

fix $i \ j \ r$
assume $\text{SAT } X \ i \ j \ r$
then show $\text{SAT } Y \ i \ j \ r$
by (*induct i j r rule: SAT.induct*) (*use assms in <auto 0 3 intro: SAT.intros>*)
qed

abbreviation $rm \ S \equiv \{(i, j) \in S. i < j\}$

qualified inductive

VIO :: $(\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ Regex.regex} \Rightarrow \text{bool}$
for *vio* **where**
VSkip: $j \neq i + n \implies \text{VIO } vio \ i \ j \ (\text{Regex.Skip } n)$
| *VTest*: $i = j \implies \text{vio } i \ x \implies \text{VIO } vio \ i \ j \ (\text{Regex.Test } x)$
| *VTest_neq*: $i \neq j \implies \text{VIO } vio \ i \ j \ (\text{Regex.Test } x)$
| *VPlus*: $\text{VIO } vio \ i \ j \ r \implies \text{VIO } vio \ i \ j \ s \implies \text{VIO } vio \ i \ j \ (\text{Regex.Plus } r \ s)$
| *VTimes*: $\forall k \in \{i \dots j\}. \text{VIO } vio \ i \ k \ r \ \vee \ \text{VIO } vio \ k \ j \ s \implies \text{VIO } vio \ i \ j \ (\text{Regex.Times } r \ s)$
| *VStar*: $i < j \implies i \in S \implies j \in T \implies S \cup T = \{i \dots j\} \implies S \cap T = \{\} \implies$
 $\forall (s, t) \in rm \ (S \times T). \text{VIO } vio \ s \ t \ r \implies \text{VIO } vio \ i \ j \ (\text{Regex.Star } r)$
| *VStar_gt*: $i > j \implies \text{VIO } vio \ i \ j \ (\text{Regex.Star } r)$

lemma *VIO_mono*[*mono*]:

assumes $X \leq Y$
shows $\text{VIO } X \leq \text{VIO } Y$
unfolding *le_fun_def le_bool_def*

proof *safe*

fix $i \ j \ r$
assume $\text{VIO } X \ i \ j \ r$
then show $\text{VIO } Y \ i \ j \ r$
by (*induct i j r rule: VIO.induct*) (*use assms in <auto 5 3 intro: VIO.intros>*)

qed

inductive *chain* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool **for** *R* :: 'a ⇒ 'a ⇒ bool **where**
 chain_singleton: *chain R* [x]
 | *chain_cons*: *chain R* (y # xs) ⇒ R x y ⇒ *chain R* (x # y # xs)

lemma

chain_Nil[simp]: ¬ *chain R* [] **and**
 chain_not_Nil: *chain R* xs ⇒ xs ≠ []
 by (auto elim: *chain.cases*)

lemma *chain_rtranclp*: *chain R* xs ⇒ R** (hd xs) (last xs)
 by (induct xs rule: *chain.induct*) auto

lemma *chain_append*:

assumes *chain R* xs *chain R* ys *R* (last xs) (hd ys)
 shows *chain R* (xs @ ys)
 using *assms*

proof (induct xs arbitrary: ys rule: *chain.induct*)

case (*chain_singleton* x)

then show ?case **by** (cases ys) (auto intro: *chain.intros*)

qed (auto intro: *chain.intros*)

lemma *tranclp_imp_exists_finite_chain_list*:

 R⁺⁺ x y ⇒ ∃ xs. *chain R* (x # xs @ [y])

proof (induct rule: *tranclp.induct*)

case (*r_into_trancl* x y)

then have *chain R* (x # [] @ [y])

by (auto intro: *chain.intros*)

then show ?case

by *blast*

next

case (*trancl_into_trancl* x y z)

note *rstar_xy* = *this(1)* **and** *ih* = *this(2)* **and** *r_yz* = *this(3)*

obtain xs **where** xs: *chain R* (x # xs @ [y]) **using** *ih* **by** *blast*

define ys **where** ys = xs @ [y]

have *chain R* (x # ys @ [z])

unfolding *ys_def* **using** *r_yz* *chain_append*[OF xs *chain_singleton*, of z] **by** auto

then show ?case **by** *blast*

qed

lemma *chain_pairwise*:

chain R xs ⇒ Suc i < length xs ⇒ R (xs ! i) (xs ! Suc i)

by (induct xs arbitrary: i rule: *chain.induct*)

 (force *simp*: *nth_Cons'* *not_le* *Suc_less_eq2* elim: *chain.cases*)+

lemma *chain_sorted_remdups*:

chain R xs ⇒ (∧ x y. R x y ⇒ x ≤ y) ⇒ sorted xs ∧ *chain R* (remdups xs)

proof (induct xs rule: *chain.induct*)

case (*chain_cons* y xs x)

then show ?case

using *sorted_remdups*[of xs] *set_remdups*[of xs] *eq_iff*[of y hd (remdups xs)]

by (cases *remdups* xs; cases y = hd (remdups xs))

 (auto intro!: *chain.intros* intro: *order_trans* elim: *chain.cases*)

qed (auto intro: *chain.intros*)

lemma *sorted_remdups*: sorted xs ⇒ sorted_wrt (<) (remdups xs)

by (induct xs) (auto dest: *le_neq_trans*)

lemma *remdups_sorted_start_end*:
sorted (i # xs @ [j]) \implies i \neq j \implies
remdups (i # xs @ [j]) = i # remdups (removeAll j (removeAll i xs)) @ [j]
by (*induct xs*) *auto*

lemma *tranclp_to_list*:
fixes *R :: 'a :: linorder \Rightarrow 'a \Rightarrow bool*
assumes *R⁺⁺ i j i \neq j \wedge x y. R x y \implies x \leq y*
obtains *xs zs where xs = i # zs @ [j]*
 $\forall k \in \{0 ..< \text{length } xs - 1\}. xs ! k < xs ! (\text{Suc } k) \wedge R (xs ! k) (xs ! (\text{Suc } k))$

proof *atomize_elim*

from $\langle R^{++} i j \rangle$ **obtain** *zs where chain R (i # zs @ [j])*
using *tranclp_imp_exists_finite_chain_list* **by** *fast*
then have *zs: sorted (i # zs @ [j]) chain R (remdups (i # zs @ [j]))*
using *chain_sorted_remdups* *assms(3)* **by** *blast+*
then have *sorted_wrt: sorted_wrt (<) (remdups (i # zs @ [j]))*
using *sorted_remdups* **by** *blast*
let *?zs = remdups (removeAll j (removeAll i zs))*
from *zs sorted_wrt* **have** *chain R (i # ?zs @ [j]) sorted_wrt (<) (i # ?zs @ [j])*
using *remdups_sorted_start_end[of i zs j]* *assms(2)* **by** *auto*
then show $\exists xs zs. xs = i \# zs @ [j] \wedge$
 $(\forall k \in \{0..<\text{length } xs - 1\}. xs ! k < xs ! \text{Suc } k \wedge R (xs ! k) (xs ! \text{Suc } k))$
by (*subst ex_comm, unfold simp_thms, intro exI[of _ ?zs]*)
(auto 0 3 dest: chain_pairwise simp del: remdups.simps
simp: sorted_wrt_iff_nth_less)

qed

abbreviation *match_rel* **where**

match_rel test r xs k \equiv (xs ! k < xs ! (Suc k) \wedge Regex.match test r (xs ! k) (xs ! (Suc k)))

lemma *list_to_chain*:

xs \neq [] \implies $\forall k \in \{0 ..< \text{length } xs - 1\}. R (xs ! k) (xs ! \text{Suc } k) \implies \text{chain } R \text{ } xs$

proof (*induct xs*)

case (*Cons a xs*)

then show *?case*

proof (*cases xs*)

case *tail: (Cons b ys)*

with *Cons(2,3)* **show** *?thesis*

by (*force intro!: chain.intros Cons(1)[unfolded tail]*)

qed (*auto intro: chain.intros*)

qed *simp*

lemma *match_rel_list_to_tranclp*:

$\exists xs zs. xs = i \# zs @ [j] \wedge (\forall k \in \{0 ..< \text{length } xs - 1\}. \text{match_rel test } r \text{ } xs \ k) \implies i \neq j \implies$
 $(\text{Regex.match test } r)^{++} i \ j$

using *chain_rtranclp[OF list_to_chain, THEN rtranclpD, of i # _ @ [j] Regex.match test r]*
by *fastforce*

lemma *completeness_SAT*:

$\forall x \in \text{Regex.atms } r. \forall i. \text{test } i \ x \longrightarrow \text{sat } i \ x \implies \text{Regex.match test } r \ i \ j \implies \text{SAT sat } i \ j \ r$

proof (*induct r arbitrary: i j*)

case (*Skip x*)

then show *?case*

by (*auto intro: SAT.SSkip*)

next

case (*Test x*)

```

then show ?case
  by (auto intro: SAT.STest)
next
  case (Plus r1 r2)
  then show ?case
    by (auto intro: SAT.SPlusL SPlusR)
next
  case (Times r1 r2)
  then obtain k where k_def: k ∈ {i .. j} ∧ SAT sat i k r1 ∧ SAT sat k j r2
    using match_le by fastforce
  then show ?case by (auto intro: SAT.STimes)
next
  case (Star r)
  then have i = j ∨ (i ≠ j ∧ (Regex.match test r)++ i j)
    using rtranclpD[of Regex.match test r i j] tranclpD[of Regex.match test r i j]
    by auto
  moreover
  {
    assume i_eq_j: i = j
    then have SAT sat i j (Regex.Star r) by (auto intro: SAT.SStar_eps)
  }
  moreover
  {
    assume i_neq_j: i ≠ j and tranclp_ij: (Regex.match test r)++ i j
    then have i_less: i < j using Star
      by (auto simp add: le_neq_implies_less match_rtranclp_le)
    then obtain xs and zs where xs_def: xs = i # zs @ [j] ∧ (∀ k ∈ {0 ..< length xs - 1}. xs ! k < xs
! (Suc k) ∧ Regex.match test r (xs ! k) (xs ! (Suc k)))
      using tranclp_to_list[OF tranclp_ij i_neq_j match_le[of test r]] by auto
    then have SAT sat i j (Regex.Star r) using i_less Star
      by (auto intro: SAT.SStar)
  }
  ultimately show ?case by auto
qed

```

lemma completeness_VIO:

$\forall x \in \text{Regex.atms } r. \forall i. \neg \text{test } i x \longrightarrow \text{vio } i x \Longrightarrow i \leq j \Longrightarrow \neg \text{Regex.match test } r i j \Longrightarrow \text{VIO vio } i j r$

proof (induct r arbitrary: i j)

```

case (Skip x)
  then show ?case
    by (auto intro: VIO.VSkip)
next
  case (Test x)
  then show ?case
    by (auto intro: VIO.VTest VIO.VTest_neq)
next
  case (Plus r1 r2)
  then show ?case
    by (auto intro: VIO.VPlus)
next
  case (Times r1 r2)
  then have  $\forall k \in \{i .. j\}. \text{VIO vio } i k r1 \vee \text{VIO vio } k j r2$ 
    by fastforce
  then show ?case
    by (auto intro: VIO.VTimes)
next
  case (Star r)
  define V where V_def: V = {i .. j}

```

```

define S where S_def: S = {k ∈ V. (Regex.match test r)** i k} ∪ {i}
define T where T_def: T = V - S
from S_def V_def have j_notin_S: j ∉ S using Star
  by auto
from S_def have i_in_S: i ∈ S
  by auto
then have j_in_T: j ∈ T using j_notin_S V_def T_def Star(3)
  by auto
from Star have nmatch_ij: ¬ (Regex.match test r)** i j
  by auto
from S_def T_def V_def Star(3) have union_ST: S ∪ T = {i .. j}
  by auto
from S_def T_def V_def Star(4) have inter_ST: S ∩ T = {}
  by auto
with i_in_S j_in_T Star(3) have i_less_j: i < j
  using le_eq_less_or_eq by blast
{
  assume not_vioست: ¬ (∀(s,t) ∈ rm (S × T). VIO vio s t r)
  then obtain s and t where st_def: (s, t) ∈ rm (S × T) ∧ ¬ VIO vio s t r
    by auto
  then have Regex.match test r s t using Star
    by auto
  then have (Regex.match test r)** i t using st_def S_def
    by auto
  then have False using T_def st_def S_def
    by auto
}
then have no_path: ∀(s,t) ∈ rm (S × T). VIO vio s t r
  by auto
then show ?case
  by (auto intro: VIO.VStar[OF i_less_j i_in_S j_in_T union_ST inter_ST no_path])
qed

```

lemma soundness_SAT:

```

∀ x ∈ Regex.atms r. ∀ i. sat i x → test i x ⇒ SAT sat i j r ⇒ Regex.match test r i j
proof(induct r arbitrary: i j)
  case (Skip x)
  then show ?case using SAT.simps[of sat i j Regex.Skip x]
    by auto
next
  case (Test x)
  then show ?case using SAT.simps[of sat i j Regex.Test x]
    by auto
next
  case (Plus r1 r2)
  then show ?case using SAT.simps[of sat i j Regex.Plus r1 r2]
    by auto
next
  case (Times r1 r2)
  then show ?case using SAT.simps[of sat i j Regex.Times r1 r2]
    by fastforce
next
  case (Star r)
  then show ?case
  proof(cases i = j)
    case True
    then show ?thesis
      by auto

```

```

next
  case False
  then obtain xs and zs where xs_form: xs = i # zs @ [j] and
    xs_props:  $\forall k \in \{0 \dots \text{length } xs - 1\}. xs ! k < xs ! (\text{Suc } k) \wedge SAT \text{ sat } (xs ! k) (xs ! (\text{Suc } k)) r$ 
    using Star(3) SAT.simps[of sat i j Regex.Star r]
    by blast
  then have kmatch:  $\forall k \in \{0 \dots \text{length } xs - 1\}. Regex.\text{match test } r (xs ! k) (xs ! (\text{Suc } k))$ 
    using Star
    by auto
  then have ex_lists:  $\exists xs \ zs. xs = i \# zs @ [j] \wedge (\forall k \in \{0 \dots \text{length } xs - 1\}. xs ! k < xs ! (\text{Suc } k) \wedge Regex.\text{match test } r (xs ! k) (xs ! (\text{Suc } k)))$ 
    using xs_form xs_props by auto
  then have  $(Regex.\text{match test } r)^{++} i j$ 
    using match_rel_list_to_tranclp[OF ex_lists False] by auto
  then show ?thesis
    by auto
qed
qed

lemma soundness_VIO:
 $\forall x \in Regex.\text{atms } r. \forall i. vio \ i \ x \longrightarrow \neg \text{test } i \ x \Longrightarrow i \leq j \Longrightarrow VIO \ vio \ i \ j \ r \Longrightarrow \neg Regex.\text{match test } r \ i \ j$ 
proof(induct r arbitrary: i j)
  case (Skip x)
  then show ?case using VIO.simps[of vio i j Regex.Skip x]
    by auto
next
  case (Test x)
  then show ?case using VIO.simps[of vio i j Regex.Test x]
    by auto
next
  case (Plus r1 r2)
  then show ?case using VIO.simps[of vio i j Regex.Plus r1 r2]
    by auto
next
  case (Times r1 r2)
  then have kvio:  $\forall k \in \{i \dots j\}. VIO \ vio \ i \ k \ r1 \vee VIO \ vio \ k \ j \ r2$ 
    using VIO.simps[of vio i j Regex.Times r1 r2]
    by auto
  have  $\forall k. Regex.\text{match test } r \ i \ k \wedge Regex.\text{match test } r \ k \ j \longrightarrow k \in \{i \dots j\}$ 
    using match_le
    by auto
  then show ?case using Times kvio match_le[of test]
    unfolding Ball_def atLeastAtMost_iff match.simps regex.simps relcompp_apply
    by (metis Un_iff)
next
  case (Star r)
  then obtain S and T where S_def: i ∈ S and T_def: j ∈ T and
    ST_props:  $S \cup T = \{i \dots j\} \wedge S \cap T = \{\}$  and
    st_vio:  $\forall (s,t) \in rm \ (S \times T). VIO \ vio \ s \ t \ r$ 
    using Star(4) VIO.simps[of vio i j Regex.Star r]
    by auto
  then have nmatch_st:  $\forall (s,t) \in rm \ (S \times T). \neg Regex.\text{match test } r \ s \ t$ 
    using Star
    by auto
  from S_def T_def ST_props have i_neq_j: i ≠ j
    by auto
  {
    assume rtranclp_ij:  $(Regex.\text{match test } r)^{**} i j$ 

```

```

then have trancplp_ij: (Regex.match test r)++ i j
  using rtrancplpD[of Regex.match test r i j] i_neq_j
  by auto

obtain xs and zs where xs_def: xs = i # zs @ [j] and
xs_prop:  $\forall k \in \{0 \dots \text{length } xs - 1\}$ . match_rel test r xs k
  using trancplp_to_list[OF trancplp_ij i_neq_j match_le[of test r]]
  by auto

with S_def T_def ST_props have  $\exists k \in \{0 \dots \text{length } xs - 1\}$ . (xs ! k)  $\in S \wedge$  (xs ! (Suc k))  $\in T$ 
proof (induction zs arbitrary: S T i j xs)
  case Nil
  then show ?case using S_def T_def xs_def
  by auto
next
  case (Cons a zs')
  from Cons(2-6) have match: Regex.match test r i a
  by force
  show ?case
  proof (cases a \in T)
  case True
  then have xs ! 0 \in S \wedge xs ! Suc 0 \in T using S_def Cons
  by (auto simp: xs_def)
  then show ?thesis using S_def Cons(1)[of _ _ _ _ xs] Cons(2-5)
  by force
  next
  case False
  from Cons(5,6) have chain (<) (i # (a # zs') @ [j])
  by (intro list_to_chain) auto
  then have sorted (i # (a # zs') @ [j])
  using chain_sorted_remdups[of (<) i # (a # zs') @ [j]]
  by auto
  then have a \in \{i .. j\}
  by auto
  with Cons(2-6) False have  $\exists k \in \{0 \dots \text{length } (tl xs) - 1\}$ . tl xs ! k \in \{i \in S. a \leq i\} \wedge tl xs ! Suc
k \in \{i \in T. a \leq i\}
  by (intro Cons(1)[of a _ j]) (auto dest: bspec[of _ _ Suc _])
  with Cons show ?thesis
  by (auto intro: beXI[of _ Suc _])
  qed
  qed
  then have False using nmatch_st xs_prop
  by auto
}
then show ?case
  by auto
qed
end

```

6 Proof System

unbundle *MFOTL_syntax*

context begin

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

$STT: SAT \sigma v i TT$
 $VFF: VIO \sigma v i FF$
 $SPred: (r, eval_trms v ts) \in \Gamma \sigma i \implies SAT \sigma v i (Pred r ts)$
 $VPred: (r, eval_trms v ts) \notin \Gamma \sigma i \implies VIO \sigma v i (Pred r ts)$
 $SEq_Const: v x = c \implies SAT \sigma v i (Eq_Const x c)$
 $VEq_Const: v x \neq c \implies VIO \sigma v i (Eq_Const x c)$
 $SNeg: VIO \sigma v i \varphi \implies SAT \sigma v i (Neg \varphi)$
 $VNeg: SAT \sigma v i \varphi \implies VIO \sigma v i (Neg \varphi)$
 $SOrL: SAT \sigma v i \varphi \implies SAT \sigma v i (Or \varphi \psi)$
 $SOrR: SAT \sigma v i \psi \implies SAT \sigma v i (Or \varphi \psi)$
 $VOr: VIO \sigma v i \varphi \implies VIO \sigma v i \psi \implies VIO \sigma v i (Or \varphi \psi)$
 $SAnd: SAT \sigma v i \varphi \implies SAT \sigma v i \psi \implies SAT \sigma v i (And \varphi \psi)$
 $VAndL: VIO \sigma v i \varphi \implies VIO \sigma v i (And \varphi \psi)$
 $VAndR: VIO \sigma v i \psi \implies VIO \sigma v i (And \varphi \psi)$
 $SImpL: VIO \sigma v i \varphi \implies SAT \sigma v i (Imp \varphi \psi)$
 $SImpR: SAT \sigma v i \psi \implies SAT \sigma v i (Imp \varphi \psi)$
 $VImp: SAT \sigma v i \varphi \implies VIO \sigma v i \psi \implies VIO \sigma v i (Imp \varphi \psi)$
 $SIffSS: SAT \sigma v i \varphi \implies SAT \sigma v i \psi \implies SAT \sigma v i (Iff \varphi \psi)$
 $SIffVV: VIO \sigma v i \varphi \implies VIO \sigma v i \psi \implies SAT \sigma v i (Iff \varphi \psi)$
 $VIffSV: SAT \sigma v i \varphi \implies VIO \sigma v i \psi \implies VIO \sigma v i (Iff \varphi \psi)$
 $VIffVS: VIO \sigma v i \varphi \implies SAT \sigma v i \psi \implies VIO \sigma v i (Iff \varphi \psi)$
 $SExists: \exists z. SAT \sigma (v (x := z)) i \varphi \implies SAT \sigma v i (Exists x \varphi)$
 $VExists: \forall z. VIO \sigma (v (x := z)) i \varphi \implies VIO \sigma v i (Exists x \varphi)$
 $SForall: \forall z. SAT \sigma (v (x := z)) i \varphi \implies SAT \sigma v i (Forall x \varphi)$
 $VForall: \exists z. VIO \sigma (v (x := z)) i \varphi \implies VIO \sigma v i (Forall x \varphi)$
 $SPrev: i > 0 \implies mem (\Delta \sigma i) I \implies SAT \sigma v (i-1) \varphi \implies SAT \sigma v i (\mathbf{Y} I \varphi)$
 $VPrev: i > 0 \implies VIO \sigma v (i-1) \varphi \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $VPrevZ: i = 0 \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $VPrevOutL: i > 0 \implies (\Delta \sigma i) < (left I) \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $VPrevOutR: i > 0 \implies enat (\Delta \sigma i) > (right I) \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $SNext: mem (\Delta \sigma (i+1)) I \implies SAT \sigma v (i+1) \varphi \implies SAT \sigma v i (\mathbf{X} I \varphi)$
 $VNext: VIO \sigma v (i+1) \varphi \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $VNextOutL: (\Delta \sigma (i+1)) < (left I) \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $VNextOutR: enat (\Delta \sigma (i+1)) > (right I) \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $SOnce: j \leq i \implies mem (\delta \sigma i j) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{P} I \varphi)$
 $VOnceOut: \tau \sigma i < \tau \sigma 0 + left I \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $VOnce: j = (case right I of \infty \Rightarrow 0$
 $\quad | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + left I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \varphi) \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $SEventually: j \geq i \implies mem (\delta \sigma j i) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{F} I \varphi)$
 $VEventually: (\bigwedge k. k \in (case right I of \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | enat b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies VIO \sigma v k \varphi) \implies$
 $\quad VIO \sigma v i (\mathbf{F} I \varphi)$
 $SHistorically: j = (case right I of \infty \Rightarrow 0$
 $\quad | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + left I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies SAT \sigma v k \varphi) \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $SHistoricallyOut: \tau \sigma i < \tau \sigma 0 + left I \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $VHistorically: j \leq i \implies mem (\delta \sigma i j) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{H} I \varphi)$
 $SAlways: (\bigwedge k. k \in (case right I of \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | enat b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies SAT \sigma v k \varphi) \implies$
 $\quad SAT \sigma v i (\mathbf{G} I \varphi)$
 $VAlways: j \geq i \implies mem (\delta \sigma j i) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{G} I \varphi)$
 $SSince: j \leq i \implies mem (\delta \sigma i j) I \implies SAT \sigma v j \psi \implies (\bigwedge k. k \in \{j <.. i\} \implies$
 $\quad SAT \sigma v k \varphi) \implies SAT \sigma v i (\varphi \mathbf{S} I \psi)$
 $VSinceOut: \tau \sigma i < \tau \sigma 0 + left I \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $VSince: (case right I of \infty \Rightarrow True$

$| \text{enat } b \Rightarrow \text{ETP } \sigma ((\tau \sigma i) - b) \leq j \Rightarrow$
 $j \leq i \Rightarrow (\tau \sigma 0) + \text{left } I \leq (\tau \sigma i) \Rightarrow \text{VIO } \sigma v j \varphi \Rightarrow$
 $(\bigwedge k. k \in \{j .. \text{LTP_p } \sigma i I\} \Rightarrow \text{VIO } \sigma v k \psi) \Rightarrow \text{VIO } \sigma v i (\varphi \mathbf{S} I \psi)$
 $| \text{VSinceInf}: j = (\text{case right } I \text{ of } \infty \Rightarrow 0$
 $| \text{enat } b \Rightarrow \text{ETP_p } \sigma i b) \Rightarrow$
 $(\tau \sigma i) \geq (\tau \sigma 0) + \text{left } I \Rightarrow$
 $(\bigwedge k. k \in \{j .. \text{LTP_p } \sigma i I\} \Rightarrow \text{VIO } \sigma v k \psi) \Rightarrow \text{VIO } \sigma v i (\varphi \mathbf{S} I \psi)$
 $| \text{SUntil}: j \geq i \Rightarrow \text{mem } (\delta \sigma j i) I \Rightarrow \text{SAT } \sigma v j \psi \Rightarrow (\bigwedge k. k \in \{i .. < j\} \Rightarrow \text{SAT } \sigma v k \varphi) \Rightarrow$
 $\text{SAT } \sigma v i (\varphi \mathbf{U} I \psi)$
 $| \text{VUntil}: (\text{case right } I \text{ of } \infty \Rightarrow \text{True}$
 $| \text{enat } b \Rightarrow j < \text{LTP_f } \sigma i b) \Rightarrow$
 $j \geq i \Rightarrow \text{VIO } \sigma v j \varphi \Rightarrow (\bigwedge k. k \in \{\text{ETP_f } \sigma i I .. j\} \Rightarrow \text{VIO } \sigma v k \psi) \Rightarrow$
 $\text{VIO } \sigma v i (\varphi \mathbf{U} I \psi)$
 $| \text{VUntilInf}: (\bigwedge k. k \in (\text{case right } I \text{ of } \infty \Rightarrow \{\text{ETP_f } \sigma i I ..\}$
 $| \text{enat } b \Rightarrow \{\text{ETP_f } \sigma i I .. \text{LTP_f } \sigma i b\}) \Rightarrow \text{VIO } \sigma v k \psi) \Rightarrow$
 $\text{VIO } \sigma v i (\varphi \mathbf{U} I \psi)$
 $| \text{SMatchP}: j \leq i \Rightarrow \text{mem } (\delta \sigma i j) I \Rightarrow \text{Regex_Proof_System.SAT } (\text{SAT } \sigma v) j i r \Rightarrow$
 $\text{SAT } \sigma v i (\text{MatchP } I r)$
 $| \text{VMatchPOut}: \tau \sigma i < \tau \sigma 0 + \text{left } I \Rightarrow \text{VIO } \sigma v i (\text{MatchP } I r)$
 $| \text{VMatchP}: k = (\text{case right } I \text{ of } \infty \Rightarrow 0 | \text{enat } b \Rightarrow \text{ETP_p } \sigma i b) \Rightarrow$
 $\tau \sigma i \geq \tau \sigma 0 + \text{left } I \Rightarrow (\bigwedge j. j \in \{k .. \text{LTP_p } \sigma i I\} \Rightarrow \text{Regex_Proof_System.VIO } (\text{VIO}$
 $\sigma v) j i r) \Rightarrow$
 $\text{VIO } \sigma v i (\text{MatchP } I r)$
 $| \text{SMatchF}: i \leq j \Rightarrow \text{mem } (\delta \sigma j i) I \Rightarrow \text{Regex_Proof_System.SAT } (\text{SAT } \sigma v) i j r \Rightarrow$
 $\text{SAT } \sigma v i (\text{MatchF } I r)$
 $| \text{VMatchF}: (\bigwedge j. j \in (\text{case right } I \text{ of } \infty \Rightarrow \{\text{ETP_f } \sigma i I ..\}$
 $| \text{enat } b \Rightarrow \{\text{ETP_f } \sigma i I .. \text{LTP_f } \sigma i b\}) \Rightarrow \text{Regex_Proof_System.VIO } (\text{VIO } \sigma v)$
 $i j r) \Rightarrow$
 $\text{VIO } \sigma v i (\text{MatchF } I r)$

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

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

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

by *auto*

moreover

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

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 $j \leq \text{ETP } \sigma (\text{Suc } (\tau \sigma i))$

unfolding *ETP_def*

proof (*intro LeastI2*[of $_ k \lambda i. j \leq i$])

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 ∈ {j <.. (min i (LTP σ (τ σ i - left I)))}
  with j(3) have k: j < k k ≤ i k ≤ Max {j. left I + τ σ j ≤ τ σ i}
    by (auto simp: LTP_def le_diff_conv2 add commute)
  with j(3) obtain l where left I + τ σ l ≤ τ σ i k ≤ l
    by (subst (asm) Max_ge_iff) (auto simp: le_diff_conv2 *
      intro!: finite_subset[of _ {0 .. ETP σ (τ σ i + 1)}])
  then have mem (τ σ i - τ σ k) I
    using k(1,2) j(3)
    by (cases right I) (auto simp: le_diff_conv le_diff_conv2 add commute dest: τ_mono
      elim: order_trans[rotated] order_trans)
  with Max_ge[of A k] k have ¬ ⟨σ, v, k⟩ ⊨ ψ
    unfolding j_def[symmetric] unfolding A_def
    by auto
}
ultimately show ?thesis using unsat
  by (auto dest!: spec[of _ j])
qed

```

lemma not_sat_UntilD:

```

assumes unsat: ¬ ⟨σ, v, i⟩ ⊨ φ U I ψ
  and witness: ∃ j ≥ i. mem (δ σ j i) I ∧ ⟨σ, v, j⟩ ⊨ ψ
shows ∃ j ≥ i. (case right I of ∞ ⇒ True | enat n ⇒ j < LTP σ (τ σ i + n))
  ∧ ¬ (⟨σ, v, j⟩ ⊨ φ) ∧ (∀ k ∈ {(max i (ETP σ (τ σ i + left I))) .. j}.
  ¬ ⟨σ, v, k⟩ ⊨ ψ)

```

proof -

```

from τ_mono have i0: τ σ 0 ≤ τ σ i by auto
from witness obtain jmax where jmax: jmax ≥ i ⟨σ, v, jmax⟩ ⊨ ψ
  mem (δ σ jmax i) I by blast
define A and j where A_def: A ≡ {j. j ≥ i ∧ j ≤ jmax
  ∧ mem (δ σ j i) I ∧ ⟨σ, v, j⟩ ⊨ ψ} and j_def: j ≡ Min A
have j: j ≥ i ⟨σ, v, j⟩ ⊨ ψ mem (δ σ j i) I
  using A_def j_def jmax Min_in[of A]
  unfolding j_def[symmetric] unfolding A_def
  by fastforce+
moreover have case_right I of ∞ ⇒ True | enat n ⇒ j ≤ LTP σ (τ σ i + n)
  using i0 j(1,3)
  by (auto simp: i_LTP_tau trans_le_add1 split: enat.splits)

```

moreover

```

{ fix k
  assume k_def: k ∈ {(max i (ETP σ (τ σ i + left I))) ..< j}
  then have ki: τ σ k ≥ τ σ i + left I using i_ETP_tau by auto
  with k_def have kj: k < j by auto
  then have τ σ k ≤ τ σ j by auto
  then have δ σ k i ≤ δ σ j i by auto
  with this j(3) have enat (δ σ k i) ≤ right I
    by (meson enat_ord_simps(1) order_subst2)
  with this ki j(3) have mem_k: mem (δ σ k i) I
    unfolding ETP_def by (auto simp: Least_le)

```

```

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

```

```

with this mem_k ki Min_le[of A k] k_def have k  $\notin$  A
  unfolding j_def[symmetric] unfolding A_def unfolding ETP_def
  using finite_nat_set_iff_bounded_le kj leD by blast
with this 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_ $\tau D$  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: right I = 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:  $\langle \sigma, v, k \rangle \models \varphi \wedge \text{mem } (\delta \sigma k i) I \wedge i \leq k$ 
    then have  $\tau \sigma k \geq \tau \sigma i + \text{left } I$ 
      using le_diff_conv2 by auto
    then have k_etp:  $k \geq ETP \sigma (\tau \sigma i + \text{left } I)$ 
      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:  $\neg \langle \sigma, v, k \rangle \models \varphi \wedge \text{mem } (\delta \sigma i k) I \wedge k \leq i$ 
    then have k_tau:  $\tau \sigma k \leq \tau \sigma i - \text{left } I$ 
      using diff_le_mono2 by fastforce
    then have k_ltp:  $k \leq \text{LTP } \sigma (\tau \sigma i - \text{left } I)$ 
      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  $\tau \sigma i < \tau \sigma j + \text{left } I$  using SHistoricallyOut by linarith
    then have  $\delta \sigma i j < \text{left } I$ 
      using SHistoricallyOut less_ $\tau$ D not_le by fastforce
    then have  $\neg \text{mem } (\delta \sigma i j) I$  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  $\notin$  {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  $\tau\ \sigma\ i < \tau\ \sigma\ j + left\ I$  using  $VSinceOut$  by linarith
    then have  $\delta\ \sigma\ i\ j < left\ I$  using  $VSinceOut\ j0$ 
    by (metis  $add.commute\ gr\_zeroI\ leI\ less\_tauD\ less\_diff\_conv2\ order\_less\_imp\_not\_less\ zero\_less\_diff$ )
    then have  $\neg\ mem\ (\delta\ \sigma\ i\ j)\ I$  by auto }
  then show ?case using  $VSinceOut$  by auto
next
  case (VSince I i j v  $\varphi$   $\psi$ )
  { fix k
    assume  $k\_def: \langle \sigma, v, k \rangle \models \psi \wedge mem\ (\delta\ \sigma\ i\ k)\ I \wedge k \leq i$ 
    then have  $\tau\ \sigma\ k \leq \tau\ \sigma\ i - left\ I$  using  $diff\_le\_mono2$  by fastforce
    then have  $k\_ltp: k \leq LTP\ \sigma\ (\tau\ \sigma\ i - left\ I)$ 
    using  $VSince\ i\_LTP\_tau\ add\_le\_imp\_le\_diff$ 
    by blast
    then have  $k < j$  using  $k\_def\ VSince(7)[of\ k]$ 
    by force
    then have  $j \in \{k <.. i\} \wedge \neg \langle \sigma, v, j \rangle \models \varphi$  using  $VSince$ 
    by auto }
  then show ?case using  $VSince$ 
  by force
next
  case (VSinceInf j I i v  $\psi$   $\varphi$ )
  { fix k
    assume  $k\_def: \langle \sigma, v, k \rangle \models \psi \wedge mem\ (\delta\ \sigma\ i\ k)\ I \wedge k \leq i$ 
    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  $VSinceInf\ i\_LTP\_tau\ add\_le\_imp\_le\_diff$ 
    by blast
    then have  $k \notin \{j .. LTP\_p\ \sigma\ i\ I\}$ 
    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 =  $\infty$ )
  (auto 0 0 simp:  $i\_ETP\_tau\ i\_LTP\_tau\ le\_diff\_conv2$ )
next

```

```

case (VUntil I j i v  $\varphi$   $\psi$ )
{ fix k
  assume k_def:  $\langle \sigma, v, k \rangle \models \psi \wedge \text{mem } (\delta \sigma k i) I \wedge i \leq k$ 
  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 VUntil i_ETP_tau by blast
  from this k_def VUntil have  $k \notin \{\text{ETP}_f \sigma i I .. j\}$  by auto
  then have  $j < k$  using k_etp k_def by auto
  then have  $j \in \{i .. < k\} \wedge \text{VIO } \sigma v j \varphi$  using VUntil k_def
    by auto }
then show ?case
  using VUntil by force
next
case (VUntilInf I i v  $\psi$   $\varphi$ )
{ fix k n
  assume r:  $\text{right } I = \text{enat } n$ 
  from this VUntilInf have tin0:  $\tau \sigma i + n \geq \tau \sigma 0$ 
    by (auto simp add: trans_le_add1)
  define j where  $j = \text{LTP } \sigma ((\tau \sigma i) + n)$ 
  from VUntilInf have j_i:  $i \leq j$ 
    by (auto simp add: i_LTP_tau trans_le_add1 j_def)
  assume k_def:  $\langle \sigma, v, k \rangle \models \psi \wedge \text{mem } (\delta \sigma k i) I \wedge i \leq k$ 
  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 VUntilInf i_ETP_tau by blast
  from this k_def VUntilInf have  $k \notin \{\text{ETP}_f \sigma i I .. j\}$ 
    by (auto simp: r j_def)
  then have  $j < k$  using VUntilInf k_def k_etp by auto
  from k_def r have  $\delta \sigma k i \leq n$  by auto
  then have  $\tau \sigma k \leq \tau \sigma i + n$  by auto
  then have  $k \leq j$ 
    using tin0 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  $\tau$ _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
next
case (SMatchP j i I v r)
then show ?case
  by (auto dest: Regex_Proof_System.soundness_SAT[rotated])
next
case (VMatchPOut i I v r)
{ 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 VMatchPOut by linarith

```

```

    then have  $\delta \sigma i j < \text{left } I$  using VMatchPOut 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 (VMatchP k I i v r)
then show ?case
  by (cases right I; force dest: Regex_Proof_System.soundness_VIO[rotated 2]
    simp: i_ETP_tau i_LTP_tau le_diff_conv le_diff_conv2 add commute)
next
case (SMatchF i j I v r)
then show ?case
  by (auto dest: Regex_Proof_System.soundness_SAT[rotated])
next
case (VMatchF I i v r)
from VMatchF show ?case
  by (cases right I; force dest: Regex_Proof_System.soundness_VIO[rotated 2]
    simp: i_ETP_tau i_LTP_tau le_diff_conv le_diff_conv2 add commute trans_le_add2)
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:  $(\langle \sigma, v, i \rangle \models \varphi \longrightarrow \text{SAT } \sigma v i \varphi) \wedge (\neg \langle \sigma, v, i \rangle \models \varphi \longrightarrow \text{VIO } \sigma v i \varphi)$ 
proof (induct φ arbitrary: v i)
  case (Prev I φ)
  show ?case using Prev
  by (auto intro: SAT_VIO.SPrev SAT_VIO.VPrev SAT_VIO.VPrevOutL SAT_VIO.VPrevOutR SAT_VIO.VPrevZ split: nat.splits)
next
case (Once I φ)
{ assume  $\langle \sigma, v, i \rangle \models \mathbf{P} I \varphi$ 
  with Once have  $\text{SAT } \sigma v i (\mathbf{P} I \varphi)$ 
  by (auto intro: SAT_VIO.SOnce) }
moreover
{ assume  $i\_l: \tau \sigma i < \tau \sigma 0 + \text{left } I$ 
  with Once have  $\text{VIO } \sigma v i (\mathbf{P} I \varphi)$ 
  by (auto intro: SAT_VIO.VOnceOut) }
moreover
{ assume unsat:  $\neg \langle \sigma, v, i \rangle \models \mathbf{P} I \varphi$ 
  and  $i\_ge: \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
  with Once have  $\text{VIO } \sigma v i (\mathbf{P} I \varphi)$ 
  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 φ)
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  $i\_ge: \tau \sigma i \geq \tau \sigma 0 + \text{left } I$ 
  with Historically have  $\text{SAT } \sigma v i (\mathbf{H} I \varphi)$ 
  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  $\text{VIO } \sigma v i (\mathbf{H} I \varphi)$ 

```

```

    by (auto intro: SAT_VIO.VHistorically) }
  moreover
  { assume  $i\_l: \tau \sigma i < \tau \sigma 0 + \text{left } I$ 
    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  $VIO \sigma v i (\mathbf{F} I \varphi)$ 
  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  $VIO \sigma v i (\mathbf{G} I \varphi)$ 
  by (auto intro: SAT_VIO.VAlways) }
moreover
{ assume  $sat: \langle \sigma, v, i \rangle \models \mathbf{G} I \varphi$ 
  with Always have  $SAT \sigma v i (\mathbf{G} I \varphi)$ 
  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  $i\_l: \tau \sigma i < \tau \sigma 0 + \text{left } I$ 
  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  $i\_ge: \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
  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  $i\_ge: \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
  from  $unsat$   $jw$   $not\_sat\_SinceD[of \sigma v i \varphi I \psi]$ 
  obtain  $j$  where  $j: j \leq i$ 
  case right  $I$  of  $\infty \Rightarrow True \mid enat \ n \Rightarrow ETP \sigma (\tau \sigma i - n) \leq j$ 
   $\neg \langle \sigma, v, j \rangle \models \varphi (\forall k \in \{j .. (\min i (LTP \sigma (\tau \sigma i - \text{left } I)))\})$ 
   $\neg \langle \sigma, v, k \rangle \models \psi$  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  $\infty \Rightarrow True \mid enat n$ 
 $\Rightarrow j < LTP \sigma (\tau \sigma i + n) \neg (\langle \sigma, v, j \rangle \models \varphi)$ 
 $(\forall k \in \{(max i (ETP \sigma (\tau \sigma i + left I)) .. j\}. \neg \langle \sigma, v, k \rangle \models \psi)$ 
  by auto
  with Until have VIO  $\sigma v i (\varphi \mathbf{U} I \psi)$ 
  using  $unsat witness$ 
  by (auto intro!: SAT_VIO.VUntil) }
moreover
{ assume  $unsat: \neg \langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi$ 
  and  $no\_witness: \forall j \geq i. \neg mem (\delta \sigma j i) I \vee \neg \langle \sigma, v, j \rangle \models \psi$ 
  with Until have VIO  $\sigma v i (\varphi \mathbf{U} I \psi)$ 
  by (auto intro!: SAT_VIO.VUntilInf simp: add_increasing2  $i0 i\_LTP\_tau i\_ETP\_tau$ 
 $split: enat.splits)$ 
}
ultimately show ?case by auto
next
case (MatchP I r)
show ?case
proof safe
  assume  $\langle \sigma, v, i \rangle \models \triangleleft I r$ 
  with MatchP show SAT  $\sigma v i (\triangleleft I r)$ 
  by (auto intro!: SMatchP Regex_Proof_System.completeness_SAT[of _ sat  $\sigma v$ ])
next
  assume  $unsat: \neg \langle \sigma, v, i \rangle \models \triangleleft I r$ 
  show VIO  $\sigma v i (\triangleleft I r)$ 
  proof (cases  $\tau \sigma i < \tau \sigma 0 + left I$ )
  case False
  with  $unsat MatchP$  show ?thesis
  by (auto intro!: VMatchP Regex_Proof_System.completeness_VIO[of _ sat  $\sigma v$ ]
 $simp: i\_ETP\_tau i\_LTP\_tau split: enat.splits)$ 
  qed (auto intro: VMatchPOut)
qed
next
case (MatchF I r)

show ?case
proof safe
  assume  $\langle \sigma, v, i \rangle \models \triangleright I r$ 
  with MatchF show SAT  $\sigma v i (\triangleright I r)$ 
  by (auto intro!: SMatchF Regex_Proof_System.completeness_SAT[of _ sat  $\sigma v$ ])
next
  assume  $unsat: \neg \langle \sigma, v, i \rangle \models \triangleright I r$ 
  with MatchF show VIO  $\sigma v i (\triangleright I r)$ 
  by (auto intro!: VMatchF Regex_Proof_System.completeness_VIO[of _ sat  $\sigma v$ ]
 $simp: i\_ETP\_tau i\_LTP\_tau trans_le_add2 add commute split: enat.splits)$ 

```

```

qed
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  $\sigma$  v i  $\varphi \vee$  VIO  $\sigma$  v i  $\varphi$ 
  using completeness[of  $\sigma$  v i  $\varphi$ ] by auto

end

unbundle no MFOTL_syntax

datatype (spatms: 'a) rsproof = SSkip nat nat | STest 'a | SPlusL 'a rsproof | SPlusR 'a rsproof
  | STimes 'a rsproof 'a rsproof | SStar_eps nat | SStar 'a rsproof list
datatype (vpatms: 'a) rvproof = VSkip nat nat | VTest 'a | VTest_neq nat nat | VPlus 'a rvproof 'a
  rvproof
  | VTimes (bool * 'a rvproof) list | VStar 'a rvproof list | VStar_gt nat nat

lemma size_hd_estimation[termination_simp]:  $xs \neq [] \implies \text{size}(\text{hd } xs) < \text{size\_list } \text{size } xs$ 
  by (cases xs) auto
lemma size_last_estimation[termination_simp]:  $xs \neq [] \implies \text{size}(\text{last } xs) < \text{size\_list } \text{size } xs$ 
  by (induct xs) auto
lemma size_rsproof_estimation[termination_simp]:  $x \in \text{spatms } p \implies y < f x \implies y < \text{size\_rsproof } f p$ 
  by (induct p) (auto simp: termination_simp)
lemma size_rsproof_estimation'[termination_simp]:  $x \in \text{spatms } p \implies y \leq f x \implies y \leq \text{size\_rsproof } f p$ 
  by (induct p) (auto simp: termination_simp)
lemma size_rvproof_estimation[termination_simp]:  $x \in \text{vpatms } p \implies y < f x \implies y < \text{size\_rvproof } f p$ 
  by (induct p) (auto simp: termination_simp sum_set_defs split: sum.splits)
lemma size_rvproof_estimation'[termination_simp]:  $x \in \text{vpatms } p \implies y \leq f x \implies y \leq \text{size\_rvproof } f p$ 
  by (induct p) (auto simp: termination_simp)

fun rs_at where
  rs_at test (SSkip k n) = (k, k + n)
| rs_at test (STest x) = (test x, test x)
| rs_at test (SPlusL p) = rs_at test p
| rs_at test (SPlusR p) = rs_at test p
| rs_at test (STimes p1 p2) = (fst (rs_at test p1), snd (rs_at test p2))
| rs_at test (SStar_eps n) = (n, n)
| rs_at test (SStar ps) = (if ps = [] then (0,0) else (fst (rs_at test (hd ps)), snd (rs_at test (last ps))))

lemma rs_at_cong[fundef_cong]:
   $p = p' \implies (\bigwedge x. x \in \text{spatms } p \implies t x = t' x) \implies \text{rs\_at } t p = \text{rs\_at } t' p'$ 
proof (induct p arbitrary: p')
  case (SStar ps)
  then show ?case using hd_in_set[of ps] last_in_set[of ps]
  by fastforce
qed auto

function(sequential) rv_at where
  rv_at test (VSkip n n') = (n, n')
| rv_at test (VTest p) = (test p, test p)
| rv_at test (VTest_neq n n') = (n, n')
| rv_at test (VPlus p1 p2) = rv_at test p1
| rv_at test (VTimes ps) = (if ps = [] then (0,0) else (fst (rv_at test (snd (hd ps))), snd (rv_at test (snd (last ps)))))
| rv_at test (VStar ps) = (Min (set (map (fst  $\circ$  (rv_at test)) ps)), Max (set (map (snd  $\circ$  (rv_at test))

```

```

ps)))
| rv_at test (VStar_gt n n') = (n, n')
  by pat_completeness auto
termination by (relation measure ( $\lambda(\_, vp). \text{size } vp$ ))
  (auto intro: less_Suc1 list.set_sel(1) size_list_estimation last_in_set simp: termination_simp)

lemma rv_at_cong[fundef_cong]:
   $p = p' \implies (\bigwedge x. x \in \text{vpatms } p \implies t x = t' x) \implies \text{rv\_at } t p = \text{rv\_at } t' p'$ 
proof (induct t p arbitrary: p' rule: rv_at.induct)
  case (5 t ps)
  then show ?case using hd_in_set[of ps] last_in_set[of ps]
  by (cases hd ps; cases last ps; fastforce)
next
  case (6 t ps)
  then show ?case
  by (force intro!: arg_cong[where f=Min] arg_cong[where f=Max] image_cong)
qed auto

```

7 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
| SMatchP ('n, 'd) sproof Regex_Proof_Object.rsproof
| SMatchF ('n, 'd) sproof Regex_Proof_Object.rsproof
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
| VMatchPOut nat
| VMatchP nat ('n, 'd) vproof Regex_Proof_Object.rvproof list
| VMatchF nat ('n, 'd) vproof Regex_Proof_Object.rvproof 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)
| s_at (SMatchP rsp) = (snd (rs_at s_at rsp))
| s_at (SMatchF rsp) = (fst (rs_at s_at rsp))
| 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
| v_at (VMatchPOut i) = i
| v_at (VMatchP i _) = i
| v_at (VMatchF i _) = i

```

definition $p_at :: ('n, 'd) \text{ proof} \Rightarrow \text{nat}$ **where** $p_at\ p = \text{case_sum } s_at\ v_at\ p$

8 Auxiliary Lemmas

lemma $\text{Cons_eq_upt_conv}: x \# xs = [m ..< n] \longleftrightarrow m < n \wedge x = m \wedge xs = [\text{Suc } m ..< n]$
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 $\text{set_Cons_eq}: \text{set_Cons } X\ XS = (\bigcup xs \in XS. (\lambda x. x \# xs) ' X)$
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 $\text{infinite } ((\lambda x. x \# xs) ' X)$

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

by blast

moreover have $((\lambda x. x \# xs) ' X) \subseteq (\bigcup xs \in XS. (\lambda x. x \# xs) ' X)$

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

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

by (simp add: infinite_super)

next
assume *infinite XS and $X \neq \{\}$*
then show *infinite $(\bigcup xs \in XS. (\lambda x. x \# xs) ' X)$*
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 = ($\lambda ns. \text{if } x = y \text{ then } 0 \# ns \text{ else } ns$) (map Suc (positions ys x))*

lemma *eq_positions_iff: length xs = length ys*
 \Longrightarrow *positions xs x = positions ys y \longleftrightarrow ($\forall n < \text{length } xs. xs ! n = x \longleftrightarrow ys ! n = y$)*
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 <auto simp: positions_eq_nil_iff[symmetric] image_iff split: if_splits>)

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*: $x \in \text{set } xs \implies \text{Min} (\text{set} (\text{positions } xs \ x)) = \text{the} (\text{fst_pos } xs \ x)$
by (*auto simp: Min_sorted_list[OF sorted_positions]*
positions_eq_nil_iff hd_positions_eq_fst_pos)

lemma *subset_positions_map_fst*: $\text{set} (\text{positions } tXs \ tX) \subseteq \text{set} (\text{positions} (\text{map } \text{fst } tXs) (\text{fst } tX))$
by (*induct tXs arbitrary: tX*)
(auto simp: subset_eq)

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

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

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

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

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

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

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

lemma *max_aux*: $\text{finite } X \implies \text{Suc } j \in X \implies \text{Max} (\text{insert} (\text{Suc } j) (X - \{j\})) = \text{Max} (\text{insert } j \ X)$
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*

lemma *ball_if_distrib*: $(\forall x \in B. \text{if } p \ \text{then } f \ x \ \text{else } g \ x) \longleftrightarrow (\text{if } p \ \text{then } (\forall x \in B. f \ x) \ \text{else } (\forall x \in B. g \ x))$
by *simp*

context *fixes* *test* :: 'a \Rightarrow 'b \Rightarrow bool **and** *testi* :: 'b \Rightarrow nat **begin**
fun *rs_check* **where**
rs_check (*Regex.Skip* *n*) (*SSkip* *x y*) = ((*snd* (*rs_at* *testi* (*SSkip* *x y*))) = *x* + *n*)
| *rs_check* (*Regex.Test* *x*) (*STest* *y*) = *test* *x y*
| *rs_check* (*Regex.Plus* *r r'*) (*SPlusL* *z*) = *rs_check* *r z*
| *rs_check* (*Regex.Plus* *r r'*) (*SPlusR* *z*) = *rs_check* *r' z*
| *rs_check* (*Regex.Times* *r r'*) (*STimes* *p1 p2*) =
(*snd* (*rs_at* *testi* *p1*) = *fst* (*rs_at* *testi* *p2*)) \wedge *rs_check* *r p1* \wedge *rs_check* *r' p2*)
| *rs_check* (*Regex.Star* *r*) (*SStar_eps* *n*) = *True*

```

| rs_check (Regex.Star r) (SStar ps) = (ps ≠ [] ∧
  (∀ k ∈ {1 ..< length ps}. fst (rs_at testi (ps ! k)) = snd (rs_at testi (ps ! (k-1)))) ∧
  (∀ k ∈ {0 ..< length ps}. fst (rs_at testi (ps ! k)) < snd (rs_at testi (ps ! k)) ∧ rs_check r (ps ! k)))
| rs_check _ _ = False
end

```

lemma *rs_check_cong*[*fundef_cong*]:

```

p = p' ⇒ (∧ x sp. x ∈ regex.atms r ⇒ sp ∈ spatms p ⇒ t x sp = t' x sp)
⇒ (∧ x. x ∈ spatms p ⇒ ti x = ti' x) ⇒ rs_check t ti r p = rs_check t' ti' r p'
proof (hypsubst_thin, induct r p' rule: rs_check.induct)

```

```

case (7 r ps)
have rs_check t ti r (ps ! k) = rs_check t' ti' r (ps ! k) if k ∈ {0 ..< length ps} for k
  using that by (intro 7) (auto simp: Bex_def in_set_conv_nth)
moreover have rs_at ti (ps ! k) = rs_at ti' (ps ! k) if k ∈ {0 ..< length ps} for k
  using that by (intro rs_at_cong 7) (auto simp: Bex_def in_set_conv_nth)
ultimately show ?case
  by auto
qed(auto cong: rs_at_cong)

```

context fixes *test* :: 'a ⇒ 'b ⇒ bool and *testi* :: 'b ⇒ nat **begin**

fun *rv_check* **where**

```

rv_check (Regex.Skip n) (VSkip i j) = (i ≤ j ∧ j ≠ i + n)
| rv_check (Regex.Test x) (VTest p) = test x p
| rv_check (Regex.Test x) (VTest_neq i j) = (i < j)
| rv_check (Regex.Plus r r') (VPlus p1 p2) =
  (rv_check r p1 ∧ rv_check r' p2 ∧ rv_at testi p1 = rv_at testi p2)
| rv_check (Regex.Times r r') (VTimes ps) = (ps ≠ [] ∧
  (∃ i j. i = fst (rv_at testi (snd (hd ps))) ∧ j = snd (rv_at testi (snd (last ps)))) ∧
  i + length ps - 1 = j ∧ (∀ k ∈ {0 ..< length ps}. let (b, p) = ps ! k in
  if b then rv_check r p ∧ rv_at testi p = (i, i + k)
  else rv_check r' p ∧ rv_at testi p = (i + k, j))))
| rv_check (Regex.Star r) (VStar ps) =
  (∃ S T i j. S = set (map (fst ∘ rv_at testi) ps) ∧ T = set (map (snd ∘ rv_at testi) ps)
  ∧ i = Min S ∧ j = Max T ∧ i ≤ j ∧ S ∩ T = {} ∧ S ∪ T = {i .. j}
  ∧ map (rv_at testi) ps = sorted_list_of_set (rm (S × T))
  ∧ (∀ k ∈ {0 ..< length ps}. rv_check r (ps ! k)))
| rv_check (Regex.Star r) (VStar_gt n n') = (n > n')
| rv_check _ _ = False

```

lemma *rv_check_code_Times*:

```

rv_check (Regex.Times r r') (VTimes ps) = (ps ≠ [] ∧
  (let i = fst (rv_at testi (snd (hd ps))); j = snd (rv_at testi (snd (last ps))) in
  i + length ps - 1 = j ∧ (∀ k ∈ {0 ..< length ps}. let (b, p) = ps ! k in
  if b then rv_check r p ∧ rv_at testi p = (i, i + k)
  else rv_check r' p ∧ rv_at testi p = (i + k, j))))
by (simp add: Let_def)

```

lemma *rv_check_code_Star*:

```

rv_check (Regex.Star r) (VStar ps) =
  (let S = set (map (fst ∘ rv_at testi) ps); T = set (map (snd ∘ rv_at testi) ps);
  i = Min S; j = Max T in i ≤ j ∧ S ∩ T = {} ∧ S ∪ T = {i .. j}
  ∧ map (rv_at testi) ps = sorted_list_of_set (rm (S × T))
  ∧ (∀ k ∈ {0 ..< length ps}. rv_check r (ps ! k)))
by (simp add: Let_def)

```

lemmas *rv_check_code*[*code*] = *rv_check.simps*(1-4) *rv_check_code_Times* *rv_check_code_Star* *rv_check.simps*(7-)

end

lemma *rv_check_cong*[*fundef_cong*]:

$p = p' \implies (\bigwedge x \text{ vp. } x \in \text{regex.atms } r \wedge \text{vp} \in \text{vpatms } p \implies t \ x \ \text{vp} = t' \ x \ \text{vp})$
 $\implies (\bigwedge x. x \in \text{vpatms } p \implies ti \ x = ti' \ x) \implies \text{rv_check } t \ ti \ r \ p = \text{rv_check } t' \ ti' \ r \ p'$

proof (*hypsubst_thin*, *induct* *r p'* rule: *rv_check.induct*)

case (*5 r r' ps*)
have $\text{rv_check } t \ ti \ r \ (\text{snd } (ps \ ! \ k)) = \text{rv_check } t' \ ti' \ r \ (\text{snd } (ps \ ! \ k))$ **if** $\text{fst } (ps \ ! \ k) \ k \in \{0 \ ..< \ \text{length } ps\}$ **for** *k*
using *that surjective_pairing*[*of ps ! k*]
by (*intro* *5(1)*[*OF that(2) refl prod.collapse that(1)*] *5(3-)*)
(auto simp: Bex_def in_set_conv_nth simp del: prod.collapse)
moreover have $\text{rv_check } t \ ti \ r' \ (\text{snd } (ps \ ! \ k)) = \text{rv_check } t' \ ti' \ r' \ (\text{snd } (ps \ ! \ k))$ **if** $\neg \text{fst } (ps \ ! \ k) \ k \in \{0 \ ..< \ \text{length } ps\}$ **for** *k*
using *that surjective_pairing*[*of ps ! k*]
by (*intro* *5(2)*[*OF that(2) refl prod.collapse that(1)*] *5(3-)*)
(auto simp: Bex_def in_set_conv_nth simp del: prod.collapse)
moreover have $\text{rv_at } ti \ (\text{snd } (ps \ ! \ k)) = \text{rv_at } ti' \ (\text{snd } (ps \ ! \ k))$ **if** $k \in \{0 \ ..< \ \text{length } ps\}$ **for** *k*
using *that surjective_pairing*[*of ps ! k*] **by** (*intro rv_at_cong 5 refl*)
(auto simp: Bex_def in_set_conv_nth simp del: prod.collapse)
ultimately show *?case*
by (*auto simp: hd_conv_nth last_conv_nth split_beta*)
next
case (*6 r ps*)
have $\text{rv_check } t \ ti \ r \ (\text{snd } (ps \ ! \ k)) = \text{rv_check } t' \ ti' \ r \ (\text{snd } (ps \ ! \ k))$ **if** $k \in \{0 \ ..< \ \text{length } ps\}$ **for** *k*
using *that by* (*intro 6*) *(auto simp: Bex_def in_set_conv_nth)*
moreover have $\text{map } (\text{rv_at } ti) \ ps = \text{map } (\text{rv_at } ti') \ ps$
by (*intro rv_at_cong 6 list.map_cong*) *(auto simp: Bex_def in_set_conv_nth)*
ultimately show *?case* **unfolding** *rv_check.simps list.map_comp[symmetric]*
by *metis*
qed (*auto cong: rv_at_cong*)

lemma *Cons_eq_upt_conv*: $x \# xs = [m \ ..< \ n] \longleftrightarrow m < n \wedge x = m \wedge xs = [Suc \ m \ ..< \ n]$

by (*induct* *n arbitrary: xs*) (*force simp: Cons_eq_append_conv*)**+**

lemma *map_setE*[*elim_format*]: $\text{map } f \ xs = ys \implies y \in \text{set } ys \implies \exists x \in \text{set } xs. f \ x = y$

by (*induct xs arbitrary: ys*) *auto*

lemma *rs_check_sound*:

$\forall x \in \text{Regex.atms } r. \forall p' \in \text{spatms } p. \text{test } x \ p' \longrightarrow \text{sat } (\text{testi } p') \ x \implies$

$\text{rs_check } \text{test } \text{testi } r \ p \implies \text{Regex_Proof_System.SAT } \text{sat } (\text{fst } (\text{rs_at } \text{testi } p)) \ (\text{snd } (\text{rs_at } \text{testi } p)) \ r$

proof (*induction* *p arbitrary: r*)

case (*SSkip x y*)

then show *?case*

by (*cases r*) (*auto intro: SAT.SSkip*)

next

case (*SSTest x*)

then show *?case*

by (*cases r*) (*auto intro: SAT.SSTest*)

next

case (*SPlusL p*)

then show *?case*

by (*cases r*) (*auto intro: SAT.SPlusL*)

next

case (*SPlusR p*)

then show *?case*

by (*cases r*) (*auto intro: SAT.SPlusR*)

next

case (*SSTimes p1 p2*)

then show *?case*

```

    by (cases r) (auto intro!: SAT.STimes)
next
case (SStar_eps x)
then show ?case
  by (cases r) (auto intro: SAT.SStar_eps)
next
case (SStar ps)
then show ?case using SStar
proof (cases r)
  case (Star r')
  then have ps_ne: ps ≠ [] and
    ps_chain: ∀ k ∈ {1 ..< length ps}. fst (rs_at testi (ps ! k)) = snd (rs_at testi (ps ! (k-1)))
    using SStar by auto

  define ts where ts = map (fst o rs_at testi) ps @ [snd (rs_at testi (last ps))]
  then have ts_len: 2 ≤ length ts and ts_ne[simp]: ts ≠ []
    using ps_ne by (cases ps; auto)+

  from SStar(2) Star
  have r'_atms: ∀ y ∈ Regex.atms r'. ∀ p' ∈ spatms (SStar ps). test y p' → sat (testi p') y
    by auto

  { fix k
    assume k_def: k ∈ {0 ..< length ps}
    then have Regex_Proof_System.SAT sat (fst (rs_at testi (ps ! k))) (snd (rs_at testi (ps ! k))) r' ∧
      fst (rs_at testi (ps ! k)) < snd (rs_at testi (ps ! k))
      using SStar(1)[of ps ! k r'] r'_atms SStar(2-3) Star by force
    }

  then have sat_props_ts: ∀ k ∈ {0 ..< length ts - 1}. ts ! k < ts ! Suc k ∧
    Regex_Proof_System.SAT sat (ts ! k) (ts ! Suc k) r'
    hd ts = fst (rs_at testi (hd ps)) last ts = snd (rs_at testi (last ps))
    using ps_ne ps_chain
    by (auto simp: ts_def nth_append last_conv_nth neq_Nil_conv less_Suc_eq)
  then have s_ts: sorted_wrt (<) ts
    by (subst sorted_wrt_iff_nth_Suc_transp) auto
  have form: ∃ zs. ts = hd ts # zs @ [last ts]
    using ts_len by (cases ts) (auto intro!: exI[of _ butlast _] append_butlast_last_id[symmetric])
  then have hd ts < last ts
    using s_ts form ts_len by (auto simp: sorted_wrt_iff_nth_less hd_conv_nth last_conv_nth)
  then show ?thesis using sat_props_ts form ts_def
    SAT.SStar[of hd ts last ts ts sat r'] Star by auto
qed auto
qed

lemma rs_check_complete:
  (∀ x ∈ Regex.atms r. ∀ i. sat i x → (∃ p'. testi p' = i ∧ test x p')) ⇒
  Regex_Proof_System.SAT sat i j r ⇒ ∃ p. rs_check test testi r p ∧ rs_at testi p = (i, j)
proof (induction r arbitrary: i j)
  case (Skip x)
  then have j_eq_i_plus_x: j = i + x
    using SAT.simps[of sat i j Regex.Skip x]
    by simp
  then have rs_check_test_testi (Regex.Skip x) (SSkip i x)
    using rs_check.simps(1)[of test testi x i x]
    by simp
  then show ?case
    using j_eq_i_plus_x rs_at.simps(1)[of testi i x]

```

```

    by blast
next
case (Test x)
then have props:  $i = j \wedge \text{sat } j \ x$ 
  using SAT.simps[of sat i j Regex.Test x]
  by auto
then obtain  $p'$  where  $p'_\text{def}: \text{test } x \ p' \wedge \text{testi } p' = j$ 
  using Test(1)
  by auto
then show ?case
  using rs_check.simps(2)[of test testi x p'] props
  rs_at.simps(2)[of testi p']
  by blast
next
case (Plus r1 r2)
from Plus(4) have Regex_Proof_System.SAT sat i j r1  $\vee$  Regex_Proof_System.SAT sat i j r2
  using SAT.simps[of sat i j Regex.Plus r1 r2]
  by simp
moreover
{
  assume sl: Regex_Proof_System.SAT sat i j r1
  from Plus(3) have r1_atms:  $\forall x \in \text{regex.atms } r1. \forall i. \text{sat } i \ x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \ p')$ 
    by auto
  from Plus(1)[OF r1_atms sl]
  obtain  $p$  where  $p_\text{check}: \text{rs\_check test testi } r1 \ p$ 
    and  $p_\text{at}: \text{rs\_at testi } p = (i, j)$ 
    by auto
  then have  $\exists p. \text{rs\_check test testi } (\text{Regex.Plus } r1 \ r2) \ p \wedge \text{rs\_at testi } p = (i, j)$ 
    using rs_check.simps(3)[of test testi r1 r2 p]
    by fastforce
}
moreover
{
  assume sr: Regex_Proof_System.SAT sat i j r2
  from Plus(3) have r2_atms:  $\forall x \in \text{regex.atms } r2. \forall i. \text{sat } i \ x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \ p')$ 
    by auto
  from Plus(2)[OF r2_atms sr]
  obtain  $p$  where  $p_\text{check}: \text{rs\_check test testi } r2 \ p$ 
    and  $p_\text{at}: \text{rs\_at testi } p = (i, j)$ 
    by auto
  then have  $\exists p. \text{rs\_check test testi } (\text{Regex.Plus } r1 \ r2) \ p \wedge \text{rs\_at testi } p = (i, j)$ 
    using rs_check.simps(4)[of test testi r1 r2 p]
    by fastforce
}
ultimately show ?case
  by auto
next
case (Times r1 r2)
then obtain  $k$  where  $ks\_r1: \text{Regex\_Proof\_System.SAT sat } i \ k \ r1$ 
  and  $ks\_r2: \text{Regex\_Proof\_System.SAT sat } k \ j \ r2$ 
  using SAT.simps[of sat i j Regex.Times r1 r2]
  by auto
from Times(3) have r1_atms:  $\forall x \in \text{regex.atms } r1. \forall i. \text{sat } i \ x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \ p')$  and
  r2_atms:  $\forall x \in \text{regex.atms } r2. \forall i. \text{sat } i \ x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \ p')$ 
  by auto
from Times(1)[OF r1_atms ks_r1] obtain  $p$  where

```

```

    p_check: rs_check test testi r1 p and p_at: rs_at testi p = (i, k)
  by auto
from Times(2)[OF r2_atms ks_r2] obtain p' where
  p'_check: rs_check test testi r2 p' and p'_at: rs_at testi p' = (k, j)
  by auto
then show ?case
  using rs_check.simps(5)[of test testi r1 r2 p p'] p_check p_at
  by fastforce
next
case (Star r')
then show ?case
proof (cases i = j)
  case True
  then show ?thesis
    using rs_check.simps(6)[of test testi r'] rs_at.simps(6)
    by blast
next
case False
then have i_less_j: i < j
  using Star SAT.simps[of sat i j Regex.Star r']
  by simp
from Star i_less_j SAT.simps[of sat i j Regex.Star r']
obtain xs and zs where xs_def: xs = i # zs @ [j] and
  k_less:  $\forall k \in \{0 \dots \text{length } xs - 1\}. xs ! k < xs ! \text{Suc } k$  and
  k_sat:  $\forall k \in \{0 \dots \text{length } xs - 1\}. \text{Regex\_Proof\_System.SAT sat } (xs ! k) (xs ! \text{Suc } k) r'$ 
  by auto
from Star(2) have r'_atms:  $\forall x \in \text{regex.atms } r'. \forall i. \text{sat } i x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x p')$ 
  by auto

{fix k
  assume k_in:  $k \in \{0 \dots \text{length } xs - 1\}$ 
  then have ksat:  $\text{Regex\_Proof\_System.SAT sat } (xs ! k) (xs ! \text{Suc } k) r'$ 
    using k_sat
    by auto
  from Star(1)[OF r'_atms ksat]
  have  $\exists p. \text{rs\_check test testi } r' p \wedge \text{rs\_at testi } p = (xs ! k, xs ! \text{Suc } k)$ 
    by simp
} thm rs_check.simps(7)
then have k_ex_p:  $\forall k \in \{0 \dots \text{length } xs - 1\}. \exists p. \text{rs\_check test testi } r' p \wedge \text{rs\_at testi } p = (xs !$ 
 $k, xs ! \text{Suc } k)$ 
  by auto
then obtain f where f_def:  $\forall k \in \{0 \dots \text{length } xs - 1\}. \text{rs\_at testi } (f k) = (xs ! k, xs ! \text{Suc } k) \wedge$ 
 $\text{rs\_check test testi } r' (f k)$ 
  using bchoice[OF k_ex_p]
  by atomize_elim auto
define ps where ps = map f [0 ..< length xs - 1]
then have ps_check_and_less:  $\forall k \in \{0 \dots \text{length } ps\}. \text{rs\_check test testi } r' (ps ! k) \wedge \text{fst } (\text{rs\_at}$ 
 $\text{testi } (ps ! k)) < \text{snd } (\text{rs\_at testi } (ps ! k))$ 
  using f_def k_less by auto
moreover
from ps_def f_def
have k_eq_prev:  $\forall k \in \{1 \dots \text{length } ps\}. \text{fst } (\text{rs\_at testi } (ps ! k)) = \text{snd } (\text{rs\_at testi } (ps ! (k - 1)))$ 
  by auto
moreover
from xs_def ps_def have ps_nnil: ps  $\neq []$ 
  by auto
moreover
from f_def have hd_eq:  $\text{fst } (\text{rs\_at testi } (\text{hd } ps)) = i$ 

```

```

    using ps_def ps_nnil upt_rec xs_def by auto
  moreover
  from xs_def ps_def f_def have last_eq: snd (rs_at testi (last ps)) = j
    using ps_nnil by auto
  ultimately show ?thesis
    by (auto intro!: exI[of _ SStar ps])
qed
qed

lemma rv_check_sound:
   $\forall x \in \text{Regex.atms } r. \forall p' \in \text{vpatms } p. \text{test } x \ p' \longrightarrow \text{vio } (\text{testi } p') \ x \Longrightarrow$ 
   $\text{rv\_check test testi } r \ p \Longrightarrow \text{Regex\_Proof\_System.VIO vio } (\text{fst } (\text{rv\_at testi } p)) \ (\text{snd } (\text{rv\_at testi } p)) \ r$ 
proof (induction p arbitrary: r)
  case (VSkip x y)
  then show ?case
    by (cases r) (auto intro: VIO.VSkip)
next
  case (VTest x)
  then show ?case
    by (cases r) (auto intro: VIO.VTest)
next
  case (VTest_neq x y)
  then show ?case
    by (cases r) (auto intro: VIO.VTest_neq)
next
  case (VPlus p1 p2)
  then show ?case
    by (cases r) (auto intro: VIO.VPlus)
next
  case (VTimes ps)
  then show ?case
  proof (cases r)
  case (Times r1 r2)
  then obtain i and j where ps_ne: ps  $\neq$  [] and i_def:  $i = \text{fst } (\text{rv\_at testi } (\text{snd } (\text{hd } ps)))$  and
    j_def:  $j = \text{snd } (\text{rv\_at testi } (\text{snd } (\text{last } ps)))$  and ij_props:  $i + \text{length } ps - 1 = j$ 
    using VTimes(3) by simp
  then have k_props:  $\forall k \in \{0 \ ..< \text{length } ps\}. \text{if } \text{fst } (ps \ ! \ k)$ 
    then  $\text{rv\_check test testi } r1 \ (\text{snd } (ps \ ! \ k)) \wedge \text{rv\_at testi } (\text{snd } (ps \ ! \ k)) = (i, i + k)$ 
    else  $\text{rv\_check test testi } r2 \ (\text{snd } (ps \ ! \ k)) \wedge \text{rv\_at testi } (\text{snd } (ps \ ! \ k)) = (i + k, j)$ 
    using VTimes(3) Times by auto
  from VTimes(2) Times have r1_atms:  $\forall y \in \text{Regex.atms } r1. \forall p' \in \text{vpatms } (VTimes \ ps). \text{test } y \ p'$ 
 $\longrightarrow \text{vio } (\text{testi } p') \ y$ 
    by auto
  from VTimes(2) Times have r2_atms:  $\forall y \in \text{Regex.atms } r2. \forall p' \in \text{vpatms } (VTimes \ ps). \text{test } y \ p'$ 
 $\longrightarrow \text{vio } (\text{testi } p') \ y$ 
    by auto

  { fix k
    assume k_def:  $k \in \{0 \ ..< \text{length } ps\}$ 
    then have if fst (ps ! k) then  $\text{Regex\_Proof\_System.VIO vio } i \ (i + k) \ r1$  else  $\text{Regex\_Proof\_System.VIO}$ 
 $\text{vio } (i + k) \ j \ r2$ 
      using VTimes(1)[of ps ! k snd (ps ! k) r1] VTimes(1)[of ps ! k snd (ps ! k) r2] Times k_props
      r1_atms r2_atms
      by (fastforce simp: prod_set_defs)
    } note k_vio = this

  define ts where  $ts = \text{map } (\lambda k. \text{if } \text{fst } (ps \ ! \ k) \text{ then}$ 
 $\text{snd } (\text{rv\_at testi } (\text{snd } (ps \ ! \ k))) \text{ else } \text{fst } (\text{rv\_at testi } (\text{snd } (ps \ ! \ k)))) \ [0 \ ..< \text{length } ps]$ 

```

```

then have ts_ps: length ts = length ps and ts_ne[simp]: ts ≠ []
  using ps_ne by (cases ps; auto)+

{ fix k
  assume k_def: k ∈ set ts
  then obtain k' where k'_def: k = i + k' k' ∈ {0 ..< length ps}
    using k_def k_props unfolding ts_def by auto
  then have iffst (ps! (k - i)) then Regex_Proof_System.VIO via i k r1 else Regex_Proof_System.VIO
via k j r2
    using k'_def k_vio[of k'] by auto
} note k_vio_ts = this

{ fix k
  assume k_def: k ∈ set ts
  with k_props ij_props have k ∈ {i .. j}
    unfolding ts_def by auto
}
}
moreover
{ fix k
  assume k_def: k ∈ {i .. j}
  then obtain n where n < length ps i + n = k
    using ij_props ps_ne by (auto simp: nat_le_iff_add neq_Nil_conv)
  then have k = ts! n
    using k_def k_props unfolding ts_def by auto
  then have k ∈ set ts using  $\langle n < \text{length } ps \rangle$  ts_ps
    by (auto simp: in_set_conv_nth)
}
ultimately have set ts = {i .. j} by blast
then show ?thesis using k_vio_ts i_def j_def ps_ne
  VIO.VTimes[of i j via r1 r2] Times unfolding rv_at.simps by (smt (verit, best) split_pairs)
qed auto
next
case (VStar ps)
then show ?case
proof (cases r)
case (Star r')
define S and T where S = set (map (fst ∘ rv_at testi) ps)
  and T = set (map (snd ∘ rv_at testi) ps)
define i and j where i = Min S and j = Max T
then have ST_props: S ∩ T = {} S ∪ T = {i .. j} and i_le_j: i ≤ j
  using VStar Star S_def T_def by auto
then have ST_not_empty: S ≠ {} T ≠ {} and ps_ne: ps ≠ []
  unfolding S_def T_def using i_le_j by auto
then have prod_not_empty: S × T ≠ {}
  by auto
from ST_props have ST_finite: finite S finite T
  unfolding S_def T_def by auto
then have i_in: i ∈ S and j_in: j ∈ T
  using Min_in[of S] Max_in[of T] ST_props ST_not_empty unfolding i_def j_def by auto
then have i_less_j: i < j
  by (metis IntI ST_props(1) equals0D i_le_j order_neq_le_trans)
then have rm_not_empty: rm (S × T) ≠ {}
  using S_def T_def i_le_j i_def j_def prod_not_empty ST_props by force
have rm_finite: finite (rm (S × T))
  by (auto simp add: Collect_case_prod_Sigma ST_finite)
then have set_eq: set (map (rv_at testi) ps) = rm (S × T)
  using S_def T_def VStar(3) Star by auto

```

```

from VStar(2) Star have r'_atms:  $\forall y \in \text{regex.atms } r'. \forall p' \in \text{vpatms } (VStar \text{ ps}). \text{test } y \text{ } p' \longrightarrow \text{vio}$ 
(testi p') y
by auto

{ fix k
assume k_in:  $k \in \{0 \dots \text{length ps}\}$ 
then have Regex_Proof_System.VIO vio (fst (rv_at testi (ps ! k))) (snd (rv_at testi (ps ! k))) r'
using VStar(1)[of ps ! k r'] Star VStar(2-3) r'_atms by force
} note k_vio = this

{ fix k
assume k_in:  $k \in \{0 \dots \text{length ps}\}$ 
then have rv_at testi (ps ! k)  $\in \text{set } (\text{map } (rv\_at \text{ testi}) \text{ ps})$ 
by simp
then have (fst (rv_at testi (ps ! k)), snd (rv_at testi (ps ! k)))  $\in \text{rm } (S \times T)$ 
using set_eq by auto
}
then have  $\forall x \in \text{set } (\text{map } (rv\_at \text{ testi}) \text{ ps}). \text{Regex\_Proof\_System.VIO } \text{vio } (\text{fst } x) (\text{snd } x) r'$ 
using k_vio by (force simp: in_set_conv_nth)
then have st_vio:  $\forall (s, t) \in \text{rm}(S \times T). \text{Regex\_Proof\_System.VIO } \text{vio } s \text{ } t \text{ } r'$ 
using set_eq[symmetric] by auto
show ?thesis
using VStar Star VIO.VStar[OF i_less_j i_in j_in _ _ st_vio] ST_props
S_def T_def by auto
qed auto
next
case (VStar_gt n n')
then show ?case
by (auto elim!: rv_check.elims intro: VIO.VStar_gt)
qed

lemma rv_check_complete:
( $\forall x \in \text{Regex.atms } r. \forall i. \text{vio } i \text{ } x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \text{ } p')$ )  $\implies$ 
Regex_Proof_System.VIO vio i j r  $\implies i \leq j \implies \exists p. \text{rv\_check } \text{test } \text{testi } r \text{ } p \wedge \text{rv\_at } \text{testi } p = (i, j)$ 
proof(induction r arbitrary: i j)
case (Skip x)
then have j_noteq:  $j \neq i + x$ 
using VIO.simps[of vio i j Regex.Skip x]
by simp
then have rv_check test testi (Regex.Skip x)  $(V\text{Skip } i \text{ } j) \wedge \text{rv\_at } \text{testi } (V\text{Skip } i \text{ } j) = (i, j)$ 
using Skip(3)
by auto
then show ?case
by (auto intro: exI[of _ VSkip i j])
next
case (Test x)
then show ?case
proof (cases i < j)
case True
then show ?thesis
using rv_check.simps(3)[of test testi x i j] Test
rv_at.simps(3)[of testi i j]
by blast
next
case False
then have i_eq_j:  $i = j \wedge \text{vio } j \text{ } x$ 
using Test VIO.simps[of vio i j Regex.Test x]
by auto

```

```

then obtain  $p$  where  $p\_def: test\ x\ p\ testi\ p = j$ 
  using  $Test$ 
  by  $auto$ 
then show  $?thesis$ 
  using  $rv\_check.simps(2)[of\ test\ testi\ x\ p]\ Test$ 
   $rv\_at.simps(2)[of\ testi\ p]\ i\_eq\_j$ 
  by  $blast$ 
qed
next
case  $(Plus\ r1\ r2)$ 
then have  $vio\_r1: Regex\_Proof\_System.VIO\ vio\ i\ j\ r1$  and  $vio\_r2: Regex\_Proof\_System.VIO\ vio\ i$ 
 $j\ r2$ 
  using  $VIO.simps[of\ vio\ i\ j\ Regex.Plus\ r1\ r2]$ 
  by  $simp+$ 
from  $Plus(3)$  have  $r1\_atms: \forall x \in regex.atms\ r1. \forall i. vio\ i\ x \longrightarrow (\exists p'. testi\ p' = i \wedge test\ x\ p')$  and
 $r2\_atms: \forall x \in regex.atms\ r2. \forall i. vio\ i\ x \longrightarrow (\exists p'. testi\ p' = i \wedge test\ x\ p')$ 
  by  $auto$ 
from  $Plus(1)[OF\ r1\_atms\ vio\_r1\ Plus(5)]$  obtain  $p1$  where
 $p1\_def: rv\_check\ test\ testi\ r1\ p1 \wedge rv\_at\ testi\ p1 = (i, j)$ 
  by  $auto$ 
from  $Plus(2)[OF\ r2\_atms\ vio\_r2\ Plus(5)]$  obtain  $p2$  where
 $p2\_def: rv\_check\ test\ testi\ r2\ p2 \wedge rv\_at\ testi\ p2 = (i, j)$ 
  by  $auto$ 
then show  $?case$ 
  using  $rv\_check.simps(4)[of\ test\ testi\ r1\ r2\ p1\ p2]\ p1\_def$ 
 $rv\_at.simps(4)[of\ testi\ p1\ p2]$ 
  by  $fastforce$ 
next
case  $(Times\ r1\ r2)$ 
then have  $k\_vio: \forall k \in \{i .. j\}. Regex\_Proof\_System.VIO\ vio\ i\ k\ r1 \vee Regex\_Proof\_System.VIO\ vio$ 
 $k\ j\ r2$ 
  using  $VIO.simps[of\ vio\ i\ j\ Regex.Times\ r1\ r2]$ 
  by  $simp$ 
from  $Times(3)$  have  $r1\_atms: \forall x \in regex.atms\ r1. \forall i. vio\ i\ x \longrightarrow (\exists p'. testi\ p' = i \wedge test\ x\ p')$  and
 $r2\_atms: \forall x \in regex.atms\ r2. \forall i. vio\ i\ x \longrightarrow (\exists p'. testi\ p' = i \wedge test\ x\ p')$ 
  by  $auto$ 

{fix  $k$ 
  assume  $k\_in: k \in \{i .. j\}$ 
  then have  $(\exists p. (rv\_check\ test\ testi\ r1\ p \wedge rv\_at\ testi\ p = (i, k)) \vee$ 
 $(rv\_check\ test\ testi\ r2\ p \wedge rv\_at\ testi\ p = (k, j)))$ 
  using  $k\_vio\ k\_in\ Times$  by  $fastforce$ 
}
then have  $k\_ex\_p: \forall k \in \{i .. j\}. (\exists p. (rv\_check\ test\ testi\ r1\ p \wedge rv\_at\ testi\ p = (i, k))$ 
 $\vee (rv\_check\ test\ testi\ r2\ p \wedge rv\_at\ testi\ p = (k, j)))$ 
  by  $auto$ 
then obtain  $f$  where  $f\_def: \forall k \in \{i .. j\}. (rv\_check\ test\ testi\ r1\ (f\ k) \wedge rv\_at\ testi\ (f\ k) = (i, k))$ 
 $\vee (rv\_check\ test\ testi\ r2\ (f\ k) \wedge rv\_at\ testi\ (f\ k) = (k, j))$ 
  using  $bchoice[OF\ k\_ex\_p]$ 
  by  $atomize\_elim\ auto$ 
define  $g$  where  $g = (\lambda k. (rv\_check\ test\ testi\ r1\ (f\ k) \wedge rv\_at\ testi\ (f\ k) = (i, k), f\ k))$ 
then obtain  $ps$  where  $ps\_def: ps = map\ g\ [i ..< Suc\ j]$ 
  by  $auto$ 
then have  $ps\_nnil: ps \neq []$ 
  using  $Times(5)$  by  $auto$ 
then have  $hd\_last\_ps: fst\ (rv\_at\ testi\ (snd\ (hd\ ps))) = i \wedge snd\ (rv\_at\ testi\ (snd\ (last\ ps))) = j$ 
  using  $g\_def\ f\_def\ ps\_def\ upt\_rec[of\ i\ j]$ 
  by  $(auto\ dest: bspec[of\ \_ \_ i]\ bspec[of\ \_ \_ j])$ 

```

```

from ps_def ps_nnil have i_plus_len_eq_j:  $i + \text{length } ps - 1 = j$ 
  by auto

{ fix k
  assume k_in:  $k \in \{0 \dots \text{length } ps\}$ 
  then obtain k' where k'_def:  $k' = i + k \wedge k' \in \{i \dots j\}$ 
    using f_def ps_def ps_nnil Times(5)
    by atomize_elim auto
  then have if fst (ps ! k) then rv_check test testi r1 (snd (ps ! k))  $\wedge$  rv_at testi (snd (ps ! k)) = (i, i
+ k)
    else rv_check test testi r2 (snd (ps ! k))  $\wedge$  rv_at testi (snd (ps ! k)) = (i + k, j)
    using ps_def g_def f_def k_in
    by (auto simp: nth_append dest: bspec[of _ _ j])
  } note k_ps_vio = this

then show ?case
  using Times rv_check.simps(5)[of test testi r1 r2 ps]
  rv_at.simps(5)[of testi ps] hd_last_ps k_vio i_plus_len_eq_j ps_nnil
  by (auto intro!: exI[of _ VTimes ps] simp: split_beta)
next
case (Star r')
then obtain S and T where S_def:  $i \in S$  and T_def:  $j \in T$  and
  ST_props:  $S \cap T = \{\}$   $\wedge$   $S \cup T = \{i \dots j\}$  and
  st_vio:  $\forall (s, t) \in \text{rm } (S \times T). \text{Regex\_Proof\_System.VIO } \text{vio } s \ t \ r'$ 
  using VIO.simps[of vio i j Regex.Star r']
  by auto
then have finiteS: finite S and finiteT: finite T
  using Un_infinite[of S T] infinite_Un[of S T]
  by auto
from ST_props finiteS finiteT S_def T_def
have i_min_un:  $i = \text{Min } (S \cup T)$  and j_max_un:  $j = \text{Max } (S \cup T)$ 
  by (auto simp: Star.prem(3) antisym)
from i_min_un have i_min:  $i = \text{Min } S$ 
  using S_def ST_props finiteS subsetD[of S S \cup T] Min_eqI[of S i]
  by fastforce
from j_max_un have j_max:  $j = \text{Max } T$ 
  using T_def ST_props finiteT subsetD[of T S \cup T] Max_eqI[of T j]
  by fastforce
from finiteS finiteT have rm_finite: finite (rm (S \times T))
  by (auto simp add: Collect_case_prod_Sigma)

then have st_ex_p:  $\forall k \in \text{rm } (S \times T). \exists p. \text{rv\_check test testi } r' \ p \wedge \text{rv\_at testi } p = k$ 
  using st_vio Star by auto
then obtain f where f_def:  $\forall (s, t) \in \text{rm } (S \times T). \text{rv\_check test testi } r' \ (f \ (s, t)) \wedge \text{rv\_at testi } (f$ 
(s, t) = (s, t)
  using bchoice[OF st_ex_p]
  by atomize_elim auto
define ps where ps = map f (sorted_list_of_set (rm (S \times T)))
then have ps_nnil:  $ps \neq []$ 
  using ST_props S_def T_def Star(4) sorted_list_of_set[of rm (S \times T)] rm_finite by fastforce
from ps_def have ps_check:  $\forall k \in \{0 \dots \text{length } ps\}. \text{rv\_check test testi } r' \ (ps ! k)$ 
  using f_def set_sorted_list_of_set[of rm (S \times T)] rm_finite
  nth_mem[of _ ps] set_map[of f sorted_list_of_set (rm (S \times T))] by force

have map_eq:  $\text{map } (\text{rv\_at testi}) \ ps = \text{sorted\_list\_of\_set } (\text{rm } (S \times T))$ 
  using set_sorted_list_of_set[OF rm_finite] ps_def f_def
  by (auto intro: map_idI)

```

```

{fix k
  assume k_def: k ∈ T ∧ (∀ j ∈ S. k ≤ j)
  then have ¬ (∃ k' ∈ rm (S × T). snd k' = k)
    by auto
  then have k ≤ i
    using k_def T_def S_def
    by auto
  then have False
    using k_def ST_props S_def T_def j_max_un_antisym
    by fastforce
  then have ∃ j ∈ S. j < k
    by auto
}note * = this
then have ∀ k ∈ T. ∃ j ∈ S. j < k
  using not_le_imp_less
  by blast
then have ∀ k ∈ T. ∃ k' ∈ rm (S × T). snd k' = k
  by force
then have t_snd: ∀ k ∈ T. ∃ k' ∈ set ps. snd (rv_at testi k') = k
  using ps_def f_def set_sorted_list_of_set[OF rm_finite]
  by fastforce

{fix k
  assume k_def: k ∈ S ∧ (∀ j ∈ T. k ≥ j)
  then have ¬ (∃ k' ∈ rm (S × T). fst k' = k)
    by auto
  then have k ≥ j
    using k_def T_def
    by auto
  then have False
    using k_def ST_props S_def T_def j_max_un_antisym
    by fastforce
  then have ∃ j ∈ T. k < j
    by auto
}
then have ∀ k ∈ S. ∃ j ∈ T. k < j
  using not_le_imp_less
  by blast
then have ∀ k ∈ S. ∃ k' ∈ rm (S × T). fst k' = k
  by force
then have ∀ k ∈ S. ∃ k' ∈ set ps. fst (rv_at testi k') = k
  using ps_def f_def set_sorted_list_of_set[OF rm_finite]
  by fastforce
then have st_map: set (map (fst ∘ (rv_at testi)) ps) = S ∧ set (map (snd ∘ (rv_at testi)) ps) = T
  using ps_def f_def rm_finite sorted_list_of_set[of rm (S × T)] t_snd by auto

then show ?case
  using Star(4) rv_check.simps(6)[of test testi r' ps] rv_at.simps(6)[of testi ps]
  j_max i_min ps_check st_map map_eq S_def T_def ST_props
  by (auto intro!: exI[of _ VStar ps])
qed

lemma rs_check_exec_rs_check:
  fixes test :: 'a ⇒ 'b ⇒ bool
  and testi :: 'b ⇒ nat
  and test' :: ('n ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ bool
  and FV :: 'a ⇒ 'n set
  and C :: 'n set ⇒ ('n ⇒ 'd) set

```

```

assumes  $C\_nonemptyI: \bigwedge A. C A \neq \{\}$ 
and  $C\_union\_eq: \bigwedge X Y. C (X \cup Y) = C X \cap C Y$ 
and  $C\_Union\_eq: \bigwedge X (Y :: 'a \Rightarrow \_). C (\bigcup (Y ' X)) = (\bigcap x \in X. C (Y x))$ 
and  $C\_extensible: \bigwedge X Y v. v \in C X \Longrightarrow X \subseteq Y \Longrightarrow \exists v'. v' \in C Y \wedge (\forall x \in X. v x = v' x)$ 
and  $cong: \bigwedge v v' x sp. \forall a \in FV x. v a = v' a \Longrightarrow test' v x sp = test' v' x sp$ 
shows  $(\bigwedge x sp. x \in regex.atms r \Longrightarrow test x sp = (\forall v \in C (FV x). test' v x sp)) \Longrightarrow$ 
 $rs\_check test testi r rsp = (\forall v \in \bigcap x \in regex.atms r. C (FV x). rs\_check (test' v) testi r rsp)$ 
proof (induct r arbitrary: rsp)
  case (Skip x)
  then show ?case
  by (cases rsp) auto
next
  case (Test x)
  with  $C\_nonemptyI$ [of FV x] show ?case
  by (cases rsp) auto
next
  case (Plus r1 r2)
  with  $C\_nonemptyI$ [of  $Regex.collect FV r1 \cup Regex.collect FV r2$ ] show ?case
  proof (cases rsp)
    case (SPlusL sp)
    with Plus show ?thesis
    by (auto 0 4 dest:  $C\_extensible$ [of  $\_ Regex.collect FV r1 Regex.collect FV r1 \cup Regex.collect FV r2,$ 
       $simplified collect\_alt C\_union\_eq C\_Union\_eq INT\_iff]$ 
      elim!:  $rs\_check\_cong$ [of  $\_ \_ \_ test' \_ test' \_ testi testi,$  THEN iffD1, rotated -1, OF  $\_ refl$ 
      cong refl])
    next
    case (SPlusR sp)
    with Plus show ?thesis
    by (auto 0 4 dest:  $C\_extensible$ [of  $\_ Regex.collect FV r2 Regex.collect FV r1 \cup Regex.collect FV r2,$ 
       $simplified collect\_alt C\_union\_eq C\_Union\_eq INT\_iff]$ 
      elim!:  $rs\_check\_cong$ [of  $\_ \_ \_ test' \_ test' \_ testi testi,$  THEN iffD1, rotated -1, OF  $\_ refl$ 
      cong refl])
    qed (auto simp: collect_alt INT_Un C_Union_eq C_union_eq)
  next
  case (Times r1 r2)
  note  $*$  =  $C\_nonemptyI$ [of  $Regex.collect FV r1 \cup Regex.collect FV r2,$ 
     $simplified collect\_alt INT_Un C\_Union\_eq C\_union\_eq]$ 
  from Times  $*$  show ?case
  proof (cases rsp)
    case (STimes sp1 sp2)
    from Times show ?thesis
    unfolding STimes  $rs\_check.simps regex.set INT\_Un ball\_conj\_distrib ball\_triv\_nonempty$ [OF  $*$ ]
    by (auto 0 4
      dest:  $C\_extensible$ [of  $\_ Regex.collect FV r1 Regex.collect FV r1 \cup Regex.collect FV r2,$ 
         $simplified collect\_alt C\_union\_eq C\_Union\_eq INT\_iff]$ 
       $C\_extensible$ [of  $\_ Regex.collect FV r2 Regex.collect FV r1 \cup Regex.collect FV r2,$ 
         $simplified collect\_alt C\_union\_eq C\_Union\_eq INT\_iff]$ 
      elim!:  $rs\_check\_cong$ [of  $\_ \_ \_ test' \_ test' \_ testi testi,$  THEN iffD1, rotated -1, OF  $\_ refl$ 
      cong refl])
    qed auto
  next
  case (Star r)
  with  $C\_nonemptyI$ [of  $Regex.collect FV r$ ] show ?case
  by (cases rsp) (auto simp: collect_alt C_Union_eq)
  qed

lemma  $rv\_check\_exec\_rv\_check:$ 
  fixes  $test :: 'a \Rightarrow 'b \Rightarrow bool$ 

```

```

and testi :: 'b ⇒ nat
and test' :: ('n ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ bool
and FV :: 'a ⇒ 'n set
and C :: 'n set ⇒ ('n ⇒ 'd) set
assumes C_nonemptyI:  $\bigwedge A. C A \neq \{\}$ 
and C_union_eq:  $\bigwedge X Y. C (X \cup Y) = C X \cap C Y$ 
and C_Union_eq:  $\bigwedge X (Y :: 'a \Rightarrow \_). C (\bigcup (Y ' X)) = (\bigcap x \in X. C (Y x))$ 
and C_extensible:  $\bigwedge X Y v. v \in C X \implies X \subseteq Y \implies \exists v'. v' \in C Y \wedge (\forall x \in X. v x = v' x)$ 
and cong:  $\bigwedge v v' x sp. \forall a \in FV x. v a = v' a \implies test' v x sp = test' v' x sp$ 
shows ( $\bigwedge x sp. x \in regex.atms r \implies test x sp = (\forall v \in C (FV x). test' v x sp)$ )  $\implies$ 
rv_check test testi r rsp = ( $\forall v \in \bigcap x \in regex.atms r. C (FV x). rv\_check (test' v) testi r rsp$ )
proof (induct r arbitrary: rsp)
  case (Skip x)
  then show ?case
  by (cases rsp) auto
next
  case (Test x)
  with C_nonemptyI[of FV x] show ?case
  by (cases rsp) auto
next
  case (Plus r1 r2)
  note * = C_nonemptyI[of Regex.collect FV r1  $\cup$  Regex.collect FV r2,
    simplified collect_alt INT_Un C_Union_eq C_union_eq]
  from Plus * show ?case
  proof (cases rsp)
    case (VPlus vp1 vp2)
    from Plus show ?thesis
    unfolding VPlus rv_check.simps regex.set INT_Un ball_conj_distrib ball_triv_nonempty[OF *]
    by (auto 0 4
      dest: C_extensible[of _ Regex.collect FV r1 Regex.collect FV r1  $\cup$  Regex.collect FV r2,
        simplified collect_alt C_union_eq C_Union_eq INT_iff]
        C_extensible[of _ Regex.collect FV r2 Regex.collect FV r1  $\cup$  Regex.collect FV r2,
        simplified collect_alt C_union_eq C_Union_eq INT_iff]
        elim!: rv_check_cong[of _ _ _ test' _ test' _ testi testi, THEN iffD1, rotated -1, OF _ refl
        cong refl])
    qed auto
  next
  case (Times r1 r2)
  note * = C_nonemptyI[of Regex.collect FV r1  $\cup$  Regex.collect FV r2,
    simplified collect_alt INT_Un C_Union_eq C_union_eq]
  from Times * show ?case
  proof (cases rsp)
    case (VTimes ps)
    from Times have IH: if fst (ps ! k)
      then rv_check test testi r1 (snd (ps ! k)) = ( $\forall v \in \bigcap x \in regex.atms r1. C (FV x). rv\_check (test' v)$ 
      testi r1 (snd (ps ! k)))
      else rv_check test testi r2 (snd (ps ! k)) = ( $\forall v \in \bigcap x \in regex.atms r2. C (FV x). rv\_check (test' v)$ 
      testi r2 (snd (ps ! k)))
    if k < length ps for k
    using that by auto
    show ?thesis
    unfolding VTimes rv_check.simps regex.set INT_Un ball_conj_distrib ball_triv_nonempty[OF *]
    ex_simps simp_thms ball_swap[of _ {0 ..< length ps}] Let_def split_beta ball_if_distrib
    by (intro conj_cong refl ball_cong if_cong)
    (auto 0 4 simp: IH
      dest: C_extensible[of _ Regex.collect FV r1 Regex.collect FV r1  $\cup$  Regex.collect FV r2,
        simplified collect_alt C_union_eq C_Union_eq INT_iff]
        C_extensible[of _ Regex.collect FV r2 Regex.collect FV r1  $\cup$  Regex.collect FV r2,

```

```

      simplified collect_alt C_union_eq C_Union_eq INT_iff]
      elim!: rv_check_cong[of _ _ _ test' _ test' _ testi testi, THEN iffD1, rotated -1, OF _ refl
cong refl])
    qed auto
  next
    case (Star r)
    note * = C_nonemptyI[of Regex.collect FV r, simplified collect_alt INT_Un C_Union_eq]
    with Star show ?case
    proof (cases rsp)
      case (VStar vps)
      then show ?thesis
        unfolding VStar rv_check.simps regex.set INT_Un ball_conj_distrib ball_triv_nonempty[OF *]
          ex_simps simp_thms ball_swap[of _ {0 ..< length vps}]
        by (intro conj_cong refl ball_cong Star) simp
    qed (auto simp: collect_alt C_Union_eq)
  qed

```

lemma chain_sorted1:

```

  fixes f :: _ ⇒ nat × nat
  assumes ∀ k ∈ {Suc 0..<length ps}. fst (f (ps ! k)) = snd (f (ps ! (k - Suc 0)))
  and ∀ k ∈ {0..<length ps}. fst (f (ps ! k)) < snd (f (ps ! k))
  and j ≤ k k < length ps
  shows fst (f (ps ! j)) ≤ fst (f (ps ! k))
  using assms
proof (induct k - j arbitrary: j)
  case (Suc x)
  then show ?case
    by (cases k) (force simp: less_Suc_eq dest!: bspec[of _ _ j] meta_spec[of _ Suc j])+
qed simp

```

lemma chain_sorted2:

```

  fixes f :: _ ⇒ nat × nat
  assumes ∀ k ∈ {Suc 0..<length ps}. fst (f (ps ! k)) = snd (f (ps ! (k - Suc 0)))
  and ∀ k ∈ {0..<length ps}. fst (f (ps ! k)) < snd (f (ps ! k))
  and j ≤ k k < length ps
  shows snd (f (ps ! j)) ≤ snd (f (ps ! k))
  using assms
proof (induct k - j arbitrary: j)
  case (Suc x)
  then show ?case
    by (cases k) (force simp: less_Suc_eq dest!: bspec[of _ _ Suc j] meta_spec[of _ Suc j])+
qed simp

```

context

```

  fixes test :: 'a ⇒ 'b ⇒ bool and testi :: 'b ⇒ nat and SAT sat
  assumes test_sound: ∀ x ∈ regex.atms r. ∀ p' ∈ spatms rsp. test x p' → SAT (testi p') x
  and SAT_sound: ∀ x ∈ regex.atms r. ∀ i. SAT i x → sat i x
begin

```

lemma rs_check_le:

```

  rs_check test testi r rsp ⇒ fst (rs_at testi rsp) ≤ snd (rs_at testi rsp)
  by (drule rs_check_sound[OF test_sound], drule soundness_SAT[OF SAT_sound], drule match_le)

```

lemma rs_check_le1:

```

  rs_check test testi r rsp ⇒ sp ∈ spatms rsp ⇒ fst (rs_at testi rsp) ≤ testi sp
proof (induct r rsp rule: rs_check.induct)
  case (γ r ps)
  then show ?case

```

```

    by (fastforce simp: in_set_conv_nth hd_conv_nth
        intro: order_trans[OF chain_sorted1[of ps rs_at testi 0]])
qed (auto dest: rs_check_le)

lemma rs_check_le2:
  rs_check test testi r rsp  $\implies$  sp  $\in$  spatms rsp  $\implies$  testi sp  $\leq$  snd (rs_at testi rsp)
proof (induct r rsp rule: rs_check.induct)
  case (7 r ps)
  then show ?case
    by (fastforce simp: in_set_conv_nth last_conv_nth
        intro: order_trans[OF chain_sorted2[of ps rs_at testi _ length ps - Suc 0]])
qed (auto dest: rs_check_le)

end

lemma rv_check_le:
  rv_check test testi r rvp  $\implies$  vp  $\in$  vpatms rvp  $\implies$  fst (rv_at testi rvp)  $\leq$  snd (rv_at testi rvp)
  by (induct r rvp rule: rv_check.induct) (auto simp: neq_Nil_conv)

lemma rv_check_le2:
  rv_check test testi r rvp  $\implies$  vp  $\in$  vpatms rvp  $\implies$  testi vp  $\leq$  snd (rv_at testi rvp)
proof (induct r rvp rule: rv_check.induct)
  case (5 r r' ps)
  from 5(4) obtain b i rvp where *: i < length ps ps ! i = (b, rvp) vp  $\in$  vpatms rvp
  unfolding rvproof.set UN_iff Bex_def in_set_conv_nth by auto
  show ?case
  proof (cases b)
    case True
    with * 5(1)[of i ps ! i b rvp] 5(3) show ?thesis
      by (auto dest: bspec[of _ _ i])
    next
    case False
    with * 5(2)[of i ps ! i b rvp] 5(3) show ?thesis
      by (auto dest: bspec[of _ _ i])
  qed
next
  case (6 r ps)
  from 6(3) obtain i rvp where *: i < length ps ps ! i = rvp vp  $\in$  vpatms rvp
  unfolding rvproof.set UN_iff Bex_def in_set_conv_nth by auto
  with 6(1)[of i] 6(2) show ?case
    by (auto elim!: order_trans)
qed auto

```

9 Proof Checker

unbundle MFOTL_syntax

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

begin

```

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

```

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

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

```

(case right I of  $\infty \Rightarrow \text{True}$  |  $\text{enat } b \Rightarrow j < \text{LTP\_f } \sigma \ i \ b) \wedge i \leq j$ 
 $\wedge \text{map } v\_at \ \text{vp2s} = [\text{ETP\_f } \sigma \ i \ I \ ..< j + 1] \wedge v\_check \ v \ \varphi \ \text{vp1}$ 
 $\wedge (\forall \text{vp2} \in \text{set } \text{vp2s}. v\_check \ v \ \psi \ \text{vp2}))$ 
| ( $\varphi \ \mathbf{U} \ I \ \psi, \text{VUntilInf } i \ \text{hi} \ \text{vp2s}$ )  $\Rightarrow$ 
( $\text{hi} = (\text{case right } I \ \text{of } \text{enat } b \Rightarrow \text{LTP\_f } \sigma \ i \ b) \wedge \text{right } I \neq \infty$ 
 $\wedge \text{map } v\_at \ \text{vp2s} = [\text{ETP\_f } \sigma \ i \ I \ ..< \text{hi} + 1]$ 
 $\wedge (\forall \text{vp2} \in \text{set } \text{vp2s}. v\_check \ v \ \psi \ \text{vp2}))$ )
| ( $\triangleleft I \ r, \text{VMatchPOut } i$ )  $\Rightarrow \tau \ \sigma \ i < \tau \ \sigma \ 0 + \text{left } I$ 
| ( $\triangleleft I \ r, \text{VMatchP } i \ \text{rvps}$ )  $\Rightarrow$ 
( $\text{let } j = \text{ETP } \sigma \ (\text{case right } I \ \text{of } \infty \Rightarrow 0 \ | \ \text{enat } n \Rightarrow \tau \ \sigma \ i - n)$ 
 $\text{in } \tau \ \sigma \ i \geq \tau \ \sigma \ 0 + \text{left } I \wedge \text{map } (\text{fst} \circ \text{rv\_at } v\_at) \ \text{rvps} = [j \ ..< \text{Suc } (\text{LTP\_p } \sigma \ i \ I)] \wedge$ 
 $(\forall \text{rvp} \in \text{set } \text{rvps}. \text{rv\_check } (v\_check \ v) \ v\_at \ r \ \text{rvp} \wedge \text{snd } (\text{rv\_at } v\_at \ \text{rvp}) = i)$ )
| ( $\triangleright I \ r, \text{VMatchF } i \ \text{rvps}$ )  $\Rightarrow$ 
( $\text{let } j = \text{LTP } \sigma \ (\text{case right } I \ \text{of } \infty \Rightarrow 0 \ | \ \text{enat } n \Rightarrow \tau \ \sigma \ i + n)$ 
 $\text{in } \text{map } (\text{snd} \circ \text{rv\_at } v\_at) \ \text{rvps} = [\text{ETP\_f } \sigma \ i \ I \ ..< \text{Suc } j] \wedge \text{right } I \neq \infty \wedge$ 
 $(\forall \text{rvp} \in \text{set } \text{rvps}. \text{rv\_check } (v\_check \ v) \ v\_at \ r \ \text{rvp} \wedge \text{fst } (\text{rv\_at } v\_at \ \text{rvp}) = i)$ )
| ( $\_ , \_$ )  $\Rightarrow \text{False}$ 

```

```

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)

```

9.1 Checker Soundness

lemma *check_soundness*:

$s_check \ v \ \varphi \ sp \Longrightarrow \text{SAT } \sigma \ v \ (s_at \ sp) \ \varphi$

$v_check \ v \ \varphi \ vp \Longrightarrow \text{VIO } \sigma \ v \ (v_at \ vp) \ \varphi$

proof (*induction* *sp* and *vp* arbitrary: $v \ \varphi$ and $v \ \varphi$)

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  $\varphi$ ) (auto intro: SAT_VIO.SIffVV)
next
  case (SExists  $x z sp$ )
  with SExists(1)[of  $v(x := z)$ ] show ?case
  by (cases  $\varphi$ ) (auto intro: SAT_VIO.SExists)
next
  case (SForall  $x part$ )
  then show ?case
  proof (cases  $\varphi$ )
  case (Forall  $y \psi$ )
  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  $z \in D$  ( $D, ?sp$ )  $\in$  SubVals part
  by blast
  with SForall(2-) Forall have  $s\_check (v(y := z)) \psi ?sp s\_at ?sp = s\_at (SForall x part)$ 
  by auto
  then show SAT  $\sigma (v(y := z)) (s\_at (SForall x part)) \psi$ 
  by (auto simp del: fun_upd_apply dest!: SForall(1)[rotated])
  qed
  qed auto
next
  case (SSince  $sps$ )
  then show ?case
  proof (cases  $\varphi$ )
  case (Since  $I \psi$ )
  show ?thesis
  using SSince(3)
  unfolding Since
  proof (intro SAT_VIO.SSince[of  $s\_at sps$ ], goal_cases le mem SAT $\psi$  SAT $\varphi$ )
  case (SAT $\varphi$   $k$ )
  then show ?case
  by (cases  $k \leq s\_at (hd sps)$ )
  (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  $\varphi$ )
  case (Once  $I \varphi$ )
  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  $\varphi$ )
  case (Eventually  $I \varphi$ )
  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  $\varphi$ ) (auto intro: SAT_VIO.SHistoricallyOut)
  next
    case (SHistorically i li sps)
    then show ?case
    proof (cases  $\varphi$ )
      case (Historically I  $\varphi$ )
      {fix k
        define j where j_def:  $j \equiv \text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \text{ETP } \sigma (\tau \sigma i - n)$ 
        assume k_def:  $k \geq j \wedge k \leq i \wedge k \leq \text{LTP } \sigma (\tau \sigma i - \text{left } I)$ 
        from SHistorically Historically j_def have map:  $\text{set } (\text{map } s\_at \text{ sps}) = \text{set } [j \dots < \text{Suc } (\text{LTP}_p \sigma i$ 
I)]
          by (auto simp: Let_def)
        then have kset:  $k \in \text{set } ([j \dots < \text{Suc } (\text{LTP}_p \sigma i I)])$  using j_def k_def by auto
        then obtain x where  $x: x \in \text{set } \text{ sps } s\_at \ x = k$  using k_def map
          unfolding set_map set_eq_iff image_iff
          by metis
        then have  $\text{SAT } \sigma \ v \ k \ \varphi$  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:  $j \equiv \text{LTP } \sigma (\tau \sigma i + n)$ 
        assume k_def:  $k \leq j \wedge k \geq i \wedge k \geq \text{ETP } \sigma (\tau \sigma i + \text{left } I)$ 
        from SAlways Always j_def have map:  $\text{set } (\text{map } s\_at \ \text{ sps}) = \text{set } [(\text{ETP}_f \sigma i I) \dots < \text{Suc } j]$ 
          by (auto simp: Let_def n_def)
        then have kset:  $k \in \text{set } [(\text{ETP}_f \sigma i I) \dots < \text{Suc } j]$  using k_def j_def by auto
        then obtain x where  $x: x \in \text{set } \text{ sps } s\_at \ x = k$  using k_def map
          unfolding set_map set_eq_iff image_iff
          by metis
        then have  $\text{SAT } \sigma \ v \ k \ \varphi$  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 SATψ SATφ)
    case (SATφ k)
    then show ?case
      by (cases k ≤ s_at (hd sps))
        (auto 0 3 simp: Let_def elim: map_setE[of _ _ k] intro: SUntil(1))
    qed (auto simp: Let_def intro: SUntil(2))
  qed auto
next
case (SNext sp)
then show ?case by (cases φ) (auto simp add: Let_def SAT_VIO.SNext)
next
case (SPrev sp)
then show ?case by (cases φ) (auto simp add: Let_def SAT_VIO.SPrev)
next
case (SMatchP rsp)
then show ?case
  by (cases φ) (auto intro: SAT_VIO.SMatchP dest!: rs_check_sound[rotated, where sat=SAT σ v])
next
case (SMatchF rsp)
then show ?case
  by (cases φ) (auto intro: SAT_VIO.SMatchF dest!: rs_check_sound[rotated, where sat=SAT σ v])
next
case VFF
then show ?case by (cases φ) (auto intro: SAT_VIO.VFF)
next
case VPred
then show ?case by (cases φ) (auto intro: SAT_VIO.VPred)
next
case VEq_Const
then show ?case by (cases φ) (auto intro: SAT_VIO.VEq_Const)
next
case VNeg
then show ?case by (cases φ) (auto intro: SAT_VIO.VNeg)
next
case VOr
then show ?case by (cases φ) (auto intro: SAT_VIO.VOr)
next
case VAndL
then show ?case by (cases φ) (auto intro: SAT_VIO.VAndL)
next
case VAndR
then show ?case by (cases φ) (auto intro: SAT_VIO.VAndR)
next
case VImp
then show ?case by (cases φ) (auto intro: SAT_VIO.VImp)
next
case VIffSV
then show ?case by (cases φ) (auto intro: SAT_VIO.VIffSV)
next
case VIffVS
then show ?case by (cases φ) (auto intro: SAT_VIO.VIffVS)
next
case (VExists x part)
then show ?case
  proof (cases φ)
    case (Exists y ψ)

```

```

show ?thesis unfolding Exists
proof (intro SAT_VIO.VExists allI)
  fix z
  let ?vp = lookup_part part z
  from lookup_part_SubVals[of z part] obtain D where z ∈ D (D, ?vp) ∈ SubsVals part
  by blast
  with VExists(2-) Exists have v_check (v(y := z)) ψ ?vp v_at ?vp = v_at (VExists x part)
  by auto
  then show VIO σ (v(y := z)) (v_at (VExists x part)) ψ
  by (auto simp del: fun_upd_apply dest!: VExists(1)[rotated])
qed
qed auto
next
case (VForall x z vp)
with VForall(1)[of v(x := z)] show ?case
by (cases φ) (auto intro: SAT_VIO.VForall)
next
case VOnceOut
then show ?case by (cases φ) (auto intro: SAT_VIO.VOnceOut)
next
case (VOnce i li vps)
then show ?case
proof (cases φ)
  case (Once I φ)
  {fix k
   define j where j_def: j ≡ case right I of ∞ ⇒ 0 | enat n ⇒ ETP σ (τ σ i - n)
   assume k_def: k ≥ j ∧ k ≤ i ∧ k ≤ LTP σ (τ σ i - left I)
   from VOnce Once j_def have map: set (map v_at vps) = set [j ..< Suc (LTP_p σ i I)]
   by (auto simp: Let_def)
   then have kset: k ∈ set ([j ..< Suc (LTP_p σ i I)]) using j_def k_def by auto
   then obtain x where x: x ∈ set vps v_at x = k using k_def map
   unfolding set_map set_eq_iff image_iff
   by metis
   then have VIO σ v k φ using VOnce unfolding Once
   by (auto simp: Let_def)
  } note * = this
show ?thesis
  using VOnce *
  unfolding Once
  by (auto simp: Let_def intro!: SAT_VIO.VOnce)
qed (auto intro: SAT_VIO.intros)
next
case (VEventually i hi vps)
then show ?case
proof (cases φ)
  case (Eventually I φ)
  obtain n where n_def: right I = enat n
  using VEventually
  by (auto simp: Eventually split: enat.splits)
  {fix k
   define j where j_def: j ≡ LTP σ (τ σ i + n)
   assume k_def: k ≤ j ∧ k ≥ i ∧ k ≥ ETP σ (τ σ i + left I)
   from VEventually Eventually j_def have map: set (map v_at vps) = set [(ETP_f σ i I) ..< Suc j]
   by (auto simp: Let_def n_def)
   then have kset: k ∈ set [(ETP_f σ i I) ..< Suc j] using k_def j_def by auto
   then obtain x where x: x ∈ set vps v_at x = k using k_def map
   unfolding set_map set_eq_iff image_iff
   by metis
  }

```

```

    then have VIO  $\sigma$  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  $map: set\ (map\ v\_at\ vps) = set\ [(ETP\_f\ \sigma\ i\ I) ..<\ Suc\ (v\_at\ vp)]$ 
  by  $(auto\ simp: Let\_def)$ 
then have  $kset: k \in set\ [(ETP\_f\ \sigma\ i\ I) ..<\ Suc\ (v\_at\ vp)]$  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  $VUntil$  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  $map: set\ (map\ v\_at\ vps) = set\ [j ..<\ Suc\ (LTP\_p\ \sigma\ i\ I)]$ 
  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  $x \in set\ vps\ v\_at\ x = k$  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:  $k \leq j \wedge k \geq i \wedge k \geq ETP \sigma (\tau \sigma i + \text{left } I)$ 
  from VUntilInf Until 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:  $k \in \text{set } [(ETP_f \sigma i I) ..< Suc j]$  using k_def j_def by auto
  then obtain x where x:  $x \in \text{set } vps \ v\_at \ x = k$  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)
next
case (VMatchPOut i rvps)
then show ?case by (cases  $\varphi$ ) (auto intro: SAT_VIO.VMatchPOut)
next
case (VMatchP i rvps)
then show ?case
proof (cases  $\varphi$ )
  case (MatchP I r)
  then have vio:  $\bigwedge rvp. rvp \in \text{set } rvps \implies \text{Regex\_Proof\_System.VIO } (VIO \sigma v) (fst (rv\_at \ v\_at \ rvp))$ 
  (snd (rv\_at \ v\_at \ rvp)) r
  using rv_check_sound[of r _ v_check v VIO  $\sigma$  v v_at] VMatchP MatchP
  by (auto simp: Let_def)
  {fix k
  define j where j_def:  $j \equiv ETP \sigma (\text{case right } I \text{ of } \infty \implies 0 \mid \text{enat } n \implies \tau \sigma i - n)$ 
  assume k_def:  $k \geq j \wedge k \leq i \wedge k \leq LTP \sigma (\tau \sigma i - \text{left } I)$ 
  from VMatchP MatchP j_def have map: set (map (fst  $\circ$  rv_at v_at) rvps) = set [j ..< Suc (LTP_p
   $\sigma$  i I)]
  by (auto simp: Let_def)
  then have kset:  $k \in \text{set } [j ..< Suc (LTP_p \sigma i I)]$  using k_def j_def by auto
  then obtain rvp where rvp:  $rvp \in \text{set } rvps \ \text{fst } (rv\_at \ v\_at \ rvp) = k$ 
  using k_def kset map
  by (auto simp: i_LTP_tau set_eq_iff image_iff dest: spec[of _ k] simp del: upt.simps)
  then have Regex_Proof_System.VIO (VIO  $\sigma$  v) k i r using VMatchP MatchP vio[of rvp]
  by (auto simp: Let_def)
} note * = this
then show ?thesis using VMatchP MatchP
by (auto simp: i_ETP_tau intro!: SAT_VIO.VMatchP split: enat.splits)
qed(auto intro: SAT_VIO.intros)
next
case (VMatchF i rvps) then show ?case
proof (cases  $\varphi$ )

```

```

case (MatchF I r)
then have vio:  $\bigwedge rvp. rvp \in \text{set } rvp \implies \text{Regex\_Proof\_System.VIO } (VIO \sigma v) (fst (rv\_at v\_at rvp))$ 
(snd (rv\_at v\_at rvp)) r
using rv\_check\_sound[of r _ v\_check v VIO  $\sigma$  v v\_at] VMatchF MatchF
by (auto simp: Let\_def)
{ fix k
define j where j\_def:  $j \equiv LTP \sigma$  (case right I of  $\infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \tau \sigma i + n$ )
assume k\_def:  $k \leq j \wedge k \geq i \wedge k \geq ETP \sigma (\tau \sigma i + \text{left } I)$ 
from VMatchF MatchF j\_def have map:  $\text{set } (\text{map } (\text{snd} \circ rv\_at v\_at) rvp) = \text{set } [ETP\_f \sigma i I$ 
..< Suc j]
by (auto simp: Let\_def)
then have kset:  $k \in \text{set } ([ETP\_f \sigma i I ..< \text{Suc } j])$  using k\_def j\_def by auto
then obtain rvp where rvp:  $rvp \in \text{set } rvp$  snd (rv\_at v\_at rvp) = k
using k\_def kset map
by (auto simp: i\_LTP\_tau set\_eq\_iff image\_iff dest: spec[of _ k] simp del: upt.simps)
then have Regex\_Proof\_System.VIO (VIO  $\sigma$  v) i k r using VMatchF MatchF vio[of rvp]
by (auto simp: Let\_def)
} note * = this
then show ?thesis using VMatchF MatchF
by (auto simp: Let\_def intro!: SAT\_VIO.VMatchF)
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 $\longleftrightarrow v \in \text{compatible_vals } X \text{ vs}$
by (auto simp: compatible_def compatible_vals_def)

lemma compatible_empty_iff: compatible {} vs v $\longleftrightarrow \text{True}$

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_vals_Union_eq:

compatible_vals ($\bigcup_{x \in X} Y x$) vs = ($\bigcap_{x \in X} \text{compatible_vals } (Y x) \text{ vs}$)
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:

$Y \subseteq X \implies \text{compatible_vals } X \text{ vs} \subseteq \text{compatible_vals } Y \text{ vs}$
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* ($\lambda x. SOME\ y. y \in vs\ x$) (*Y-X*)])

lemmas *compatible_vals_extensible* = *compatible_extensible*[*unfolded compatible_alt*]

primrec *mk_values* :: (('n, 'd) *trm* \times 'a *set*) *list* \Rightarrow 'a *list set*

where *mk_values* [] = {[]}
| *mk_values* (*T* # *Ts*) = (*case T of*
 (**v** *x*, *X*) \Rightarrow
 let terms = *map fst Ts in*
 if v x \in *set terms* *then*
 let fst_pos = *hd (positions terms (v x)) in* ($\lambda xs. (xs\ !\ fst_pos)\ \#\ xs$) ' (*mk_values Ts*)
 else set_Cons X (mk_values Ts)
| (**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 *set (map snd tXs)* \implies *tXs* \neq [] \implies *vs* \in *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*: **v** *x* \in *set (map fst tXs)* \implies

n \in *set (positions (map fst tXs) (v x))* \implies
m \in *set (positions (map fst tXs) (v x))* \implies
vs \in *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 (let (fintXs, inftXs) = partition ($\lambda tX. \text{finite (snd } tX)$) tXs in
 if inftXs = [] then {p} \times mk_values tXs \subseteq X
 else let inf_dups = filter ($\lambda tX. (\text{fst } tX) \in \text{set (map fst fintXs)}$) 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 ($\lambda tX. \text{Max (set (positions tXs } tX)) < \text{Max (set (positions (map fst tXs) (fst } tX))$)
 inf_dups
 then {p} \times mk_values tXs \subseteq X
 else (if finite X then False else Code.abort STR "subset on infinite subset" ($\lambda_. \{p\} \times$ 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*: $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies tY \in \text{set } tXs \implies$
 $\forall Y. (\text{fst } tY, Y) \in \text{set } tXs \longrightarrow \text{infinite } Y \implies \text{infinite (mk_values } tXs)$

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.prem1)

moreover have $\forall Z. (\text{trm.Var } x, Z) \in \text{set } tXs \longrightarrow \text{infinite } Z$

using Cons.prem2,3) var_in

by (cases tY \in set tXs; clarsimp)

(metis (no_types, lifting) Cons.hyps Cons.prem1)

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 ($\lambda xs. xs ! \text{hd (positions (map fst tXs) (trm.Var } x)) \# xs$)

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.prem1

by (auto intro!: mk_values_nemptyI)

then show ?case

using Cons var_out infinite_set_ConsI(1)[OF $\langle \text{mk_values } tXs \neq \{\} \rangle \langle \text{infinite (snd } tX) \rangle$]

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 ≠ {}
using Cons.prem
by (auto intro!: mk_values_nemptyI)
then show ?case
using Cons const infinite_set_ConsI(1)[OF ‹mk_values tXs ≠ {}› ‹infinite (snd tX)›]
by auto
qed
qed simp

lemma infinite_mk_values2:  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies$ 
 $tY \in \text{set } tXs \implies \text{infinite } (\text{snd } tY) \implies$ 
 $\text{Max } (\text{set } (\text{positions } tXs \ tY)) \geq \text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tY))) \implies$ 
 $\text{infinite } (\text{mk\_values } tXs)$ 
proof (induct tXs arbitrary: tY)
case (Cons tX tXs)
hence obs1:  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\}$ 
by (simp add: Cons.prem(1))
note IH = Cons.hyps[OF obs1 _ ‹infinite (snd tY)›]
have obs2:  $tY \in \text{set } tXs \implies$ 
 $\text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tY))) \leq \text{Max } (\text{set } (\text{positions } tXs \ tY))$ 
using Cons.prem(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
 $\text{trm.split[of infinite] if\_split[of infinite]}$ 
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 (( $\lambda Xs. Xs ! \text{hd } (\text{positions } (\text{map } \text{fst } tXs) (\text{trm.Var } x)) \# Xs$ ) ‹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  $\text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } (tX \# tXs)) (\text{fst } tY)))$ 
 $= \text{Suc } (\text{Max } (\text{set } (\text{positions } (\text{map } \text{fst } tXs) (\text{fst } tY))))$ 
using Cons.prem var_in
by (simp only: list.map(2))
( $\text{subst Max\_set\_positions\_Cons\_tl; force simp: image\_iff}$ )
moreover have  $tY \notin \text{set } tXs \implies \text{Max } (\text{set } (\text{positions } (tX \# tXs) \ tY)) = (0::\text{nat})$ 
using Cons.prem Max_set_positions_Cons_hd by fastforce
ultimately show False
using Cons.prem(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.prem1 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.prem1
  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:  $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies$ 
   $\text{mk\_values\_subset } p \text{ } tXs \ X \iff \{p\} \times \text{mk\_values } tXs \subseteq X$ 
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 bex_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 ∈ 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:  $\text{filter } (\lambda x. \text{infinite } (\text{snd } x)) \ tXs \neq []$ 
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:  $cs \in \text{mk\_values } (vs \llbracket ts \rrbracket) \implies$ 
 $\exists v \in \text{compatible\_vals } (fv \ (r \dagger \ ts)) \ vs. cs = v \llbracket ts \rrbracket$ 
proof (induct ts arbitrary: cs vs)
case (Cons t ts)
show ?case
proof (cases t)
case (Var x)
let  $?Ts = vs \llbracket ts \rrbracket$ 
have  $vs \llbracket (t \# \ ts) \rrbracket = (v \ x, \ vs \ x) \# \ ?Ts$ 
using Var by (simp add: eval_trms_set_def)
show ?thesis
proof (cases v x \in set ts)
case True
then obtain n where  $n\_in: n \in \text{set } (\text{positions } ts \ (v \ x))$ 
and  $nth\_n: ts \ ! \ n = v \ x$ 
by (meson fst_pos_in_positions nth_fst_pos)
hence  $n\_in': n \in \text{set } (\text{positions } (\text{map } \text{fst } ?Ts) \ (v \ x))$ 
by (induct ts arbitrary: n)
 $(\text{auto simp: eval_trms_set_def split: if_splits})$ 
have key:  $v \ x \in \text{set } (\text{map } \text{fst } ?Ts)$ 
using True by (induct ts)
 $(\text{auto simp: eval_trms_set_def})$ 
then obtain a as
where  $as\_head: as \ ! \ (\text{hd } (\text{positions } (\text{map } \text{fst } ?Ts) \ (v \ x))) = a$ 
and  $as\_tail: as \in \text{mk\_values } ?Ts$ 
and  $as\_shape: cs = a \# \ as$ 
using Cons(2)
by (clarsimp simp add: eval_trms_set_def Var image_iff)
obtain v where  $v\_hyps: v \in \text{compatible\_vals } (fv \ (r \dagger \ ts)) \ vs$ 
 $as = v \llbracket ts \rrbracket$ 
using Cons(1)[OF as_tail] by blast
hence  $as' \ nth: as \ ! \ n = v \ x$ 
using  $nth\_n \ \text{positions\_length}$ [OF n_in]
by (simp add: eval_trms_def)
have  $evals\_neq\_Nil: ?Ts \neq []$ 
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.premis
  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_Cmpl r vs ts i = ({r} × mk_values (vs{ts})) ⊆ - Γ σ i

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

lemma mk_values_alt:
  mk_values (vs{ts}) =
  {cs. ∃ v ∈ compatible_vals (∪ (fv_trm ' set ts)) vs. cs = v{ts}}
by (auto dest!: mk_values_sound intro: mk_values_complete)

lemma check_values_neq_NoneI:
assumes v ∈ compatible_vals (∪ (fv_trm ' set ts) - dom f) vs ∧ x y. f x = Some y ⇒ y ∈ vs x
shows check_values vs ts ((λx. case f x of None ⇒ v x | Some y ⇒ y){ts}) (Some f) ≠ None
using assms

```

```

proof (induct ts arbitrary: f)
  case (Cons t ts)
  then show ?case
  proof (cases t)
    case (Var x)
    show ?thesis
  proof (cases f x)
    case None
    with Cons(2) Var have v_in[simp]: v x ∈ vs x
      by (auto simp: compatible_vals_def)
    from Cons(2) have v ∈ compatible_vals (∪ (fv_trm ‘ set ts) – dom (f(x ↦ v x))) vs
      by (auto simp: compatible_vals_def)
    then have check_values vs ts
      ((λz. case (f(x ↦ v x)) z of None ⇒ v z | Some y ⇒ y) [[ts]])
      (Some (f(x ↦ v x))) ≠ None
    using Cons(3) None Var
    by (intro Cons(1)) (auto simp: compatible_vals_def split: if_splits)
  then show ?thesis
    by (elim eq_neq_eq_imp_neq[OF __ refl, rotated])
      (auto simp add: eval_trms_def compatible_vals_def Var None split: if_splits option.splits
        intro!: arg_cong2[of _ _ _ _ check_values vs ts] eval_trm_fv_cong)
  next
    case (Some y)
    with Cons(1)[of f] Cons(2–) Var fun_upd_triv[of f x] show ?thesis
      by (auto simp: domI eval_trms_def compatible_vals_def split: option.splits)
  qed
next
  case (Const c)
  with Cons show ?thesis
    by (auto simp: eval_trms_def compatible_vals_def split: option.splits)
  qed
qed (simp add: eval_trms_def)

```

```

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

```

```

lemma mk_values_subset_Cmpl_code[code]:
  mk_values_subset_Cmpl r vs ts i = (∀ (q, us) ∈ Γ σ i. q ≠ r ∨ check_values vs ts us (Some Map.empty)
= 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])

```

9.2 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{ts}) ( $\Gamma$   $\sigma$  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
  | ( $\varphi \rightarrow_F$   $\psi$ , SImpL vp1)  $\Rightarrow$  v_check_exec vs  $\varphi$  vp1
  | ( $\varphi \rightarrow_F$   $\psi$ , 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)
    in x = y  $\wedge$  ( $\forall$  (sub, sp)  $\in$  SubsVals sp_part. s_at sp = i  $\wedge$  s_check_exec (vs (x := sub))  $\varphi$  sp))
  | (Y I  $\varphi$ , SPrev sp)  $\Rightarrow$ 
    (let j = s_at sp; i = s_at (SPrev sp) in
    i = j+1  $\wedge$  mem ( $\Delta$   $\sigma$  i) I  $\wedge$  s_check_exec vs  $\varphi$  sp)
  | (X I  $\varphi$ , SNext sp)  $\Rightarrow$ 
    (let j = s_at sp; i = s_at (SNext sp) in
    j = i+1  $\wedge$  mem ( $\Delta$   $\sigma$  j) I  $\wedge$  s_check_exec vs  $\varphi$  sp)
  | (P I  $\varphi$ , SOnce i sp)  $\Rightarrow$ 
    (let j = s_at sp in
    j  $\leq$  i  $\wedge$  mem ( $\tau$   $\sigma$  i -  $\tau$   $\sigma$  j) I  $\wedge$  s_check_exec vs  $\varphi$  sp)
  | (F I  $\varphi$ , SEventually i sp)  $\Rightarrow$ 
    (let j = s_at sp in
    j  $\geq$  i  $\wedge$  mem ( $\tau$   $\sigma$  j -  $\tau$   $\sigma$  i) I  $\wedge$  s_check_exec vs  $\varphi$  sp)
  | (H I  $\varphi$ , SHistoricallyOut i)  $\Rightarrow$ 
     $\tau$   $\sigma$  i <  $\tau$   $\sigma$  0 + left I
  | (H I  $\varphi$ , SHistorically i li sps)  $\Rightarrow$ 
    (li = (case right I of  $\infty \Rightarrow$  0 | enat b  $\Rightarrow$  ETP  $\sigma$  ( $\tau$   $\sigma$  i - b))
     $\wedge$   $\tau$   $\sigma$  0 + left I  $\leq$   $\tau$   $\sigma$  i
     $\wedge$  map s_at sps = [li ..< (LTP_p  $\sigma$  i I) + 1]
     $\wedge$  ( $\forall$  sp  $\in$  set sps. s_check_exec vs  $\varphi$  sp))
  | (G I  $\varphi$ , SAlways i hi sps)  $\Rightarrow$ 
    (hi = (case right I of enat b  $\Rightarrow$  LTP_f  $\sigma$  i b)
     $\wedge$  right I  $\neq$   $\infty$ 
     $\wedge$  map s_at sps = [(ETP_f  $\sigma$  i I) ..< hi + 1]
     $\wedge$  ( $\forall$  sp  $\in$  set sps. s_check_exec vs  $\varphi$  sp))
  | ( $\varphi$  S I  $\psi$ , SSince sp2 sp1s)  $\Rightarrow$ 
    (let i = s_at (SSince sp2 sp1s); j = s_at sp2 in
    j  $\leq$  i  $\wedge$  mem ( $\tau$   $\sigma$  i -  $\tau$   $\sigma$  j) I
     $\wedge$  map s_at sp1s = [j+1 ..< i+1])

```

$$\begin{aligned}
& \wedge s_check_exec\ vs\ \psi\ sp2 \\
& \wedge (\forall sp1 \in set\ sp1s.\ s_check_exec\ vs\ \varphi\ sp1)) \\
| (\varphi\ \mathbf{U}\ I\ \psi,\ SUntil\ sp1s\ sp2) \Rightarrow \\
& (let\ i = s_at\ (SUntil\ sp1s\ sp2); j = s_at\ sp2\ in \\
& j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \\
& \wedge map\ s_at\ sp1s = [i ..< j] \wedge s_check_exec\ vs\ \psi\ sp2 \\
& \wedge (\forall sp1 \in set\ sp1s.\ s_check_exec\ vs\ \varphi\ sp1)) \\
| (\triangleleft I\ r,\ SMatchP\ rsp) \Rightarrow \\
& (let\ (j, i) = rs_at\ s_at\ rsp\ in\ j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge rs_check\ (s_check_exec\ vs)\ s_at\ r \\
rsp) \\
| (\triangleright I\ r,\ SMatchF\ rsp) \Rightarrow \\
& (let\ (i, j) = rs_at\ s_at\ rsp\ in\ i \leq j \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge rs_check\ (s_check_exec\ vs)\ s_at\ r \\
rsp) \\
| (_ , _) \Rightarrow False) \\
| v_check_exec\ vs\ f\ p = (case\ (f, p)\ of \\
& (\perp, VFF\ i) \Rightarrow True \\
& (r\ \dagger\ ts, VPred\ i\ pred\ ts') \Rightarrow \\
& (r = pred \wedge ts = ts' \wedge mk_values_subset_Compl\ r\ vs\ ts\ i) \\
& (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow \\
& (c = c' \wedge x = x' \wedge c \notin vs\ x) \\
& (\neg_F\ \varphi, VNeg\ sp) \Rightarrow s_check_exec\ vs\ \varphi\ sp \\
& (\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 \\
& (\varphi\ \wedge_F\ \psi, VAndL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \\
& (\varphi\ \wedge_F\ \psi, VAndR\ vp2) \Rightarrow v_check_exec\ vs\ \psi\ vp2 \\
& (\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\ \longleftarrow_F\ \psi, VIfFSV\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at \\
vp2) \\
& (\varphi\ \longleftarrow_F\ \psi, VIfFVS\ vp1\ sp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge v_at\ vp1 = s_at \\
sp2) \\
& (\exists_Fx.\ \varphi, VExists\ y\ vp_part) \Rightarrow (let\ i = v_at\ (part_hd\ vp_part) \\
& in\ x = y \wedge (\forall (sub, vp) \in SubsVals\ vp_part.\ v_at\ vp = i \wedge v_check_exec\ (vs\ (x := sub))\ \varphi\ vp)) \\
& (\forall_Fx.\ \varphi, VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v_check_exec\ (vs\ (x := \{val\}))\ \varphi\ vp) \\
& (\mathbf{Y}\ I\ \varphi, VPrev\ vp) \Rightarrow \\
& (let\ j = v_at\ vp; i = v_at\ (VPrev\ vp)\ in \\
& i = j+1 \wedge v_check_exec\ vs\ \varphi\ vp) \\
& (\mathbf{Y}\ I\ \varphi, VPrevZ) \Rightarrow True \\
& (\mathbf{Y}\ I\ \varphi, VPrevOutL\ i) \Rightarrow \\
& i > 0 \wedge \Delta\ \sigma\ i < left\ I \\
& (\mathbf{Y}\ I\ \varphi, VPrevOutR\ i) \Rightarrow \\
& i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I \\
& (\mathbf{X}\ I\ \varphi, VNext\ vp) \Rightarrow \\
& (let\ j = v_at\ vp; i = v_at\ (VNext\ vp)\ in \\
& j = i+1 \wedge v_check_exec\ vs\ \varphi\ vp) \\
& (\mathbf{X}\ I\ \varphi, VNextOutL\ i) \Rightarrow \\
& \Delta\ \sigma\ (i+1) < left\ I \\
& (\mathbf{X}\ I\ \varphi, VNextOutR\ i) \Rightarrow \\
& enat\ (\Delta\ \sigma\ (i+1)) > right\ I \\
& (\mathbf{P}\ I\ \varphi, VOnceOut\ i) \Rightarrow \\
& \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I \\
& (\mathbf{P}\ I\ \varphi, VOnce\ i\ li\ vps) \Rightarrow \\
& (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b) \\
& \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \\
& \wedge map\ v_at\ vps = [li ..< (LTP_p\ \sigma\ i\ I) + 1] \\
& \wedge (\forall vp \in set\ vps.\ v_check_exec\ vs\ \varphi\ vp)) \\
& (\mathbf{F}\ I\ \varphi, VEventually\ i\ hi\ vps) \Rightarrow \\
& (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty \\
& \wedge map\ v_at\ vps = [(ETP_f\ \sigma\ i\ I) ..< hi + 1]
\end{aligned}$$

```

   $\wedge (\forall vp \in \text{set } vps. v\_check\_exec \text{ vs } \varphi \text{ } vp))$ 
| (H  $I \varphi$ , VHistorically  $i \text{ } vp$ )  $\Rightarrow$ 
  (let  $j = v\_at \text{ } vp$  in
   $j \leq i \wedge \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge v\_check\_exec \text{ vs } \varphi \text{ } vp$ )
| (G  $I \varphi$ , VAlways  $i \text{ } vp$ )  $\Rightarrow$ 
  (let  $j = v\_at \text{ } vp$ 
  in  $j \geq i \wedge \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge v\_check\_exec \text{ vs } \varphi \text{ } vp$ )
| ( $\varphi$  S  $I \psi$ , VSinceOut  $i$ )  $\Rightarrow$ 
   $\tau \sigma i < \tau \sigma 0 + \text{left } I$ 
| ( $\varphi$  S  $I \psi$ , VSince  $i \text{ } vp1 \text{ } vp2s$ )  $\Rightarrow$ 
  (let  $j = v\_at \text{ } vp1$  in
  (case right  $I$  of  $\infty \Rightarrow \text{True} \mid \text{enat } b \Rightarrow \text{ETP\_p } \sigma \text{ } i \text{ } b \leq j$ )  $\wedge j \leq i$ 
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
 $\wedge \text{map } v\_at \text{ } vp2s = [j \text{ ..} < (\text{LTP\_p } \sigma \text{ } i \text{ } I) + 1] \wedge v\_check\_exec \text{ vs } \varphi \text{ } vp1$ 
 $\wedge (\forall vp2 \in \text{set } vp2s. v\_check\_exec \text{ vs } \psi \text{ } vp2)$ )
| ( $\varphi$  S  $I \psi$ , VSinceInf  $i \text{ } li \text{ } vp2s$ )  $\Rightarrow$ 
  ( $li = (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } b \Rightarrow \text{ETP\_p } \sigma \text{ } i \text{ } b)$ 
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
 $\wedge \text{map } v\_at \text{ } vp2s = [li \text{ ..} < (\text{LTP\_p } \sigma \text{ } i \text{ } I) + 1]$ 
 $\wedge (\forall vp2 \in \text{set } vp2s. v\_check\_exec \text{ vs } \psi \text{ } vp2)$ )
| ( $\varphi$  U  $I \psi$ , VUntil  $i \text{ } vp2s \text{ } vp1$ )  $\Rightarrow$ 
  (let  $j = v\_at \text{ } vp1$  in
  (case right  $I$  of  $\infty \Rightarrow \text{True} \mid \text{enat } b \Rightarrow j < \text{LTP\_f } \sigma \text{ } i \text{ } b$ )  $\wedge i \leq j$ 
 $\wedge \text{map } v\_at \text{ } vp2s = [\text{ETP\_f } \sigma \text{ } i \text{ } I \text{ ..} < j + 1] \wedge v\_check\_exec \text{ vs } \varphi \text{ } vp1$ 
 $\wedge (\forall vp2 \in \text{set } vp2s. v\_check\_exec \text{ vs } \psi \text{ } vp2)$ )
| ( $\varphi$  U  $I \psi$ , VUntilInf  $i \text{ } hi \text{ } vp2s$ )  $\Rightarrow$ 
  ( $hi = (\text{case right } I \text{ of } \text{enat } b \Rightarrow \text{LTP\_f } \sigma \text{ } i \text{ } b) \wedge \text{right } I \neq \infty$ 
 $\wedge \text{map } v\_at \text{ } vp2s = [\text{ETP\_f } \sigma \text{ } i \text{ } I \text{ ..} < hi + 1]$ 
 $\wedge (\forall vp2 \in \text{set } vp2s. v\_check\_exec \text{ vs } \psi \text{ } vp2)$ )
| ( $\triangleleft I \text{ } r$ , VMatchPOut  $i$ )  $\Rightarrow \tau \sigma i < \tau \sigma 0 + \text{left } I$ 
| ( $\triangleleft I \text{ } r$ , VMatchP  $i \text{ } rvps$ )  $\Rightarrow$ 
  (let  $j = \text{ETP } \sigma \text{ } (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \tau \sigma i - n)$ 
  in  $\tau \sigma i \geq \tau \sigma 0 + \text{left } I \wedge \text{map } (\text{fst} \circ \text{rv\_at } v\_at) \text{ } rvps = [j \text{ ..} < \text{Suc } (\text{LTP\_p } \sigma \text{ } i \text{ } I)] \wedge$ 
  ( $\forall rvp \in \text{set } rvps. \text{rv\_check } (v\_check\_exec \text{ vs}) \text{ } v\_at \text{ } r \text{ } rvp \wedge \text{snd } (\text{rv\_at } v\_at \text{ } rvp) = i$ ))
| ( $\triangleright I \text{ } r$ , VMatchF  $i \text{ } rvps$ )  $\Rightarrow$ 
  (let  $j = \text{LTP } \sigma \text{ } (\text{case right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } n \Rightarrow \tau \sigma i + n)$ 
  in  $\text{map } (\text{snd} \circ \text{rv\_at } v\_at) \text{ } rvps = [\text{ETP\_f } \sigma \text{ } i \text{ } I \text{ ..} < \text{Suc } j] \wedge \text{right } I \neq \infty \wedge$ 
  ( $\forall rvp \in \text{set } rvps. \text{rv\_check } (v\_check\_exec \text{ vs}) \text{ } v\_at \text{ } r \text{ } rvp \wedge \text{fst } (\text{rv\_at } v\_at \text{ } rvp) = i$ ))
| ( $\_ \text{ , } \_$ )  $\Rightarrow \text{False}$ 

```

```

declare  $s\_check\_exec.\text{simps}[simp \text{ del}] \text{ } v\_check\_exec.\text{simps}[simp \text{ del}]$ 
simps_of_case  $s\_check\_exec.\text{simps}[simp]: s\_check\_exec.\text{simps}[\text{unfolded prod.case}]$  (splits: formula.split
sproof.split)
simps_of_case  $v\_check\_exec.\text{simps}[simp]: v\_check\_exec.\text{simps}[\text{unfolded prod.case}]$  (splits: formula.split
vproof.split)

```

lemma *check_fv_cong*:

assumes $\forall x \in \text{fv } \varphi. v \text{ } x = v' \text{ } x$

shows $s_check \text{ } v \text{ } \varphi \text{ } sp \longleftrightarrow s_check \text{ } v' \text{ } \varphi \text{ } sp \wedge v_check \text{ } v \text{ } \varphi \text{ } vp \longleftrightarrow v_check \text{ } v' \text{ } \varphi \text{ } vp$

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  $\varphi_1$   $\varphi_2$ )
{
  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  $\varphi_1$   $\varphi_2$ )
{
  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  $\varphi_1$   $\varphi_2$ )
{
  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
}
next
case (MatchP I r)
{
  case 1
  with MatchP[of _ v v'] show ?case
  by (cases sp) (auto simp: collect_alt elim!: rs_check_cong[THEN iffD1, rotated -1])
next
  case 2
  with MatchP[of _ v v'] show ?case
  by (cases vp) (auto simp: collect_alt Let_def elim!: rv_check_cong[THEN iffD1, rotated -1])
}
next
case (MatchF I r)
{
  case 1
  with MatchF[of _ v v'] show ?case
  by (cases sp) (auto simp: collect_alt elim!: rs_check_cong[THEN iffD1, rotated -1])
next
  case 2
  with MatchF[of _ v v'] show ?case
  by (cases vp) (auto simp: collect_alt Let_def elim!: rv_check_cong[THEN iffD1, rotated -1])
}
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*: $(X, t) \in SubsVals\ part \implies X \neq \{\}$

by transfer (auto simp: partition_on_def image_iff)

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

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

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

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

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

lemma check_exec_check:
assumes $\forall x. vs\ x \neq \{\}$
shows $s_check_exec\ vs\ \varphi\ sp \longleftrightarrow (\forall v \in \text{compatible_vals } (fv\ \varphi)\ vs. s_check\ v\ \varphi\ sp)$
and $v_check_exec\ vs\ \varphi\ vp \longleftrightarrow (\forall v \in \text{compatible_vals } (fv\ \varphi)\ vs. v_check\ v\ \varphi\ vp)$
using assms

proof (induct φ arbitrary: vs sp vp)
case TT
 {
case 1
then show ?case **using** compatible_vals_nonemptyI
by (cases sp)
 auto
next
case 2
then show ?case **using** compatible_vals_nonemptyI
by auto
 }
next
case FF
 {
case 1
then show ?case **using** compatible_vals_nonemptyI
by (cases sp)
 auto
next
case 2
then show ?case **using** compatible_vals_nonemptyI
by (cases vp)
 auto
 }
next
case (Pred p ts)
 {

```

case 1
have obs:  $\forall tX \in \text{set } (vs \{ts\}). \text{snd } tX \neq \{\}$ 
  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_Cmpl_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  $\varphi2 \ \_ \ sp'$ ] 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  $\varphi_1 \cup \text{fv } \varphi_2$ ] 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  $\varphi_1 \cup \text{fv } \varphi_2$ ]]
  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 (And  $\varphi_1 \varphi_2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi_1 \cup \text{fv } \varphi_2$ ] show ?case
  proof (cases sp)
    case (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  $\varphi_1 \cup \text{fv } \varphi_2$ ]]
    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 (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 (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  $\varphi_1 \cup$  fv  $\varphi_2$ ] show ?case
  proof (cases sp)
    case (SImpL  $vp^{\wedge}$ )
    from check_fv_cong(2)[of  $\varphi_1$  _ _  $vp^{\wedge}$ ] show ?thesis
    unfolding SImpL s_check_exec_simps s_check_simps fv.simps Imp(2)[OF 1, of  $vp^{\wedge}$ ]
    by (metis (mono_tags, lifting) 1 IntE Un_upper1 compatible_vals_extensible compatible_vals_union_eq)
  next
    case (SImpR  $sp^{\wedge}$ )
    from check_fv_cong(1)[of  $\varphi_2$  _ _  $sp^{\wedge}$ ] show ?thesis
    unfolding SImpR s_check_exec_simps s_check_simps fv.simps Imp(3)[OF 1, of  $sp^{\wedge}$ ]
    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 (VImp  $sp_1$   $vp_2$ )
  from check_fv_cong(1)[of  $\varphi_1$  _ _  $sp_1$ ] check_fv_cong(2)[of  $\varphi_2$  _ _  $vp_2$ ] show ?thesis
  unfolding VImp v_check_exec_simps v_check_simps fv.simps ball_conj_distrib
    Imp(1)[OF 2, of  $sp_1$ ] Imp(4)[OF 2, of  $vp_2$ ]
    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 (Iff  $\varphi_1$   $\varphi_2$ )
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of fv  $\varphi_1 \cup$  fv  $\varphi_2$ ] show ?case
  proof (cases sp)
    case (SIffSS  $sp_1$   $sp_2$ )
    from check_fv_cong(1)[of  $\varphi_1$  _ _  $sp_1$ ] check_fv_cong(1)[of  $\varphi_2$  _ _  $sp_2$ ] show ?thesis
    unfolding SIffSS s_check_exec_simps s_check_simps fv.simps ball_conj_distrib
      Iff(1)[OF 1, of  $sp_1$ ] Iff(3)[OF 1, of  $sp_2$ ]
      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
  next
    case (SIffVV  $vp_1$   $vp_2$ )

```

```

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 = ( $\forall v \in$  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  $\neq \{\} \implies v\_check\_exec (vs(x := Z)) \varphi vp = (\forall v \in compatible\_vals (fv \varphi) (vs(x := Z))). v\_check$ 
    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  $\neq \{\} \implies s\_check\_exec (vs(x := Z)) \varphi sp = (\forall v \in compatible\_vals (fv \varphi) (vs(x := Z))). s\_check$  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}))  $\varphi vp = (\forall v \in compatible\_vals (fv \varphi) (vs(x := \{z\}))). v\_check$  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  $\varphi 1$  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 (SSince sp' sps)
  from check_fv_cong(1)[of  $\varphi 2$  _ _ 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  $\varphi 1 \cup$  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 (VSince i vp' vps)
  from check_fv_cong(2)[of  $\varphi 1$  _ _ 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  $\varphi 1 \cup$  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 ( $\varphi 2$  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 ( $\varphi 2$  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 (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  $\varphi_1 \cup$  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 (MatchP I r)
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of Regex.collect fv r] show ?case
  proof (cases sp)
    case (SMatchP rsp)
    show ?thesis
    unfolding SMatchP s_check_exec_simps s_check_simps fv_simps Let_def split_beta ball_conj_distrib
      ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of Regex.collect fv r]]
    unfolding collect_alt compatible_vals_Union_eq
    by (intro conj_cong refl
      rs_check_exec_rs_check compatible_vals_nonemptyI 1
      compatible_vals_union_eq compatible_vals_Union_eq
      compatible_vals_extensible check_fv_cong(1) MatchP(1)[OF _ 1])
  qed auto
  next
  case 2
  with compatible_vals_nonemptyI[OF 2, of Regex.collect fv r] show ?case
  proof (cases vp)
    case (VMatchP i rvps)
    show ?thesis
    unfolding VMatchP v_check_exec_simps v_check_simps fv_simps Let_def split_beta ball_conj_distrib
      ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of Regex.collect fv r]]
    unfolding collect_alt compatible_vals_Union_eq ball_swap[of _ set rvps]
    by (intro conj_cong refl ball_cong
      rv_check_exec_rv_check compatible_vals_nonemptyI 2
      compatible_vals_union_eq compatible_vals_Union_eq
      compatible_vals_extensible check_fv_cong(2) MatchP(2)[OF _ 2])
  qed auto
}
next
case (MatchF I r)
{
  case 1
  with compatible_vals_nonemptyI[OF 1, of Regex.collect fv r] show ?case
  proof (cases sp)
    case (SMatchF rsp)
    show ?thesis
    unfolding SMatchF s_check_exec_simps s_check_simps fv_simps Let_def split_beta ball_conj_distrib
      ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 1, of Regex.collect fv r]]

```

```

unfolding collect_alt compatible_vals_Union_eq
by (intro arg_cong2[of _ _ _ _] (∧)] refl
    rs_check_exec_rs_check compatible_vals_nonemptyI 1
    compatible_vals_Union_eq compatible_vals_Union_eq
    compatible_vals_extensible check_fv_cong(1) MatchF(1)[OF _ 1])
qed auto
next
case 2
with compatible_vals_nonemptyI[OF 2, of Regex.collect fv r] show ?case
proof (cases vp)
  case (VMatchF i rvps)
  show ?thesis
unfolding VMatchF v_check_exec_simps v_check_simps fv_simps Let_def split_beta ball_conj_distrib
  ball_triv_nonempty[OF compatible_vals_nonemptyI[OF 2, of Regex.collect fv r]]
unfolding collect_alt compatible_vals_Union_eq ball_swap[of _ set rvps]
by (intro conj_cong refl ball_cong
    rv_check_exec_rv_check compatible_vals_nonemptyI 2
    compatible_vals_Union_eq compatible_vals_Union_eq
    compatible_vals_extensible check_fv_cong(2) MatchF(2)[OF _ 2])
qed auto
}
qed

```

```

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

```

```

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

```

9.3 Latest Relevant Time-Point

```

fun rLRTP :: ('a ⇒ nat ⇒ nat option) ⇒ 'a Regex.regex ⇒ nat ⇒ nat option where
  rLRTP LRTP (Regex.Skip n) i = Some i
| rLRTP LRTP (Regex.Test x) i = LRTP x i
| rLRTP LRTP (Regex.Plus r s) i = max_opt (rLRTP LRTP r i) (rLRTP LRTP s i)
| rLRTP LRTP (Regex.Times r s) i = max_opt (rLRTP LRTP r i) (rLRTP LRTP s i)
| rLRTP LRTP (Regex.Star r) i = rLRTP LRTP r i

```

```

lemma rLRTP_cong[fundef_cong]:
  (∧x. x ∈ regex.atms r ⇒ LRTP x i = LRTP' x i) ⇒ rLRTP LRTP r i = rLRTP LRTP' r i
by (induct r) auto

```

```

lemma fb_rLRTP:
  assumes ∀φ ∈ regex.atms r. future_bounded φ ∧ ¬ Option.is_none (LRTP φ i)
  shows ¬ Option.is_none (rLRTP LRTP r i)
  using assms by (induct r) (auto simp: max_opt_def Option.is_none_def)

```

```

fun LRTP :: ('n, 'd) formula ⇒ nat ⇒ nat option where
  LRTP ⊤ i = Some i
| LRTP ⊥ i = Some i
| LRTP (⊥ † _) i = Some i
| LRTP (⊥ ≈ _) i = Some i
| LRTP (¬F φ) i = LRTP φ i
| LRTP (φ ∨F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (φ ∧F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)
| LRTP (φ →F ψ) i = max_opt (LRTP φ i) (LRTP ψ i)

```

```

| LRTP ( $\varphi \longleftrightarrow_F \psi$ )  $i = \max\_opt$  (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  $\sigma$   $i$   $b$ )))
| LRTP ( $\triangleleft$   $I$   $r$ )  $i =$ 
  (let  $X = (\lambda\varphi. LRTP$   $\varphi$   $i$ ) ‘regex.atms  $r$  in
  if  $X = \{\}$  then Some  $i$  else if  $None \in X$  then None else Some (Max (the ‘ $X$ ’)))
| LRTP ( $\triangleright$   $I$   $r$ )  $i = (case$  right  $I$  of  $\infty \Rightarrow None$  |  $enat$   $b \Rightarrow$ 
  let  $X = (\lambda\varphi. LRTP$   $\varphi$  (LTP_f  $\sigma$   $i$   $b$ )) ‘regex.atms  $r$  in
  if  $X = \{\}$  then Some (LTP_f  $\sigma$   $i$   $b$ ) else if  $None \in X$  then None else Some (Max (the ‘ $X$ ’)))

```

lemma *fb_LRTP*:

```

  assumes future_bounded  $\varphi$ 
  shows  $\neg Option.is\_none$  (LRTP  $\varphi$   $i$ )
  using assms
proof (induction  $\varphi$   $i$  rule: LRTP.induct)
  case (20  $I$   $r$   $i$ )
  from 20(2) show ?case
  by (auto 0 4 simp add: max_opt_def Option.is_none_def Let_def regex.pred_set dest: 20(1)[rotated])
next
  case (21  $I$   $r$   $i$ )
  from 21(2) show ?case
  by (auto 0 4 simp add: max_opt_def Option.is_none_def Let_def regex.pred_set dest: 21(1)[rotated])
qed (auto simp: max_opt_def Option.is_none_def)

```

lemma *not_none_fb_LRTP*:

```

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

```

lemma *is_some_fb_LRTP*:

```

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

```

lemma *enat_trans[simp]*: $enat$ $i \leq enat$ $j \wedge enat$ $j \leq enat$ $k \implies enat$ $i \leq enat$ k

by *auto*

9.4 Active Domain

definition *AD* :: ($'n, 'd$) *formula* $\Rightarrow nat \Rightarrow 'd$ *set*

where AD φ $i = consts$ $\varphi \cup (\bigcup k \leq the$ (LRTP φ i). \bigcup (set ‘*snd* ‘ Γ σ k ’))

lemma *val_in_AD_iff*:

```

 $x \in fv$   $\varphi \implies v$   $x \in AD$   $\varphi$   $i \iff v$   $x \in consts$   $\varphi \vee$ 
( $\exists r$   $ts$   $k. k \leq the$  (LRTP  $\varphi$   $i$ )  $\wedge (r, v[ts]) \in \Gamma$   $\sigma$   $k \wedge x \in \bigcup$  (set (map fv_trm  $ts$ )))
unfolding AD_def Un_iff UN_iff Bex_def atMost_iff set_map
  ex_comm[of  $P :: \_ \Rightarrow nat \Rightarrow \_$  for  $P$ ] ex_simps image_iff

```

proof (*safe intro!*: *arg_cong*[of $_ _ \lambda x. _ \vee x$] *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 (λd. if v x = d then (v x) else c d) ds]
(auto simp: eval_trms_def o_def map_idI))
next
case (*RL i r ts t*)
then show *?case*
by (*intro exI[of _ v[[ts]] conjI*
(auto simp: eval_trms_def image_iff in_fv_trm_conv intro!: bexI[of _ t]))
qed

lemma *val_notin_AD_iff*:
 $x \in fv \varphi \implies v x \notin AD \varphi i \iff v x \notin consts \varphi \wedge$
 $(\forall r ts k. k \leq the (LRTP \varphi i) \wedge x \in \bigcup (set (map fv_trm ts)) \longrightarrow (r, v[[ts]]) \notin \Gamma \sigma k)$
using *val_in_AD_iff by blast*

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

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

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

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

9.5 Congruence Modulo Active Domain

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

lemma *AD_simps_regex*[simp]:
future_bounded ($\triangleleft I r$) \implies *regex.atms* $r \neq \{\}$ \implies *AD* ($\triangleleft I r$) $i = (\bigcup \varphi \in \text{regex.atms } r. \text{AD } \varphi i)$
future_bounded ($\triangleright I r$) \implies *regex.atms* $r \neq \{\}$ \implies *AD* ($\triangleright I r$) $i = (\bigcup \varphi \in \text{regex.atms } r. \text{AD } \varphi (LTP_f$
 $\sigma i (the_enat (right I))))$
unfolding *AD_def*
by (*auto* 0 6 *simp*: *Let_def collect_alt regex.pred_set image_image Ball_def not_none_fb_LRTP*
dest!: *Max_ge_iff*[*THEN iffD1, rotated -1*] *sym*[*of None*]
dest: *spec*[*where* $P = \lambda x. x \leq \text{Max } _ \longrightarrow _ x$, *THEN mp, OF* $_ \text{Max_ge_iff}$ [*THEN iffD2*]] *split*:
if_splits)

lemma *LTP_p_mono*: $i \leq j \implies LTP_p_safe \sigma i I \leq LTP_p_safe \sigma j I$
unfolding *LTP_p_safe_def*
by (*smt* (*verit*, *ccfv_threshold*) τ_mono *bot_nat_0.extremum diff_le_mono order.trans i_LTP_tau*
le_cases3 min.bounded_iff)

lemma *LTP_tau_mono*:
assumes $\tau \sigma i \leq u$
shows $LTP \sigma (\tau \sigma i) \leq LTP \sigma u$
using *assms* **unfolding** *LTP_def*
proof (*intro Max_mono*)
show *finite* $\{i. \tau \sigma i \leq u\}$
unfolding *finite_nat_set_iff_bounded_le Ball_def mem_Collect_eq*
by (*meson* τ_mono *ex_le_tau nle_le order_trans*)
qed *auto*

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* $\varphi \implies i \leq j \implies the (LRTP \varphi i) \leq the (LRTP \varphi j)$
proof (*induct* φ *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* $_ \text{Since}$ (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)
next
  case (MatchP I r)
  { assume ne: regex.atms r  $\neq \{\}$  and fb:  $\bigwedge \varphi. \varphi \in \text{regex.atms } r \implies \text{future\_bounded } \varphi$ 
    then obtain  $\varphi$  where  $\varphi: \varphi \in \text{regex.atms } r$  the (LRTP  $\varphi j$ ) = (MAX  $\varphi \in \text{regex.atms } r$ . the (LRTP
 $\varphi j$ ))
    using obtains_MAX[of regex.atms r  $\lambda \varphi$ . the (LRTP  $\varphi j$ ) thesis] by auto
    assume  $\forall x \in \text{regex.atms } r. \exists a \in \text{regex.atms } r. \neg \text{the (LRTP } a i) \leq \text{the (LRTP } x j)$ 
    with  $\varphi(1)$  obtain  $\psi$  where  $\psi: \psi \in \text{regex.atms } r \neg \text{the (LRTP } \psi i) \leq \text{the (LRTP } \varphi j)$ 
      by blast
    moreover have the (LRTP  $\psi i$ )  $\leq$  the (LRTP  $\psi j$ )
      using MatchP(1)[OF  $\psi(1)$  fb[OF  $\psi(1)$ ] MatchP(3)] .
    moreover have the (LRTP  $\psi j$ )  $\leq$  the (LRTP  $\varphi j$ )
      unfolding  $\varphi(2)$  by (subst Max_ge_iff) (auto simp: ne  $\psi(1)$ )
    ultimately have False by auto
  }
  with MatchP(2-) show ?case
  by (force simp: Let_def regex.pred_set not_none_fb_LRTP Max_ge_iff dest!: sym[of None])
next
  case (MatchF I r)
  let ?j = LTP_f  $\sigma j$  (the_enat (right I))
  let ?i = LTP_f  $\sigma i$  (the_enat (right I))
  { assume ne: regex.atms r  $\neq \{\}$  and fb:  $\bigwedge \varphi. \varphi \in \text{regex.atms } r \implies \text{future\_bounded } \varphi$ 
    then obtain  $\varphi$  where  $\varphi: \varphi \in \text{regex.atms } r$  the (LRTP  $\varphi ?j$ ) = (MAX  $\varphi \in \text{regex.atms } r$ . the (LRTP
 $\varphi ?j$ ))
    using obtains_MAX[of regex.atms r  $\lambda \varphi$ . the (LRTP  $\varphi ?j$ ) thesis] by auto
    assume  $\forall x \in \text{regex.atms } r. \exists a \in \text{regex.atms } r. \neg \text{the (LRTP } a ?i) \leq \text{the (LRTP } x ?j)$ 
    with  $\varphi(1)$  obtain  $\psi$  where  $\psi: \psi \in \text{regex.atms } r \neg \text{the (LRTP } \psi ?i) \leq \text{the (LRTP } \varphi ?j)$ 
      by blast
    moreover have the (LRTP  $\psi ?i$ )  $\leq$  the (LRTP  $\psi ?j$ )
      using MatchF(1)[OF  $\psi(1)$  fb[OF  $\psi(1)$ ] LTP_f_mono[OF MatchF(3)]] .
    moreover have the (LRTP  $\psi ?j$ )  $\leq$  the (LRTP  $\varphi ?j$ )
      unfolding  $\varphi(2)$  by (subst Max_ge_iff) (auto simp: ne  $\psi(1)$ )
    ultimately have False by auto
  }
  with MatchF(2-) show ?case
  by (auto simp: Let_def regex.pred_set not_none_fb_LRTP Max_ge_iff LTP_f_mono dest!: sym[of
None] elim!: meta_mp)
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 φ
 and $(\forall x \in \text{fv } \varphi. v x = v' x \vee (v x \notin AD \varphi i \wedge v' x \notin AD \varphi i))$
 shows $(s_at \text{ sp} = i \implies s_check v \varphi \text{ sp} \longleftrightarrow s_check v' \varphi \text{ sp})$
 $(v_at \text{ vp} = i \implies v_check v \varphi \text{ vp} \longleftrightarrow v_check v' \varphi \text{ vp})$
 using assms

proof (induction $v \varphi \text{ sp}$ and $v \varphi \text{ vp}$ arbitrary: $i v'$ and $i v'$ rule: s_check_v_check.induct)

```

case (1 v f sp)
note IH = 1(1-25)[OF refl] and hyps = 1(26-28)
show ?case
proof (cases sp)
  case (SPred j r ts)
  then show ?thesis
  proof (cases f)
    case (Pred q us)
    with SPred hyps show ?thesis
    using eval_trms_fv_cong[of ts v v']
    by (force simp: val_notin_AD_iff dest!: spec[of _ i] spec[of _ r] spec[of _ ts])
  qed auto
next
  case (SEq_Const j r ts)
  with hyps show ?thesis
  by (cases f) (auto simp: val_notin_AD_iff)
next
  case (SNeg vp')
  then show ?thesis
  using IH(1)[of _ _ _ v'] hyps
  by (cases f) auto
next
  case (SOrL sp')
  then show ?thesis
  using IH(2)[of _ _ _ _ v'] hyps
  by (cases f) auto
next
  case (SOrR sp')
  then show ?thesis
  using IH(3)[of _ _ _ _ v'] hyps
  by (cases f) auto
next
  case (SAnd sp1 sp2)
  then show ?thesis
  using IH(4,5)[of _ _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
  case (SImpL vp')
  then show ?thesis
  using IH(6)[of _ _ _ _ v'] hyps
  by (cases f) auto
next
  case (SImpR sp')
  then show ?thesis
  using IH(7)[of _ _ _ _ _ v'] hyps
  by (cases f) auto
next
  case (SIffSS sp1 sp2)
  then show ?thesis
  using IH(8,9)[of _ _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
  case (SIffVV vp1 vp2)
  then show ?thesis
  using IH(10,11)[of _ _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
  case (SExists x z sp')

```

```

then show ?thesis
  using IH(12)[of x _ x z sp' i v'(x := z)] hyps
  by (cases f) (auto simp add: fun_upd_def)
next
case (SForall x part)
then show ?thesis
  using IH(13)[of x _ x part _ _ D _ z _ v'(x := z) for D z, OF _ _ _ _ refl _ refl] hyps
  by (cases f) (auto simp add: fun_upd_def)
next
case (SPrev sp')
then show ?thesis
  using IH(14)[of _ _ _ _ _ v'] hyps
  by (cases f) auto
next
case (SNext sp')
then show ?thesis
  using IH(15)[of _ _ _ _ _ v'] hyps
  by (cases f) (auto simp add: Let_def)
next
case (SONce j sp')
then show ?thesis
proof (cases f)
  case (Once I  $\varphi$ )
  { 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  $l = s\_at \text{ sp and } u = LTP\_p \sigma i I$ 
    assume *:  $sp \in set \text{ sps } \tau \sigma 0 + \text{left } I \leq \tau \sigma i$ 
    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  $j: sp = \text{sps ! } j \ j < \text{length sps}$ 
      unfolding in_set_conv_nth by auto
    moreover
    assume eq:  $map \text{ s\_at sps} = [k ..< Suc u]$ 
    then have len:  $\text{length sps} = Suc u - k$ 
      by (auto dest!: arg_cong[where f=length])
    moreover
    have  $s\_at (\text{sps ! } j) = k + j$ 
      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 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 Historically SHistorically show ?thesis
    using IH(17)[OF Historically SHistorically _ refl, of _ v'] hypos(1,2)
    by auto
  qed auto
next
case (SEventually j sp')
then show ?thesis
proof (cases f)
  case (Eventually I φ)
  { 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 hypos(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'] hypos(1,2)
    by (auto simp: Let_def)
  qed auto
next
case (SAlways j k sps)
then show ?thesis
proof (cases f)
  case (Always I φ)
  { fix sp :: ('n, 'd) sproof
    define l and u where  $l = s\_at sp$  and  $u = LTP\_f \sigma i (the\_enat (right I))$ 
    assume  $*$ :  $sp \in set\ sps$ 
    then obtain j where  $j: sp = sps ! j \wedge j < length\ sps$ 
      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 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 dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
    }
  with Always SAlways show ?thesis
    using IH(19)[OF Always SAlways _ refl, of _ v'] hypos(1,2)
    by auto
  qed auto
next
case (SSince sp' sps)
then show ?thesis
proof (cases f)
  case (Since φ I ψ)
  { fix sp :: ('n, 'd) sproof
    define l where  $l = s\_at sp$ 
    assume  $*$ :  $sp \in set\ sps$ 
    from  $*$ (1) obtain j where  $j: sp = sps ! j \wedge j < length\ sps$ 
      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  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi l \wedge v' x \notin AD \varphi l$ 
    by (auto simp: dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
  }
}
moreover
{ fix k
  assume  $\tau \sigma$  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  $\forall x \in fv \psi. v x = v' x \vee v x \notin AD \psi k \wedge v' x \notin AD \psi k$ 
  by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
}
ultimately show ?thesis
  using 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
next
case (SMatchP rsp)
then show ?thesis
proof (cases  $\forall sp' \in spatms \text{rsp}. s\_at \text{sp}' \leq s\_at \text{sp}$ )
  case True
  with SMatchP show ?thesis
  proof (cases f)
    case (MatchP I r)
    show ?thesis unfolding SMatchP MatchP s_check_simps Let_def split_beta
    proof ((rule conj_cong refl rs_check_cong IH(24) prod.collapse refl SMatchP MatchP | assumption)
    +, goal_cases fb AD _)
      case (fb x sp)
      with MatchP hyps show ?case by (auto simp: regex.pred_set collect_alt)
    next
      case (AD x sp)
      with hyps True show ?case unfolding MatchP
        by (subst (asm) (1 2) AD_simps_regex)
        (auto simp: regex.pred_set collect_alt dest!: bspec dest: AD_mono[THEN set_mp, rotated
-1])
      qed simp
    qed simp_all
  next
  case False
  with SMatchP show ?thesis
    by (cases f) (auto dest: rs_check_le2[rotated 2])
  qed
next
case (SMatchF rsp)
then show ?thesis
proof (cases f)
  case (MatchF I r)
  show ?thesis
  proof (cases  $\forall sp' \in spatms \text{rsp}. s\_at \text{sp}' \leq LTP\_f \sigma (s\_at \text{sp}) (the\_enat (right I))$ )
    case True
    show ?thesis unfolding SMatchF MatchF s_check_simps Let_def split_beta
    proof ((rule conj_cong refl rs_check_cong IH(25) prod.collapse refl SMatchF MatchF | assumption)
    +, goal_cases fb AD _)
      case (fb x sp)
      with MatchF hyps show ?case by (auto simp: regex.pred_set collect_alt)
    next
      case (AD x sp)
      with hyps True show ?case unfolding MatchF
        by (subst (asm) (1 2) AD_simps_regex)
        (auto simp: regex.pred_set collect_alt dest!: bspec
        dest: AD_mono[THEN set_mp, rotated -1] order_trans[OF _ i_le_LTPi_add])
      qed simp
    next
    case False
    then obtain sp' where sp' : sp' ∈ spatms rsp ∧ s_at sp' ≤ LTP_f σ (s_at sp) (the_enat (right
I))

```

```

    by auto
  show ?thesis unfolding SMatchF MatchF s_check_simps Let_def split_beta
  proof (intro conj_cong refl iffI, goal_cases LR RL)
    case LR
    have  $\forall sp \in spatms\ rsp. s\_at\ sp \leq snd\ (rs\_at\ s\_at\ rsp)$ 
      using rs_check_le2[OF __ LR(3)] by auto
    with LR(2) sp' hyps(2) show ?case
      using i_le_LTPi_add[of snd (rs_at s_at rsp)  $\sigma\ 0$ ]
      unfolding SMatchF MatchF s_at_simps future_bounded_simps
      by (elim notE order_trans) (auto intro!: LTP_ $\tau$ _mono elim!: order_trans)
    next
    case RL
    have  $\forall sp \in spatms\ rsp. s\_at\ sp \leq snd\ (rs\_at\ s\_at\ rsp)$ 
      using rs_check_le2[OF __ RL(3)] by auto
    with RL(2) sp' hyps(2) show ?case
      using i_le_LTPi_add[of snd (rs_at s_at rsp)  $\sigma\ 0$ ]
      unfolding SMatchF MatchF s_at_simps future_bounded_simps
      by (elim notE order_trans) (auto intro!: LTP_ $\tau$ _mono elim!: order_trans)
  qed
  qed
  qed simp_all
  qed (cases f; simp_all)+
next
case (2 v f vp)
note IH = 2(1-27)[OF refl] and hyps = 2(28-30)
show ?case
proof (cases vp)
  case (VPred j r ts)
  then show ?thesis
  proof (cases f)
    case (Pred q us)
    with VPred hyps show ?thesis
      using eval_trms_fv_cong[of ts v v']
      by (force simp: val_notin_AD_iff dest!: spec[of _ i] spec[of _ r] spec[of _ ts])
  qed auto
next
case (VEq_Const j r ts)
with hyps show ?thesis
  by (cases f) (auto simp: val_notin_AD_iff)
next
case (VNeg sp^)
then show ?thesis
  using IH(1)[of _ _ _ v'] hyps
  by (cases f) auto
next
case (VOr vp1 vp2)
then show ?thesis
  using IH(2,3)[of _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
case (VAndL vp^)
then show ?thesis
  using IH(4)[of _ _ _ _ v'] hyps
  by (cases f) auto
next
case (VAndR vp^)
then show ?thesis
  using IH(5)[of _ _ _ _ v'] hyps

```

```

    by (cases f) auto
next
case (VImp sp1 vp2)
then show ?thesis
  using IH(6,7)[of _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
case (VIffSV sp1 vp2)
then show ?thesis
  using IH(8,9)[of _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
case (VIffVS vp1 sp2)
then show ?thesis
  using IH(10,11)[of _ _ _ _ v'] hyps
  by (cases f) (auto 7 0)+
next
case (VExists x part)
then show ?thesis
  using IH(12)[of x _ x part _ _ D _ z _ v'(x := z) for D z, OF _ _ _ _ refl _ refl] hyps
  by (cases f) (auto simp add: fun_upd_def)
next
case (VForall x z vp')
then show ?thesis
  using IH(13)[of x _ x z vp' i v'(x := z)] hyps
  by (cases f) (auto simp add: fun_upd_def)
next
case (VPrev vp')
then show ?thesis
  using IH(14)[of _ _ _ _ _ v'] hyps
  by (cases f) auto
next
case (VNext vp')
then show ?thesis
  using IH(15)[of _ _ _ _ _ v'] hyps
  by (cases f) auto
next
case (VOnce j k vps)
then show ?thesis
proof (cases f)
  case (Once I φ)
  { fix vp :: ('n, 'd) vproof
    define l and u where l = v_at vp and u = LTP_p σ i I
    assume *: vp ∈ set vps τ σ 0 + left I ≤ τ σ i
    then have u_def: u = LTP_p_safe σ i I
      by (auto simp: LTP_p_safe_def u_def)
    from *(1) obtain j where j: vp = vps ! j j < length vps
      unfolding in_set_conv_nth by auto
    moreover
    assume eq: map v_at vps = [k ..< Suc u]
    then have len: length vps = Suc u - k
      by (auto dest!: arg_cong[where f=length])
    moreover
    have v_at (vps ! j) = k + j
      using arg_cong[where f=λxs. nth xs j, OF eq] j len *(2)
      by (auto simp: nth_append)
    ultimately have l ≤ u
      unfolding l_def by auto
  }

```

```

    with Once hyps(2,3) have  $\forall x \in fv \varphi. v x = v' x \vee v x \notin AD \varphi l \wedge v' x \notin AD \varphi l$ 
      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  $\tau \sigma k \leq the\_enat (right I) + \tau \sigma i$ 
    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 Always 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 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 sp and u = LTP_p  $\sigma$  i I
    assume *: sp  $\in$  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: sp = vps ! j j < length vps
      unfolding in_set_conv_nth by auto
    moreover
    assume eq: map v_at vps = [v_at vp' ..< Suc u]
    then have len: length vps = Suc u - v_at vp'
      by (auto dest!: arg_cong[where f=length])
    moreover
    have v_at (vps ! j) = v_at vp' + j
      using arg_cong[where f= $\lambda$ xs. 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  $\forall x \in fv \psi. v x = v' x \vee v x \notin AD \psi l \wedge v' x \notin AD \psi l$ 
      by (auto simp: u_def dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
  }
  moreover
  { fix k
    assume k: k  $\leq$  i
    with Since 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 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  $*$ :  $sp \in set vps \wedge v\_at vp' \leq LTP\_f \sigma i (the\_enat (right I))$ 
    from  $*(1)$  obtain  $j$  where  $j$ :  $sp = vps ! j \wedge j < length vps$ 
    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  $j$ :  $vp = vps ! j \wedge j < length vps$ 
    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 ≤ LTP_f σ i (the_enat (right I))
    by (auto simp: l_def u_def dest!: arg_cong[where f=λxs. nth xs j]
        simp del: upt.simps split: if_splits)
  with Until hyps(2,3) have ∀ x∈fv ψ. v x = v' x ∨ v x ∉ AD ψ l ∧ v' x ∉ AD ψ l
    by (auto dest!: bspec dest: AD_mono[THEN set_mp, rotated -1])
}
with Until VUntilInf show ?thesis
  using IH(25)[OF Until VUntilInf _ refl, of _ v'] hyps(1,2)
  by auto
qed auto
next
case (VMatchP i rvps)
then show ?thesis
proof (cases ∀ rvp ∈ set rvps. ∀ vp' ∈ vpatms rvp. v_at vp' ≤ v_at vp)
  case True
  with VMatchP show ?thesis
  proof (cases f)
    case (MatchP I r)
    show ?thesis unfolding VMatchP MatchP v_check_simps Let_def split_beta
    proof ((rule conj_cong ball_cong refl rv_check_cong IH(26) prod.collapse refl VMatchP MatchP
| assumption)+, goal_cases fb AD _ _)
      case (fb x sp)
      with MatchP hyps show ?case by (auto simp: regex.pred_set collect_alt)
    next
      case (AD x sp)
      with hyps True show ?case unfolding MatchP
      by (subst (asm) (1 2) AD_simps_regex)
      (auto simp: regex.pred_set collect_alt dest!: bspec dest: AD_mono[THEN set_mp, rotated
-1])
    qed simp_all
  qed simp_all
next
case False
with VMatchP show ?thesis
  by (cases f) (auto simp: Let_def dest: rv_check_le2)
qed
next
case (VMatchF ii rvps)
then show ?thesis
proof (cases f)
  case (MatchF I r)
  show ?thesis
  proof (cases ∀ rvp ∈ set rvps. ∀ vp' ∈ vpatms rvp. v_at vp' ≤ LTP_f σ (v_at vp) (the_enat (right
I)))
    case True
    show ?thesis unfolding VMatchF MatchF v_check_simps Let_def split_beta
    proof ((rule conj_cong ball_cong refl rv_check_cong IH(27) prod.collapse refl VMatchF MatchF
| assumption)+, goal_cases fb AD _ _)
      case (fb x sp)
      with MatchF hyps show ?case by (auto simp: regex.pred_set collect_alt)
    next
      case (AD x sp)
      with hyps True show ?case unfolding MatchF
      by (subst (asm) (1 2) AD_simps_regex)
      (auto simp: regex.pred_set collect_alt dest!: bspec
      dest: AD_mono[THEN set_mp, rotated -1] order_trans[OF _ i_le_LTPi_add])
    qed simp_all
  end
end

```

```

next
  case False
  then obtain k rvp vp' where vp': rvp = rvps ! k k < length rvps vp' ∈ vpatms rvp ¬ v_at vp' ≤
LTP_f σ (v_at vp) (the_enat (right I))
  by (auto simp: in_set_conv_nth)
  moreover from vp' hyps(1) have v_check v f vp ⇒ v_at vp' ≤ LTP_f σ (v_at vp) (the_enat
(right I)) for v
  unfolding VMatchF MatchF v_at.simps v_check_simps
  using [[linarith_split_limit=15]]
  by (auto simp: Let_def in_set_conv_nth nth_append nth_Cons'
      dest!: rv_check_le2[of _ _ _ vp'] bspec[of _ _ rvp]
      dest: arg_cong[where f=λxs. (length xs, nth xs k)] split: if_splits)
  ultimately show ?thesis by auto
qed
qed simp_all
qed (cases f; simp_all)+
qed

```

9.6 Checker Completeness

lemma part_hd_tabulate: $\text{distinct } xs \Rightarrow \text{part_hd } (\text{tabulate } xs \ f \ z) = (\text{case } xs \ \text{of } [] \Rightarrow z \mid (x \ \# \ _) \Rightarrow (\text{if } \text{set } xs = \text{UNIV} \ \text{then } f \ x \ \text{else } z))$
by (transfer, auto split: list.splits)

lemma s_at_tabulate:

```

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

```

lemma v_at_tabulate:

```

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

```

lemma s_check_tabulate:

```

assumes future_bounded φ
and ∀ z. s_at (mypick z) = i
and ∀ z. s_check (v(x:=z)) φ (mypick z)
and mypart = tabulate (sorted_list_of_set (AD φ i)) mypick (mypick (SOME z. z ∉ AD φ i))
shows ∀ (sub, vp) ∈ SubsVals mypart. ∀ z ∈ sub. s_check (v(x := z)) φ vp
using assms

```

proof (transfer fixing: σ φ mypick i v x, goal_cases 1)

```

case (1 mypart)
{ fix z
  assume s_at_assm: ∀ z. s_at (mypick z) = i
  and s_check_assm: ∀ z. s_check (v(x := z)) φ (mypick z)
  and fb_assm: future_bounded φ
  and z_notin_AD: z ∉ (AD φ i)
  have s_at_mypick: s_at (mypick (SOME z. z ∉ AD φ i)) = i
  using s_at_assm by simp
  have s_check_mypick: Checker.s_check σ (v(x := SOME z. z ∉ AD φ i)) φ (mypick (SOME z. z ∉
AD φ i))
  using s_check_assm by simp
  have s_check (v(x := z)) φ (mypick (SOME z. z ∉ AD φ i))
  using z_notin_AD
  by (subst check_AD_cong(1)[of φ v(x := z) v(x := (SOME z. z ∉ Checker.AD σ φ i))] i mypick

```

```

(SOME z. z ∉ AD ∅ i), OF fb_assm _ s_atmypick])
  (auto simp add: someI[of λz. z ∉ AD ∅ i z] s_checkmypick fb_assm split: if_splits)
}
with 1 show ?case
  by auto
qed

```

lemma *v_check_tabulate*:

```

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

```

lemma *s_at_part_hd_tabulate*:

```

assumes future_bounded ∅
  and ∀z. s_at (f z) = i
  and mypart = tabulate (sorted_list_of_set (AD ∅ i)) f (f (SOME z. z ∉ AD ∅ 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 ∅
  and ∀z. v_at (f z) = i
  and mypart = tabulate (sorted_list_of_set (AD ∅ i)) f (f (SOME z. z ∉ AD ∅ i))
shows v_at (part_hd mypart) = i
using assms by (simp add: part_hd_tabulate split: list_splits)

```

lemma *check_completeness_aux*:

```

(SAT ∅ v i ∅ → future_bounded ∅ → (∃ sp. s_at sp = i ∧ s_check v ∅ sp)) ∧
(VIO ∅ v i ∅ → future_bounded ∅ → (∃ vp. v_at vp = i ∧ v_check v ∅ vp))
proof (induct v i ∅ 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 ( $\exists_{Fx}.$   $\varphi$ )
  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  $v\_at (VEventually i j vps) = i \wedge v\_check v (F I \varphi) (VEventually i j vps)$ 
    using VEventually b_def j_def by simp
  then show  $\exists vp. v\_at vp = i \wedge v\_check v (F I \varphi) vp$ 
    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) =  $i \wedge s\_check v (H I \varphi)$  (SHistorically i j sps)
      using SHistorically by auto
    then show  $\exists sp. s\_at sp = i \wedge s\_check v (H I \varphi) sp$ 
      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  $\wedge$  v_check v  $\varphi$  vp1
    using fb_since VSince by auto
  obtain mypick where mypick_def:  $\forall k \in \{j \ .. \ LTP\_p \ \sigma \ i \ I\}$ . v_at (mypick k) = k  $\wedge$  v_check v  $\psi$ 
    (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  $\sigma$  i I)]  $\wedge$  ( $\forall$  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 ([j ..< Suc
    (LTP_p  $\sigma$  i I])])])
  then have v_at (VSince i vp1 vp2s) = i  $\wedge$  v_check v ( $\varphi$  S I  $\psi$ ) (VSince i vp1 vp2s)
    using vp1_def VSince by auto

```

```

    then show  $\exists vp. v\_at\ vp = i \wedge v\_check\ v\ (\varphi\ \mathbf{S}\ I\ \psi)\ vp$ 
      by blast
  qed
next
case (VSinceInf j I i v  $\psi\ \varphi$ )
show ?case
proof
  assume fb_since: future_bounded ( $\varphi\ \mathbf{S}\ I\ \psi$ )
  then have fb: future_bounded  $\varphi$  future_bounded  $\psi$ 
    by simp_all
  obtain mypick where mypick_def:  $\forall k \in \{j \dots LTP\_p\ \sigma\ i\ I\}. v\_at\ (mypick\ k) = k \wedge v\_check\ v\ \psi$ 
    (mypick k)
    using VSinceInf fb_since by atomize_elim (rule bchoice, simp)
  then obtain vp2s where vp2s_def:  $map\ (v\_at)\ vp2s = [j \dots Suc\ (LTP\_p\ \sigma\ i\ I)] \wedge (\forall vp \in set\ vp2s. v\_check\ v\ \psi\ vp)$ 
    by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([j ..< Suc (LTP_p  $\sigma$  i I)])])
  then have v_at (VSinceInf i j vp2s) =  $i \wedge v\_check\ v\ (\varphi\ \mathbf{S}\ I\ \psi)$  (VSinceInf i j vp2s)
    using VSinceInf by auto
  then show  $\exists vp. v\_at\ vp = i \wedge v\_check\ v\ (\varphi\ \mathbf{S}\ I\ \psi)\ vp$ 
    by blast
  qed
next
case (SUntil j 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 sp2 where sp2_def:  $s\_at\ sp2 = j \wedge s\_check\ v\ \psi\ sp2$ 
    using fb SUntil by blast
  {
    assume  $i \geq j$ 
    then have s_at (SUntil [] sp2) =  $i \wedge s\_check\ v\ (\varphi\ \mathbf{U}\ I\ \psi)$  (SUntil [] sp2)
      using sp2_def SUntil by auto
    then have  $\exists sp. s\_at\ sp = i \wedge s\_check\ v\ (\varphi\ \mathbf{U}\ I\ \psi)\ sp$ 
      by blast
  }
  moreover
  {
    assume  $i < j$ 
    obtain mypick where mypick_def:  $\forall k \in \{i \dots j\}. s\_at\ (mypick\ k) = k \wedge s\_check\ v\ \varphi\ (mypick\ k)$ 
      using SUntil fb_until by atomize_elim (rule bchoice, simp)
    then obtain sp1s where sp1s_def:  $map\ (s\_at)\ sp1s = [i \dots j] \wedge (\forall sp \in set\ sp1s. s\_check\ v\ \varphi\ sp)$ 
      by atomize_elim (auto intro!: trans[OF list.map_cong list.map_id] exI[of _ map mypick ([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  $i < j$ 
      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
next
case (SMatchP j i I v r)
then show ?case
  by (safe dest!: rs_check_complete[rotated, where test=s_check v and testi=s_at])
  (force simp: regex.pred_set intro: exI[of _ SMatchP _])
next
case (VMatchPOut i I v r)
then show ?case
  by (auto intro: exI[of _ VMatchPOut i])
next
case (VMatchP k I i v r)
  { fix j
    assume fb: regex.pred_regex future_bounded r and j: j  $\in$  {k..LTP_p  $\sigma i I$ }
    then have j  $\leq$  i
      by auto
    with j have  $\exists p$ . rv_check (v_check v) v_at r p  $\wedge$  rv_at v_at p = (j, i)
      by (rule rv_check_complete[rotated, where test=v_check v and testi=v_at, OF VMatchP(3)])
  }

```

```

      (use fb in ⟨auto simp: regex.pred_set⟩)
    } note * = this
  { assume fb: regex.pred_regex future_bounded r
    from *[OF this] obtain f where rv_check (v_check v) v_at r (f j) rv_at v_at (f j) = (j, i)
      if j ∈ {k..LTP_p σ i I} for j by metis
    with VMatchP(1,2) fb have ∃ vp. v_at vp = i ∧ v_check v (◁ I r) vp
      by (intro exI[of_ VMatchP i (map f [k ..< Suc (LTP_p σ i I])])
        (auto simp: Let_def o_def intro: map_idI split: enat.splits)
    }
  } then show ?case
    by simp
next
case (SMatchF i j I v r)
then show ?case
  by (safe dest!: rs_check_complete[rotated, where test=s_check v and testi=s_at])
    (force simp: regex.pred_set intro: exI[of_ SMatchF _])+
next
case (VMatchF I i v r)
let ?J = case right I of enat b ⇒ {ETP_f σ i I..LTP_f σ i b} | ∞ ⇒ {ETP_f σ i I..}
{ fix j
  assume fb: regex.pred_regex future_bounded r and j: j ∈ ?J
  then have i ≤ j
    by (auto split: enat.splits)
  with j have ∃ p. rv_check (v_check v) v_at r p ∧ rv_at v_at p = (i, j)
    by (rule rv_check_complete[rotated, where test=v_check v and testi=v_at, OF VMatchF(1)])
      (use fb in ⟨auto simp: regex.pred_set⟩)
  } note * = this
  { assume fb: regex.pred_regex future_bounded r right I ≠ ∞
    from *[OF this(1)] obtain f where rv_check (v_check v) v_at r (f j) rv_at v_at (f j) = (i, j)
      if j ∈ ?J for j by metis
    with fb have ∃ vp. v_at vp = i ∧ v_check v (▷ I r) vp
      by (intro exI[of_ VMatchF i (map f [ETP_f σ i I ..< Suc (LTP_f σ i (the_enat (right I))])])
        (auto simp: Let_def o_def intro: map_idI split: enat.splits)
    }
  } then show ?case
    by simp
qed

```

```

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

```

definition $p_check\ v\ \varphi\ p = (case\ p\ of\ Inl\ sp\ \Rightarrow\ s_check\ v\ \varphi\ sp\ | \ Inr\ vp\ \Rightarrow\ v_check\ v\ \varphi\ vp)$

definition $p_check_exec\ vs\ \varphi\ p = (case\ p\ of\ Inl\ sp\ \Rightarrow\ s_check_exec\ vs\ \varphi\ sp\ | \ Inr\ vp\ \Rightarrow\ v_check_exec\ vs\ \varphi\ vp)$

definition $valid\ ::\ ('n,\ 'd)\ envset\ \Rightarrow\ nat\ \Rightarrow\ ('n,\ 'd)\ formula\ \Rightarrow\ ('n,\ 'd)\ proof\ \Rightarrow\ bool\ \mathbf{where}$

```

  valid vs i φ p =
    (case p of
      Inl p ⇒ s_check_exec vs φ p ∧ s_at p = i
    | Inr p ⇒ v_check_exec vs φ p ∧ v_at p = i)

```

end

9.7 Lifting the Checker to PDTs

fun $check_one\ \mathbf{where}$

```

  check_one σ v φ (Leaf p) = p_check σ v φ p

```

| *check_one* σ v φ (Node x part) = *check_one* σ v φ (*lookup_part* part (v x))

fun *check_all_aux* **where**

check_all_aux σ vs φ (Leaf p) = *p_check_exec* σ vs φ p
| *check_all_aux* σ vs φ (Node x part) = $(\forall (D, e) \in \text{set } (\text{subvals part}). \text{check_all_aux } \sigma (vs(x := D)) \varphi e)$

fun *collect_paths_aux* **where**

collect_paths_aux DS σ vs φ (Leaf p) = (if *p_check_exec* σ vs φ p then {} else rev ' DS)
| *collect_paths_aux* DS σ vs φ (Node x part) = $(\bigcup (D, e) \in \text{set } (\text{subvals part}). \text{collect_paths_aux } (\text{Cons } D ' DS) \sigma (vs(x := D)) \varphi e)$

lemma *check_one_cong*: $\forall x \in \text{fv } \varphi \cup \text{vars } e. v x = v' x \implies \text{check_one } \sigma v \varphi e = \text{check_one } \sigma v' \varphi e$

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*: $\forall x. vs x \neq \{\} \implies \text{distinct_paths } e \implies (\forall x \in \text{vars } e. vs x = \text{UNIV}) \implies$

check_all_aux σ vs φ $e \longleftrightarrow (\forall v \in \text{compatible_vals } (\text{fv } \varphi) \text{ vs}. \text{check_one } \sigma v \varphi e)$

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_subvals*[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** $d \in D$
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 $\text{fv } \varphi$ 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  $\Rightarrow$  ('n, 'd) formula  $\Rightarrow$  ('n, 'd) expl  $\Rightarrow$  bool
where
  check_all  $\sigma \varphi e =$  (distinct_paths e  $\wedge$  check_all_aux  $\sigma (\lambda\_ . UNIV) \varphi e$ )

lemma check_one_alt: check_one  $\sigma v \varphi e =$  p_check  $\sigma v \varphi$  (eval_pdt v e)
by (induct e) auto

lemma check_all_alt: check_all  $\sigma \varphi e =$  (distinct_paths e  $\wedge$  ( $\forall v. p\_check \sigma v \varphi$  (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) = ( $\forall pdt \in$  Vals part. pdt_at i pdt)

lemma pdt_at_p_at_eval_pdt: pdt_at i e  $\Longrightarrow$  p_at (eval_pdt v e) = i
by (induct e) auto

lemma check_all_completeness_aux:
  fixes  $\varphi ::$  ('n, 'd :: {default, linorder}) formula
  shows set vs  $\subseteq$  fv  $\varphi \Longrightarrow$  future_bounded  $\varphi \Longrightarrow$  distinct vs  $\Longrightarrow$ 
   $\exists e. pdt\_at\ i\ e \wedge vars\_order\ vs\ e \wedge (\forall v. (\forall x. x \notin set\ vs \longrightarrow v\ x = w\ x) \longrightarrow p\_check\ \sigma\ v\ \varphi\ (eval\_pdt\ v\ e))$ 
proof (induct vs arbitrary: w)
  case Nil
  then show ?case
  proof (cases sat  $\sigma w i \varphi$ )
  case True
    then have SAT  $\sigma w i \varphi$  by (rule completeness)
    with Nil obtain sp where s_at sp = i s_check  $\sigma w \varphi$  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  $\sigma w i \varphi$  by (rule completeness)
    with Nil obtain vp where v_at vp = i v_check  $\sigma w \varphi$  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  $\Rightarrow$  'd)  $\Rightarrow$  ('n  $\Rightarrow$  'd)  $\Rightarrow$  bool where eq = rel_fun (eq_onp ( $\lambda x. x \notin set\ vs$ )) (=)
  from Cons have  $\forall w. \exists e. pdt\_at\ i\ e \wedge vars\_order\ vs\ e \wedge$ 
    ( $\forall v. (\forall x. x \notin set\ vs \longrightarrow v\ x = w\ x) \longrightarrow p\_check\ \sigma\ v\ \varphi\ (eval\_pdt\ v\ e)$ ) by simp
  then obtain pick :: 'd  $\Rightarrow$  ('n, 'd) expl where pick: pdt_at i (pick a) vars_order vs (pick a) and
    eq_pick:  $\bigwedge v. eq\ v\ (w(x := a)) \Longrightarrow p\_check\ \sigma\ v\ \varphi\ (eval\_pdt\ v\ (pick\ a))$  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 _  $\lambda a. f\ (w(x := a))$ ])
  qed
let ?a = SOME z. z  $\notin$  AD  $\sigma \varphi i$ 
let ?AD = sorted_list_of_set (AD  $\sigma \varphi i$ )

```

```

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  $w' x \notin AD \sigma \varphi i \implies ?a \notin AD \sigma \varphi i$ 
  by (metis some_eq_imp)
moreover have  $eq (w'(x := ?a)) (w(x := ?a))$ 
  using p_check by (auto simp: eq_def rel_fun_def eq_onp_def)
moreover have  $eq w' (w(x := w' x))$ 
  using p_check by (auto simp: eq_def rel_fun_def eq_onp_def)
ultimately show ?case
  using pick Cons(2-) eq_pick[of w' w' x] eq_pick[of w'(x := ?a) ?a]
    pdt_at_p_at_eval_pdt[of i pick ?a w'] eval_pdt_fun_upd[of vs pick ?a x w' ?a]
  by (auto simp: p_check_def p_at_def
    elim!: check_AD_cong[THEN iffD1, rotated -1, of _ _ _ _ i]
    split: if_splits sum.splits sum.splits)
qed (use Cons(2-) pick in <simp_all add: vars_order.intros>)
qed

lemma check_all_completeness:
  fixes  $\varphi :: ('n, 'd :: \{\text{default}, \text{linorder}\}) \text{ formula}$ 
  assumes future_bounded  $\varphi$ 
  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  $\varphi$ 
  by (meson finite_distinct_list finite_fv)
have s: s_check  $\sigma v \varphi sp$ 
  if vars_order vs e
  and  $\forall v. (\forall sp. \text{eval\_pdt } v e = \text{Inl } sp \longrightarrow (\exists x. x \notin \text{fv } \varphi \wedge v x \neq \text{undefined}) \vee \text{s\_check } \sigma v \varphi sp) \wedge$ 
     $(\forall vp. \text{eval\_pdt } v e = \text{Inr } vp \longrightarrow (\exists x. x \notin \text{fv } \varphi \wedge v x \neq \text{undefined}) \vee \text{v\_check } \sigma v \varphi vp)$ 
  and  $\text{eval\_pdt } v e = \text{Inl } sp$  for e v sp
  using that eval_pdt_cong[of e v  $\lambda x. \text{if } x \in \text{fv } \varphi \text{ then } v x \text{ else undefined}$ ]
    check_fv_cong[of  $\varphi v \lambda x. \text{if } x \in \text{fv } \varphi \text{ then } v x \text{ else undefined}$ ]
  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  $\forall v. (\forall sp. \text{eval\_pdt } v e = \text{Inl } sp \longrightarrow (\exists x. x \notin \text{fv } \varphi \wedge v x \neq \text{undefined}) \vee \text{s\_check } \sigma v \varphi sp) \wedge$ 
     $(\forall vp. \text{eval\_pdt } v e = \text{Inr } vp \longrightarrow (\exists x. x \notin \text{fv } \varphi \wedge v x \neq \text{undefined}) \vee \text{v\_check } \sigma v \varphi vp)$ 
  and  $\text{eval\_pdt } v e = \text{Inr } vp$  for e v vp
  using that eval_pdt_cong[of e v  $\lambda x. \text{if } x \in \text{fv } \varphi \text{ then } v x \text{ else undefined}$ ]
    check_fv_cong[of  $\varphi v \lambda x. \text{if } x \in \text{fv } \varphi \text{ then } v x \text{ else undefined}$ ]
  by (auto dest!: spec[of _ vp] vars_order_vars simp: subset_eq)
show ?thesis
  using check_all_completeness_aux[of vs  $\varphi i \lambda_. \text{undefined } \sigma$ ] assms
  unfolding check_all_alt p_check_def
  by (auto simp: isl_def p_check_def p_at_def dest!: spec[of _ v]
    dest: check_soundness soundness split: sum.splits)
qed

lemma check_all_soundness_aux:  $\text{check\_all } \sigma \varphi e \implies p = \text{eval\_pdt } v e \implies \text{isl } p \longleftrightarrow \text{sat } \sigma v (p\_at p) \varphi$ 
  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:  $\text{check\_all } \sigma \varphi e \implies \text{pdt\_at } i e \implies \text{isl } (\text{eval\_pdt } v e) \longleftrightarrow \text{sat } \sigma v i \varphi$ 
  by (drule check_all_soundness_aux[OF _ refl, of _ _ v]) (auto simp: pdt_at_p_at_eval_pdt)

```

`unbundle no MFOTL_syntax`

10 Type of Events

10.1 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 \Rightarrow string8 \rightarrow (OCaml) id`
`| constant literal_string8 :: string8 \Rightarrow String.literal \rightarrow (OCaml) id`

`ML <structure String8 = struct fun to_term x = @{term Abs_string8} $ HOLogic.mk_string x; end;>`

`code_printing`

`type_constructor string8 \rightarrow (Eval) string`
`| constant string8_literal :: String.literal \Rightarrow string8 \rightarrow (Eval) _`
`| constant HOL.equal :: string8 \Rightarrow string8 \Rightarrow bool \rightarrow (Eval) infixl 6 =`
`| constant (\leq) :: string8 \Rightarrow string8 \Rightarrow bool \rightarrow (Eval) infixl 6 <=`
`| constant (<) :: string8 \Rightarrow string8 \Rightarrow bool \rightarrow (Eval) infixl 6 <`
`| constant empty_string :: string8 \rightarrow (Eval)`
`| constant Code_Evaluation.term_of :: string8 \Rightarrow term \rightarrow (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

```

10.2 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

```

11 Code Generation

11.1 Type Class Instances

```

class universe =
  fixes universe :: 'a list option
  assumes infinite: universe = None ⇒ 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 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  $\times$  'b) list option where universe_prod =
  (case (universe, universe) of (Some xs, Some ys)  $\implies$  Some (List.product xs ys) | _  $\implies$  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)  $\implies$  Some (map Inl xs @ map Inr ys) | _  $\implies$  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  $\implies$  Some (None # map Some xs) | _  $\implies$  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  $\implies$  '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 ≠ b for a b :: 'b
  using that by (metis (full_types) UNIV_I card_1_singleton_iff singletonD)
have 2: ys = zs
if distinct (xs::'a list) and length ys = length xs and length zs = length xs
and ∀ a. the (map_of (zip xs ys) a) = the (map_of (zip xs zs) a)
for xs :: 'a list and ys zs :: 'b list
using that by (metis map_fst_zip map_of_eqI map_of_zip_inject map_of_zip_is_None option.expand)
have 3: ∃ zs. length zs = length xs ∧ set zs ⊆ set ys ∧ (∀ x. f x = the (map_of (zip xs zs) x))
if ∀ x. x ∈ set xs ∀ x. x ∈ set ys
for xs ys and f :: 'a ⇒ 'b
using that by (metis length_map map_of_zip_map option.sel subsetI)
show OFCLASS('a ⇒ 'b, universe_class)
by standard
(auto 0 3 simp: universe_fun_def set_eq_iff fun_eq_iff image_iff set_n_lists distinct_map
inj_on_def distinct_n_lists finite_UNIV_fun dest!: infinite finite
split: option.splits list.splits intro: 1 2 3)
qed
end
instantiation event_data :: universe begin
definition universe_event_data :: event_data list option where universe_event_data = None
instance by standard (simp_all add: infinite_UNIV_event_data universe_event_data_def)
end

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

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

instance event_data :: equal by standard

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

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

```

11.2 Progress

```

fun progress :: ('n, 'd) trace ⇒ ('n, 'd) Formula.formula ⇒ nat ⇒ nat where
  progress σ Formula.TT j = j
| progress σ Formula.FF j = j

```

```

| progress σ (Formula.Eq_Const _ _) j = j
| progress σ (Formula.Pred _ _) j = j
| progress σ (Formula.Neg φ) j = progress σ φ j
| progress σ (Formula.Or φ ψ) j = min (progress σ φ j) (progress σ ψ j)
| progress σ (Formula.And φ ψ) j = min (progress σ φ j) (progress σ ψ j)
| progress σ (Formula.Imp φ ψ) j = min (progress σ φ j) (progress σ ψ j)
| progress σ (Formula.Iff φ ψ) j = min (progress σ φ j) (progress σ ψ j)
| progress σ (Formula.Exists _ φ) j = progress σ φ j
| progress σ (Formula.Forall _ φ) j = progress σ φ j
| progress σ (Formula.Prev I φ) j = (if j = 0 then 0 else min (Suc (progress σ φ j)) j)
| progress σ (Formula.Next I φ) j = progress σ φ j - 1
| progress σ (Formula.Once I φ) j = progress σ φ j
| progress σ (Formula.Historically I φ) j = progress σ φ j
| progress σ (Formula.Eventually I φ) j =
  Inf {i. ∀k. k < j ∧ k ≤ (progress σ φ j) → (τ σ k - τ σ i) ≤ right I}
| progress σ (Formula.Always I φ) j =
  Inf {i. ∀k. k < j ∧ k ≤ (progress σ φ j) → (τ σ k - τ σ i) ≤ right I}
| progress σ (Formula.Since φ I ψ) j = min (progress σ φ j) (progress σ ψ j)
| progress σ (Formula.Until φ I ψ) j =
  Inf {i. ∀k. k < j ∧ k ≤ min (progress σ φ j) (progress σ ψ j) → (τ σ k - τ σ i) ≤ right I}
| progress σ (Formula.MatchP I r) j = min_regex_default (progress σ) r j
| progress σ (Formula.MatchF I r) j = Inf {i. ∀k. k < j ∧ k ≤ min_regex_default (progress σ) r j →
τ σ i + right I ≥ τ σ k}

```

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: intro: cInf_lower intro!: antisym[OF _ Min_le]*)

(*metis Inf_nat_def1 empty_iff mem_Collect_eq*)

lemma *progress_Eventually_code*: $progress \sigma (Formula.Eventually I \varphi) j =$

(*let m = min j (Suc (progress σ φ j)) - 1 in Min (Set.filter (λi. enat (δ σ m i) ≤ right I) \{..j\})*)

proof -

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

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*: $progress \sigma (Formula.Always I \varphi) j =$

(*let m = min j (Suc (progress σ φ j)) - 1 in Min (Set.filter (λi. enat (δ σ m i) ≤ right I) \{..j\})*)

proof -

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

have P_j : $P j$

```

    by (auto simp: P_def enat_0)
  have all_wit: ( $\forall k \in \{..<m\}. \text{enat } (\delta \sigma k i) \leq \text{right } I \iff (\text{enat } (\delta \sigma (m - 1) i) \leq \text{right } I)$ ) 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  $\text{less\_Suc\_eq\_le}$ 
      by (rule  $\text{diff\_le\_mono}$ )
    then show ?thesis
      by (auto simp: Suc) (meson order.trans enat_ord_simps(1))
  qed (auto simp: enat_0)
  show ?thesis
    unfolding  $\text{progress.simps Let\_def P\_def[symmetric] Inf\_Min}$  [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: ( $\forall k \in \{..<m\}. \text{enat } (\delta \sigma k i) \leq \text{right } I \iff (\text{enat } (\delta \sigma (m - 1) i) \leq \text{right } I)$ ) 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  $\text{less\_Suc\_eq\_le}$ 
        by (rule  $\text{diff\_le\_mono}$ )
      then show ?thesis
        by (auto simp: Suc) (meson order.trans enat_ord_simps(1))
    qed (auto simp: enat_0)
    show ?thesis
      unfolding  $\text{progress.simps Let\_def P\_def[symmetric] Inf\_Min}$  [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

```

11.3 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  $\bigwedge x. x \in \text{set list} \implies \text{snd } x \leq m$ 
  and  $\bigwedge x. x \in \text{set list} \implies \text{finite } (\text{fst } x)$ 
  shows  $\text{ssorted } (\text{smap snd } (\text{list } @- \text{smap } (\lambda n. (\{\}, n + m)) \text{nats})) \wedge$ 
     $\text{sincreasing } (\text{smap snd } (\text{list } @- \text{smap } (\lambda n. (\{\}, n + m)) \text{nats})) \wedge$ 
     $\text{sfinite } (\text{smap fst } (\text{list } @- \text{smap } (\lambda n. (\{\}, n + m)) \text{nats}))$ 
  proof -
    have  $A: \forall x \in \text{set list}. n \leq \text{snd } x \implies n \leq m \implies$ 
       $n \leq (\text{map snd list } @- \text{smap } (\lambda x. x + m) \text{nats}) !! i$  for  $n i$ 
    and  $\text{list} :: ('a \text{ set } \times \text{nat}) \text{ list}$ 

```

```

proof (induction i arbitrary: n)
  case (Suc i)
  then show ?case
    by (auto simp: shift_snth nth_tl)
qed (auto simp add: list.map_sel(1))
then have ssorted (smap snd (list @- smap (λn. ({}), n + m)) nats))
  using assms
  by (induction list) (auto simp: stream.map_comp o_def ssorted_iff_mono snth_Stream_eq
    split: nat.splits)
moreover have sincreasing (smap snd (list @- smap (λn. ({}), n + m)) nats))
  using assms
proof (induction list)
  case Nil
  then show ?case
    by (simp add: sincreasing_def) presburger
next
  case (Cons a as)
  have IH:  $\bigwedge x. \exists i. x < \text{smap snd } (as @- \text{smap } (\lambda n. (\{\}, n + m)) \text{ nats}) !! i$ 
    using Cons
    by (simp add: sincreasing_def)
  show ?case
    using IH
    by (simp add: sincreasing_def)
      (metis snth_Stream)
qed
moreover have sfinite (smap fst (list @- smap (λn. ({}), n + m)) nats))
  using assms(3)
proof (induction list)
  case Nil
  then show ?case by (simp add: sfinite_def)
next
  case (Cons a as)
  then have fin: finite (fst a)
    by simp
  show ?case
    using Cons
    by (auto simp add: sfinite_def snth_Stream_eq split: nat.splits)
qed
ultimately show ?thesis
  by simp
qed

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

setup_lifting type_definition_trace_mapping

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

lift_definition trace_mapping_of_list :: ('a set × nat) list ⇒ 'a trace_mapping is
  λxs. if sorted (map snd xs) ∧ (∀ x ∈ set xs. finite (fst x)) then (if xs = [] then (0, 0, Mapping.empty)

```

```

else (length xs, snd (last xs), Mapping.bulkload xs)
else (0, 0, Mapping.empty)
by (auto simp: lookup_bulkload_Some sorted_iff_nth_Suc last_conv_nth
    list_all_iff_in_set_conv_nth Ball_def Bex_def image_iff split: nat.splits)

```

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

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

proof (goal_cases)

case (1 prod)

obtain n m t **where** prod_def: prod = (n, m, t)

by (cases prod) auto

have props: Mapping.keys $t = \{..<n\}$

sorted (map (snd \circ (the \circ Mapping.lookup t)) $[0..<n]$)

(case n of 0 \Rightarrow True | Suc n' \Rightarrow (case Mapping.lookup t n' of Some (X', t') \Rightarrow $t' \leq m$ | None \Rightarrow False))

($\forall n' < n.$ case Mapping.lookup t n' of Some (X', t') \Rightarrow finite X' | None \Rightarrow False)

using 1 **by** (auto simp add: prod_def)

have aux: $x \in$ set (map (the \circ Mapping.lookup t) $[0..<n]$) \Longrightarrow snd $x \leq m$ **for** x

using props(2,3) less_Suc_eq_le

by (fastforce simp: sorted_iff_nth_mono split: nat.splits option.splits)

have aux2: $x \in$ set (map (the \circ Mapping.lookup t) $[0..<n]$) \Longrightarrow finite (fst x) **for** x

using props(1,4)

by (auto split: nat.splits option.splits)

show ?case

unfolding prod_def prod.case

by (rule extend_is_stream[**where** ? $m=m$]) (use props aux aux2 **in** \langle auto simp: prod_def \rangle)

qed

code_datatype Trace_Mapping

definition trace_of_list $xs =$ Trace_Mapping (trace_mapping_of_list xs)

lemma Γ _rbt_code[code]: Γ (Trace_Mapping t) $i =$ fst (trace_mapping_nth t i)

by transfer (auto split: prod.splits)

lemma τ _rbt_code[code]: τ (Trace_Mapping t) $i =$ snd (trace_mapping_nth t i)

by transfer (auto split: prod.splits)

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

$xs ! i =$ (Γ (trace_of_list xs) i , τ (trace_of_list xs) i)

unfolding trace_of_list_def

by transfer (auto simp: lookup_bulkload_Some)

11.4 Auxiliary results

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

lemma sum_proofs_empty[simp]: sum_proofs f [] = 0

by (auto simp: sum_proofs_def)

lemma sum_proofs_fundef_cong[fundef_cong]: ($\wedge x. x \in$ set $xs \Longrightarrow f$ $x = f'$ x) \Longrightarrow

sum_proofs f $xs =$ sum_proofs f' xs

by (induction xs) (auto simp: sum_proofs_def)

lemma *sum_proofs_Cons*:

fixes $f :: 'a \Rightarrow \text{nat}$

shows $\text{sum_proofs } f (p \# qs) = f p + \text{sum_proofs } f qs$

by (*auto simp: sum_proofs_def split: list.splits*)

lemma *sum_proofs_app*:

fixes $f :: 'a \Rightarrow \text{nat}$

shows $\text{sum_proofs } f (qs @ [p]) = f p + \text{sum_proofs } f qs$

by (*auto simp: sum_proofs_def split: list.splits*)

context

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

begin

function (*sequential*) $s_pred :: ('n, 'd) \text{ sproof} \Rightarrow \text{nat}$

and $v_pred :: ('n, 'd) \text{ vproof} \Rightarrow \text{nat}$ **where**

$s_pred (STT _) = 1$

| $s_pred (SEq_Const _ _ _) = 1$

| $s_pred (SPred _ r _) = w r$

| $s_pred (SNeg vp) = (v_pred vp) + 1$

| $s_pred (SOrL sp1) = (s_pred sp1) + 1$

| $s_pred (SOrR sp2) = (s_pred sp2) + 1$

| $s_pred (SAnd sp1 sp2) = (s_pred sp1) + (s_pred sp2) + 1$

| $s_pred (SImpL vp1) = (v_pred vp1) + 1$

| $s_pred (SImpR sp2) = (s_pred sp2) + 1$

| $s_pred (SIffSS sp1 sp2) = (s_pred sp1) + (s_pred sp2) + 1$

| $s_pred (SIffVV vp1 vp2) = (v_pred vp1) + (v_pred vp2) + 1$

| $s_pred (SExists _ _ sp) = (s_pred sp) + 1$

| $s_pred (SForall _ _ part) = (\text{sum_proofs } s_pred (vals part)) + 1$

| $s_pred (SPrev sp) = (s_pred sp) + 1$

| $s_pred (SNext sp) = (s_pred sp) + 1$

| $s_pred (SOnce _ sp) = (s_pred sp) + 1$

| $s_pred (SEventually _ sp) = (s_pred sp) + 1$

| $s_pred (SHistorically _ _ sps) = (\text{sum_proofs } s_pred sps) + 1$

| $s_pred (SHistoricallyOut _) = 1$

| $s_pred (SAlways _ _ sps) = (\text{sum_proofs } s_pred sps) + 1$

| $s_pred (SSince sp2 sp1s) = (\text{sum_proofs } s_pred (sp2 \# sp1s)) + 1$

| $s_pred (SUntil sp1s sp2) = (\text{sum_proofs } s_pred (sp1s @ [sp2])) + 1$

| $v_pred (VFF _) = 1$

| $v_pred (VEq_Const _ _ _) = 1$

| $v_pred (VPred _ r _) = w r$

| $v_pred (VNeg sp) = (s_pred sp) + 1$

| $v_pred (VOr vp1 vp2) = ((v_pred vp1) + (v_pred vp2)) + 1$

| $v_pred (VAndL vp1) = (v_pred vp1) + 1$

| $v_pred (VAndR vp2) = (v_pred vp2) + 1$

| $v_pred (VImp sp1 vp2) = ((s_pred sp1) + (v_pred vp2)) + 1$

| $v_pred (VIffSV sp1 vp2) = ((s_pred sp1) + (v_pred vp2)) + 1$

| $v_pred (VIffVS vp1 sp2) = ((v_pred vp1) + (s_pred sp2)) + 1$

| $v_pred (VExists _ part) = (\text{sum_proofs } v_pred (vals part)) + 1$

| $v_pred (VForall _ _ vp) = (v_pred vp) + 1$

| $v_pred (VPrev vp) = (v_pred vp) + 1$

| $v_pred (VPrevZ) = 1$

| $v_pred (VPrevOutL _) = 1$

| $v_pred (VPrevOutR _) = 1$

| $v_pred (VNext vp) = (v_pred vp) + 1$

| $v_pred (VNextOutL _) = 1$

| $v_pred (VNextOutR _) = 1$

| $v_pred (VOnceOut _) = 1$

```

| 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

11.5 v_check_exec setup

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

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

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

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

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

lemma $LTP_plus_ge_iff: j \leq LTP \ \sigma \ (\tau \ \sigma \ i + n) \iff \delta \ \sigma \ j \ i \leq n$
 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 $\text{not_in}: k \notin \{ia. \tau \ \sigma \ ia \leq \tau \ \sigma \ i - n\}$ **if** $j \leq k$ **for** k
 using $\text{assms } \tau_mono[OF \ \text{that}, \ of \ \sigma]$
 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 $\text{finite } \{ia. \tau \ \sigma \ ia \leq \tau \ \sigma \ i - n\}$
 using $\text{subset_eq_atLeast0_lessThan_finite}$
 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  $LTP \sigma (\tau \sigma i - n) < j \iff (if \neg j \leq i \wedge n = 0 \text{ then } \delta \sigma j i > 0 \text{ else } \delta \sigma i j < n)$ 
proof (cases  $j \leq i$ )
  case True
  then show ?thesis
    using assms  $i\_le\_LTPi\_minus[of \sigma n i]$  LTP_minus_lt_iff[ $of j i \sigma n$ ]
    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  $j = LTP \sigma (\tau \sigma i - n) \iff$ 
  ( $case\ n\ of\ 0 \Rightarrow i \leq j \wedge \delta \sigma j i = 0 \wedge \delta \sigma (Suc\ j) j > 0$ 
  |  $\_ \Rightarrow j \leq i \wedge n \leq \delta \sigma i j \wedge \delta \sigma i (Suc\ j) < n$ )
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[ $of \sigma (LTP \sigma (\tau \sigma i)) Suc j$ ]
    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:  $Suc\_eq\_iff[of j LTP \sigma (\tau \sigma i - Suc\ n')]$  less_Suc_eq_le[ $of LTP \sigma (\tau \sigma i - Suc\ n') j$ ,
      symmetric] LTP_minus_ge_iff LTP_minus_lt_iff)
  qed

```

lemma *LTP_plus_eq_iff*:

shows $j = LTP\ \sigma\ (\tau\ \sigma\ i + n) \longleftrightarrow (\delta\ \sigma\ j\ i \leq n \wedge \delta\ \sigma\ (Suc\ j)\ i > n)$
unfolding *eq_iff[of j LTP σ (τ σ i + n)]*
by (*meson LTP_plus_ge_iff linorder_not_less not_less_eq_eq*)

lemma *LTP_p_def*: $\tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \implies LTP_p\ \sigma\ i\ I = (case\ left\ I\ of\ 0 \Rightarrow i \mid _ \Rightarrow LTP\ \sigma\ (\tau\ \sigma\ i - left\ I))$

using *i_le_LTPi* **by** (*auto simp: min_def i_LTP_tau split: nat.splits*)

definition *check_upt_LTP_p* $\sigma\ I\ li\ xs\ i \longleftrightarrow (case\ xs\ of\ [] \Rightarrow$

(*case left I of 0 ⇒ i < li | Suc n ⇒*
(if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I))
 $\mid _ \Rightarrow xs = [li..<li + length\ xs] \wedge$
(case left I of 0 ⇒ li + length xs - 1 = i | Suc n ⇒
(li + length xs - 1 ≤ i ∧ left I ≤ δ σ i (li + length xs - 1) ∧ δ σ i (li + length xs) < left I)))

lemma *check_upt_l_cong*:

assumes $\bigwedge j. j \leq max\ i\ li \implies \tau\ \sigma\ j = \tau\ \sigma'\ j$

shows $check_upt_LTP_p\ \sigma\ I\ li\ xs\ i = check_upt_LTP_p\ \sigma'\ I\ li\ xs\ i$

proof –

have $li + length\ ys \leq i \implies Suc\ n \leq \delta\ \sigma'\ i\ (li + length\ ys) \implies$

$(Suc\ (li + length\ ys)) \leq i$ **for** $ys :: nat\ list$ **and** n

by (*cases li + length ys = i*) *auto*

then show *?thesis*

using *assms*

by (*fastforce simp: check_upt_LTP_p_def Let_def simp del: upt.simps split: list.splits nat.splits*)

qed

lemma *check_upt_LTP_p_eq*:

assumes $\tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$

shows $xs = [li..<Suc\ (LTP_p\ \sigma\ i\ I)] \longleftrightarrow check_upt_LTP_p\ \sigma\ I\ li\ xs\ i$

proof –

have $li + length\ xs = Suc\ (LTP_p\ \sigma\ i\ I) \longleftrightarrow li + length\ xs - 1 = LTP_p\ \sigma\ i\ I$ **if** $xs \neq []$

using *that*

by (*cases xs*) *auto*

then have $xs = [li..<Suc\ (LTP_p\ \sigma\ i\ I)] \longleftrightarrow (xs = [] \wedge LTP_p\ \sigma\ i\ I < li) \vee$

$(xs \neq [] \wedge xs = [li..<li + length\ xs] \wedge li + length\ xs - 1 = LTP_p\ \sigma\ i\ I)$

by *auto*

moreover have $\dots \longleftrightarrow (xs = [] \wedge$

(case left I of 0 ⇒ i < li | Suc n ⇒

(if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I))) \vee

$(xs \neq [] \wedge xs = [li..<li + length\ xs] \wedge$

(case left I of 0 ⇒ li + length xs - 1 = i | Suc n ⇒

(case left I of 0 ⇒ i ≤ li + length xs - 1 ∧

$\delta\ \sigma\ (li + length\ xs - 1)\ i = 0 \wedge 0 < \delta\ \sigma\ (Suc\ (li + length\ xs - 1))\ (li + length\ xs - 1)$

$\mid Suc\ n \Rightarrow li + length\ xs - 1 \leq i \wedge$

$left\ I \leq \delta\ \sigma\ i\ (li + length\ xs - 1) \wedge \delta\ \sigma\ i\ (Suc\ (li + length\ xs - 1)) < left\ I))$)

using *LTP_p_def[OF assms, symmetric]*

unfolding *LTP_minus_lt_iff[OF assms, symmetric]*

unfolding *LTP_minus_eq_iff[OF assms, symmetric]*

by (*auto split: nat.splits*)

moreover have $\dots \longleftrightarrow (case\ xs\ of\ [] \Rightarrow$

(case left I of 0 ⇒ i < li | Suc n ⇒

(if ¬ li ≤ i ∧ left I = 0 then 0 < δ σ li i else δ σ i li < left I)))

$\mid _ \Rightarrow xs = [li..<li + length\ xs] \wedge$

(case left I of 0 ⇒ li + length xs - 1 = i | Suc n ⇒

$(li + length\ xs - 1 \leq i \wedge left\ I \leq \delta\ \sigma\ i\ (li + length\ xs - 1) \wedge \delta\ \sigma\ i\ (li + length\ xs) < left\ I))$)

by (*auto split: nat.splits list.splits*)

ultimately show *?thesis*
unfolding *check_upt_LTP_p_def*
by *simp*
qed

declare *v_check_exec_simps [code] s_check_exec_simps [code]*

lemma *v_check_exec_Once_code[code]: v_check_exec σ vs (Formula.Once I φ) vp = (case vp of*
VOnce i li vps \Rightarrow
(case right I of $\infty \Rightarrow$ li = 0 | enat b \Rightarrow ((li = 0 \vee b < δ σ i (li - 1)) \wedge δ σ i li \leq b))
 \wedge τ σ 0 + left I \leq τ σ i
 *\wedge *check_upt_LTP_p σ I li (map v_at vps) i \wedge Ball (set vps) (v_check_exec σ vs φ)**
| VOnceOut i \Rightarrow τ σ i < τ σ 0 + left I
| _ \Rightarrow False)
by *(auto simp: Let_def check_upt_LTP_p_eq ETP_minus_le_iff ETP_minus_eq_iff split: vproof.splits*
enat.splits simp del: upt_Suc)

lemma *s_check_exec_Historically_code[code]: s_check_exec σ vs (Formula.Historically I φ) vp = (case*
vp of
SHistorically i li vps \Rightarrow
(case right I of $\infty \Rightarrow$ li = 0 | enat b \Rightarrow ((li = 0 \vee b < δ σ i (li - 1)) \wedge δ σ i li \leq b))
 \wedge τ σ 0 + left I \leq τ σ i
 *\wedge *check_upt_LTP_p σ I li (map s_at vps) i \wedge Ball (set vps) (s_check_exec σ vs φ)**
| SHistoricallyOut i \Rightarrow τ σ i < τ σ 0 + left I
| _ \Rightarrow False)
by *(auto simp: Let_def check_upt_LTP_p_eq ETP_minus_le_iff ETP_minus_eq_iff split: sproof.splits*
enat.splits simp del: upt_Suc)

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

lemma *ETP_f_le_iff: max i (ETP σ (τ σ i + a)) \leq j \longleftrightarrow i \leq j \wedge δ σ 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 σ (τ σ i + n)) \longleftrightarrow (case n of 0 \Rightarrow j \leq i*
| Suc n' \Rightarrow (case j of 0 \Rightarrow True | Suc j' \Rightarrow δ σ j' i < n))

proof *(cases n)*

case 0

then show *?thesis*

by *(auto simp: max_def) (metis i_ge_etpi verit_la_disequality)*

next

case (Suc n')

have *max: max i (ETP σ (τ σ i + n)) = ETP σ (τ σ i + n)*

by *(auto simp: max_def Suc)*

(metis Groups.ab_semigroup_add_class.add.commute ETP_ge less_or_eq_imp_le plus_1_eq_Suc)

have $j \leq \max i (ETP \sigma (\tau \sigma i + n)) \longleftrightarrow (\forall ia. \tau \sigma i + n \leq \tau \sigma ia \longrightarrow j \leq ia)$
unfolding *max*
unfolding *ETP_def*
by (*meson LeastI_ex Least_le order.trans ex_le_τ*)
moreover have $\dots \longleftrightarrow (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \neg \tau \sigma i + n \leq \tau \sigma j')$
by (*auto split: nat.splits*) (*meson i_ETP_tau le_trans not_less_eq_eq*)
moreover have $\dots \longleftrightarrow (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta \sigma j' i < n)$
by (*auto simp: Suc split: nat.splits*)
ultimately show *?thesis*
by (*auto simp: Suc*)
qed

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

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

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 $xs = [ETP_f\ \sigma\ i\ I..<Suc\ hi] \longleftrightarrow (let\ j = Suc\ hi - length\ xs\ in$
 $(xs = [] \wedge (case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta \sigma hi\ i < left\ I)) \vee$

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

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

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

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

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 σ i I, symmetric*]

by *auto*

moreover have $\dots \longleftrightarrow (let\ j = Suc\ hi - length\ xs\ in$
 $(case\ xs\ of\ [] \Rightarrow (case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta \sigma hi\ i < left\ I)$

$\mid _ \Rightarrow (xs = [j..<Suc\ hi] \wedge$

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

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

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

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*]: $v_check_exec\ \sigma\ vs\ (Formula.Eventually\ I\ \varphi)\ vp = (case$
 $vp\ of$
 $\quad VEventually\ i\ hi\ vps \Rightarrow$

$(\text{case right } I \text{ of } \infty \Rightarrow \text{False} \mid \text{enat } b \Rightarrow (\delta \sigma \text{ hi } i \leq b \wedge b < \delta \sigma (\text{Suc hi}) i)) \wedge$
 $\text{check_upt_ETP_f } \sigma \text{ I } i (\text{map } v_at \text{ vps}) \text{ hi} \wedge \text{Ball } (\text{set } vps) (v_check_exec \sigma \text{ vs } \varphi)$
 $\mid _ \Rightarrow \text{False})$
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]: $s_check_exec \sigma \text{ vs } (\text{Formula.Always } I \varphi) \text{ sp} = (\text{case sp of}$
 $SAlways \text{ i hi sps} \Rightarrow$
 $(\text{case right } I \text{ of } \infty \Rightarrow \text{False} \mid \text{enat } b \Rightarrow (\delta \sigma \text{ hi } i \leq b \wedge b < \delta \sigma (\text{Suc hi}) i))$
 $\wedge \text{check_upt_ETP_f } \sigma \text{ I } i (\text{map } s_at \text{ sps}) \text{ hi} \wedge \text{Ball } (\text{set } sps) (s_check_exec \sigma \text{ vs } \varphi)$
 $\mid _ \Rightarrow \text{False})$
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]: $v_check_exec \sigma \text{ vs } (\text{Formula.Until } \varphi \text{ I } \psi) \text{ vp} = (\text{case vp of}$
 $VUntil \text{ i vp2s vp1} \Rightarrow$
 $\text{let } j = v_at \text{ vp1} \text{ in}$
 $(\text{case right } I \text{ of } \infty \Rightarrow \text{True} \mid \text{enat } b \Rightarrow j < \text{LTP_f } \sigma \text{ i } b)$
 $\wedge i \leq j \wedge \text{check_upt_ETP_f } \sigma \text{ I } i (\text{map } v_at \text{ vp2s}) \text{ j}$
 $\wedge v_check_exec \sigma \text{ vs } \varphi \text{ vp1} \wedge \text{Ball } (\text{set } vp2s) (v_check_exec \sigma \text{ vs } \psi)$
 $\mid VUntilInf \text{ i hi vp2s} \Rightarrow$
 $(\text{case right } I \text{ of } \infty \Rightarrow \text{False} \mid \text{enat } b \Rightarrow (\delta \sigma \text{ hi } i \leq b \wedge b < \delta \sigma (\text{Suc hi}) i)) \wedge$
 $\text{check_upt_ETP_f } \sigma \text{ I } i (\text{map } v_at \text{ vp2s}) \text{ hi} \wedge \text{Ball } (\text{set } vp2s) (v_check_exec \sigma \text{ vs } \psi)$
 $\mid _ \Rightarrow \text{False})$
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*)

11.6 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**
 $\text{ETP_rec } \sigma \text{ t } i = (\text{if } \tau \sigma i \geq t \text{ then } i \text{ else } \text{ETP_rec } \sigma \text{ t } (i + 1))$
by *pat_completeness auto*
termination
using *ETP_aux*
by (*relation measure* $(\lambda(\sigma, t, i). \text{Suc } (\text{ETP } \sigma \text{ t}) - i)$)
(fastforce simp: ETP_def)+

lemma *ETP_rec_sound*: $\text{ETP_rec } \sigma \text{ t } j = (\text{LEAST } i. i \geq j \wedge t \leq \tau \sigma i)$

proof (*induction* $\sigma \text{ t } j$ *rule: ETP_rec.induct*)
case $(1 \sigma \text{ t } i)$
show *?case*
proof (*cases* $\tau \sigma i \geq t$)
case *True*
then show *?thesis*
by *simp (metis (no_types, lifting) Least_equality order_refl)*
next
case *False*
then show *?thesis*
using $1[\text{OF } \text{False}]$
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]: $\text{ETP } \sigma \text{ t} = \text{ETP_rec } \sigma \text{ t } 0$

```

using ETP_rec_sound[of  $\sigma$   $t$  0]
by (auto simp: ETP_def)

lemma LTP_aux:
  assumes  $\tau$   $\sigma$  (Suc  $i$ )  $\leq t$ 
  shows  $i \leq \text{Max } \{i. \tau \sigma i \leq t\}$ 
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 ( $\lambda(\sigma, t, i). \text{Suc } (LTP \sigma t) - i$ )) (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:  $\{i. \text{Suc } j \leq i \wedge \tau \sigma i \leq t\} = \{i. j \leq i \wedge \tau \sigma i \leq t\} - \{j\}$ 
    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_ $\tau$ D)
  qed
qed

lemma LTP_code[code]: LTP  $\sigma$   $t$  = (if  $t < \tau \sigma$  0
  then Code.abort (STR "LTP: undefined") ( $\lambda\_.$  LTP  $\sigma$   $t$ )
  else LTP_rec  $\sigma$   $t$  0)
using LTP_rec_sound[of  $\sigma$   $t$  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)

declare subset_code [code]

```

lemma *coset_subset_set_code*[code]:
 (*List.coset* (*xs* :: *_* :: *universe list*) \subseteq *set ys*) = (*case universe of None* \Rightarrow *False*
 | *Some zs* $\Rightarrow \forall z \in \text{set } zs. z \in \text{set } xs \vee z \in \text{set } ys$)
using *finite_compl finite_subset*
by (*auto split: option.splits dest!: infinite finite*)

declare *is_empty_set* [code]

lemma *is_empty_coset*[code]: *Set.is_empty* (*List.coset* (*xs* :: *_* :: *universe list*)) =
 (*case universe of None* \Rightarrow *False*
 | *Some zs* $\Rightarrow \forall z \in \text{set } zs. z \in \text{set } xs$)
using *coset_subset_set_code*[of *xs*] **by** (*auto simp: split: option.splits dest: infinite finite*)

11.7 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 $\sigma \varphi e = (\text{if } (\text{distinct_paths } e \wedge \text{check_all_aux } \sigma (\lambda_. \text{UNIV}) \varphi e) \text{ then } \text{None} \text{ else } \text{Some } (\text{collect_paths_aux } \{\}\} \sigma (\lambda_. \text{UNIV}) \varphi e))$

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* \times '*a*') *list* \Rightarrow (*event_data*, '*a*') *part* **is**
 $\lambda xs.$
let *Ds* = *map fst xs* *in*
if $\{\} \in \text{set } Ds$
 $\vee (\exists D \in \text{set } Ds. \exists E \in \text{set } Ds. D \neq E \wedge D \cap E \neq \{\})$
 $\vee \neg \text{distinct } Ds$
 $\vee (\bigcup D \in \text{set } Ds. D) \neq \text{UNIV}$ *then* [(*UNIV*, *undefined*)] *else* *xs*
by (*auto simp: partition_on_def disjoint_def*)

lemma *rm_code*[*code_unfold*]: $rm\ S = Set.filter\ (\lambda(i,j).\ i < j)\ S$
by *auto*

export_code *interval enat nat_of_integer integer_of_nat*
STT SSkip VSkip Formula.TT Regex.Skip 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*

12 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 ∧ A1 A2. A1 ∈ set (map fst (inP1 P1 f v1 part2)) ⇒
A2 ∈ set (map fst (inP1 P1 f v1 part2)) ⇒ A1 ≠ A2 ⇒ *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 {} ∉ set (map fst (inP1 P1 f v1 part2))

```

using assms
by (auto simp: map_filter_def split: if_splits)
show distinct (map fst ((P1, v1) # part1)) ==> distinct (map fst part2) ==>
distinct (map fst (inP1 P1 f v1 part2))
using partition_onD2[OF assms(3), unfolded disjoint_def, simplified, rule_format]
by (clarsimp simp: inP1_def map_filter_def distinct_map inj_on_def Ball_def) blast
qed

lemma wf_part_list_filter_minus:
defines notinP2 P1 f v1 part2
   $\equiv$  List.map_filter ( $\lambda(P2, v2).$  if  $P2 - P1 \neq \{\}$  then Some( $P2 - P1, v2$ ) else None) part2
assumes partition_on X (set (map fst ((P1, v1) # part1)))
and partition_on X (set (map fst part2))
shows partition_on (X - P1) (set (map fst (notinP2 P1 f v1 part2)))
and distinct (map fst ((P1, v1) # part1)) ==> distinct (map fst (part2)) ==>
distinct (map fst (notinP2 P1 f v1 part2))
proof (rule partition_onI)
show  $\bigcup$  (set (map fst (notinP2 P1 f v1 part2))) =  $X - P1$ 
proof -
have  $\exists P2. ((\exists x \in P2. x \notin P1) \longrightarrow (\exists v2. (P2, v2) \in \text{set part2})) \wedge (\exists x \in P2. x \notin P1) \wedge x \in P2$ 
if  $\bigcup$  (fst ' set part2) =  $P1 \cup \bigcup$  (fst ' set part1)  $x \notin P1$  ( $P1', v1$ )  $\in$  set part1  $x \in P1'$  for  $x P1' v1$ 
using that by (metis (no_types, lifting) UN_iff Un_iff fst_conv prod.collapse)
with partition_onD1[OF assms(2)] partition_onD1[OF assms(3)] show ?thesis
by (auto simp: map_filter_def subset_eq split_beta notinP2_def split: if_splits)
qed
show  $\bigwedge A1 A2. A1 \in \text{set (map fst (notinP2 P1 f v1 part2))} \implies$ 
 $A2 \in \text{set (map fst (notinP2 P1 f v1 part2))} \implies A1 \neq A2 \implies \text{disjnt } A1 A2$ 
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  $\{\} \notin \text{set (map fst (notinP2 P1 f v1 part2))}$ 
using assms
by (auto simp: map_filter_def split: if_splits)
show distinct (map fst ((P1, v1) # part1)) ==> distinct (map fst part2) ==>
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  $\bigcup$  (set (map fst part1)) =  $X - P1$ 
using partition_onD1[OF assms(1)] partition_onD2[OF assms(1)] assms(2)
by (auto simp: disjoint_def image_iff)
show  $\bigwedge A1 A2. A1 \in \text{set (map fst part1)} \implies A2 \in \text{set (map fst part1)} \implies A1 \neq A2 \implies \text{disjnt } A1$ 
 $A2$ 
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  $\{\} \notin \text{set (map fst part1)}$ 
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*: $\text{partition_on } X \text{ (set } xs) \implies \text{partition_on } Y \text{ (set } ys) \implies X \cap Y = \{\} \implies$
 $\text{partition_on } (X \cup Y) \text{ (set } (xs @ ys))$
by (*auto simp: partition_on_def intro!: disjoint_union*)

lemma *wf_part_list_merge_part2_raw*:

$\text{partition_on } X \text{ (set (map fst part1))} \wedge \text{distinct (map fst part1)} \implies$
 $\text{partition_on } X \text{ (set (map fst part2))} \wedge \text{distinct (map fst part2)} \implies$
 $\text{partition_on } X \text{ (set (map fst (merge_part2_raw f part1 part2)))}$
 $\wedge \text{distinct (map fst (merge_part2_raw f part1 part2))}$

proof(*induct f part1 part2 arbitrary: X rule: merge_part2_raw.induct*)

case (*2 f P1 v1 part1 part2*)

let *?inP1* = *List.map_filter* ($\lambda(P2, v2).$ *if* $P1 \cap P2 \neq \{\}$ *then* *Some* ($P1 \cap P2, f v1 v2$) *else* *None*)
part2

and *?notinP1* = *List.map_filter* ($\lambda(P2, v2).$ *if* $P2 - P1 \neq \{\}$ *then* *Some* ($P2 - P1, v2$) *else* *None*)
part2

have $P1 \cup X = X$

using *2.prem*s

by (*auto simp: partition_on_def*)

have *wf_part1*: $\text{partition_on } (X - P1) \text{ (set (map fst part1))}$

$\text{distinct (map fst part1)}$

using *wf_part_list_tail 2.prem*s **by** *auto*

moreover **have** *wf_notinP1*: $\text{partition_on } (X - P1) \text{ (set (map fst ?notinP1))}$

$\text{distinct (map fst (?notinP1))}$

using *wf_part_list_filter_minus[OF 2(2)[THEN conjunct1]]*

$2.prem$ s **by** *auto*

ultimately **have** *IH*: $\text{partition_on } (X - P1) \text{ (set (map fst (merge_part2_raw f part1 (?notinP1))))}$

$\text{distinct (map fst (merge_part2_raw f part1 (?notinP1)))}$

using *2.hyps[OF refl refl]* **by** *auto*

moreover **have** *wf_inP1*: $\text{partition_on } P1 \text{ (set (map fst ?inP1))}$ $\text{distinct (map fst ?inP1)}$

using *wf_part_list_filter_inter[OF 2(2)[THEN conjunct1]]*

$2.prem$ s **by** *auto*

moreover **have** $(\text{fst ' set } ?inP1) \cap (\text{fst ' set (merge_part2_raw f part1 (?notinP1)))} = \{\}$

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)]* $\langle P1 \cup X = X \rangle$ *wf_inP1(2) IH(2)*

by *simp*

qed *simp*

lemma *wf_part_list_merge_part3_raw*:

$\text{partition_on } X \text{ (set (map fst part1))} \wedge \text{distinct (map fst part1)} \implies$

$\text{partition_on } X \text{ (set (map fst part2))} \wedge \text{distinct (map fst part2)} \implies$

$\text{partition_on } X \text{ (set (map fst part3))} \wedge \text{distinct (map fst part3)} \implies$

$\text{partition_on } X \text{ (set (map fst (merge_part3_raw f part1 part2 part3)))}$

$\wedge \text{distinct (map fst (merge_part3_raw f part1 part2 part3))}$

proof(*induct f part1 part2 part3 arbitrary: X rule: merge_part3_raw.induct*)

case (*4 f v va vb vc vd ve*)

have $\text{partition_on } X \text{ (set (map fst (v \# va)))} \wedge \text{distinct (map fst (vb \# vc))}$

using *4* **by** *blast*

moreover **have** $\text{partition_on } X \text{ (set (map fst (vb \# vc)))} \wedge \text{distinct (map fst (vb \# vc))}$

using *4* **by** *blast*

ultimately **have** $\text{partition_on } X \text{ (set (map fst (merge_part2_raw f (v \# va) (vb \# vc))))}$

$\wedge \text{distinct (map fst (merge_part2_raw f (v \# va) (vb \# vc)))}$

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*))] (*λpt3*
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** $\langle \oplus \rangle$ 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* **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 **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 **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 **where**
do_iff *p1* *p2* = (case (*p1*, *p2*) of
(*Inl* *sp1*, *Inl* *sp2*) \Rightarrow [*Inl* (*SIfSS* *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* (*SIfVV* *vp1* *vp2*)]]

definition *do_exists* :: '*n* \Rightarrow ('*n*, '*d*::{default,linorder}) proof + ('*d*, ('*n*, '*d*) proof) part \Rightarrow ('*n*, '*d*) proof list **where**
do_exists *x* *p_part* = (case *p_part* of
Inl *p* \Rightarrow (case *p* of
Inl *sp* \Rightarrow [*Inl* (*SExists* *x* default *sp*)]
| *Inr* *vp* \Rightarrow [*Inr* (*VExists* *x* (trivial_part *vp*)]]
| *Inr* *part* \Rightarrow (if ($\exists x \in \text{Vals part. isl } x$) then
map ($\lambda(D,p). \text{map_sum } (SExists \ x \ (Min \ D)) \ id \ p$) (filter ($\lambda(_, p). \text{isl } p$) (subsvals 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 **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 ($\forall x \in \text{Vals part. isl } x$) then
[*Inl* (*SForall* *x* (map_part projl part))]
else
map ($\lambda(D,p). \text{map_sum } id \ (VForall \ x \ (Min \ D)) \ p$) (filter ($\lambda(_, p). \neg \text{isl } p$) (subsvals part)))))

definition *do_prev* :: *nat* \Rightarrow \mathcal{I} \Rightarrow *nat* \Rightarrow ('*n*, '*d*) proof \Rightarrow ('*n*, '*d*) proof list **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* :: *nat* \Rightarrow \mathcal{I} \Rightarrow *nat* \Rightarrow ('*n*, '*d*) proof \Rightarrow ('*n*, '*d*) 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) ⇒ (if mem t I then [Inr (VNext vp)] else [Inr (VNext vp), Inr (VNextOutR i)])

definition do_once_base :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_once_base i a p' = (case (p', a = 0) of
 (Inl sp', True) ⇒ [Inl (SONce i sp')]
 | (Inr vp', True) ⇒ [Inr (VOnce i i [vp'])]
 | (_ , False) ⇒ [Inr (VOnce i i [])])

definition do_once :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_once i a p p' = (case (p, a = 0, p') of
 (Inl sp, True, Inr _) ⇒ [Inl (SONce i sp)]
 | (Inl sp, True, Inl (SONce _ sp')) ⇒ [Inl (SONce i sp'), Inl (SONce i sp)]
 | (Inl _ , False, Inl (SONce _ sp')) ⇒ [Inl (SONce i sp')]
 | (Inl _ , False, Inr (VOnce _ li vps')) ⇒ [Inr (VOnce i li vps')]
 | (Inr _ , True, Inl (SONce _ sp')) ⇒ [Inl (SONce i sp')]
 | (Inr vp, True, Inr vp') ⇒ [(Inr vp') ⊕ (Inr vp)]
 | (Inr _ , False, Inl (SONce _ sp')) ⇒ [Inl (SONce i sp')]
 | (Inr _ , False, Inr (VOnce _ li vps')) ⇒ [Inr (VOnce i li vps')])

definition do_eventually_base :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_eventually_base i a p' = (case (p', a = 0) of
 (Inl sp', True) ⇒ [Inl (SEventually i sp')]
 | (Inr vp', True) ⇒ [Inr (VEventually i i [vp'])]
 | (_ , False) ⇒ [Inr (VEventually i i [])])

definition do_eventually :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_eventually i a p p' = (case (p, a = 0, p') of
 (Inl sp, True, Inr _) ⇒ [Inl (SEventually i sp)]
 | (Inl sp, True, Inl (SEventually _ sp')) ⇒ [Inl (SEventually i sp'), Inl (SEventually i sp)]
 | (Inl _ , False, Inl (SEventually _ sp')) ⇒ [Inl (SEventually i sp')]
 | (Inl _ , False, Inr (VEventually _ hi vps')) ⇒ [Inr (VEventually i hi vps')]
 | (Inr _ , True, Inl (SEventually _ sp')) ⇒ [Inl (SEventually i sp')]
 | (Inr vp, True, Inr vp') ⇒ [(Inr vp') ⊕ (Inr vp)]
 | (Inr _ , False, Inl (SEventually _ sp')) ⇒ [Inl (SEventually i sp')]
 | (Inr _ , False, Inr (VEventually _ hi vps')) ⇒ [Inr (VEventually i hi vps')])

definition do_historically_base :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_historically_base i a p' = (case (p', a = 0) of
 (Inl sp', True) ⇒ [Inl (SHistorically i i [sp'])]
 | (Inr vp', True) ⇒ [Inr (VHistorically i vp')]
 | (_ , False) ⇒ [Inl (SHistorically i i [])])

definition do_historically :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_historically i a p p' = (case (p, a = 0, p') of
 (Inl _ , True, Inr (VHistorically _ vp')) ⇒ [Inr (VHistorically i vp')]
 | (Inl sp, True, Inl sp') ⇒ [(Inl sp') ⊕ (Inl sp)]
 | (Inl _ , False, Inl (SHistorically _ li sps')) ⇒ [Inl (SHistorically i li sps')]
 | (Inl _ , False, Inr (VHistorically _ vp')) ⇒ [Inr (VHistorically i vp')]
 | (Inr vp, True, Inl _) ⇒ [Inr (VHistorically i vp)]
 | (Inr vp, True, Inr (VHistorically _ vp')) ⇒ [Inr (VHistorically i vp), Inr (VHistorically i vp')]
 | (Inr _ , False, Inl (SHistorically _ li sps')) ⇒ [Inl (SHistorically i li sps')]
 | (Inr _ , False, Inr (VHistorically _ vp')) ⇒ [Inr (VHistorically i vp')])

definition do_always_base :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**

do_always_base i a p' = (case (p', a = 0) of
 (Inl sp', True) ⇒ [Inl (SAlways i i [sp'])]
 | (Inr vp', True) ⇒ [Inr (VAlways i vp')]
 | (_ , False) ⇒ [Inl (SAlways i i [])])

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

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

definition $do_since :: nat \Rightarrow nat \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof list$ **where**

$do_since\ i\ a\ p1\ p2\ p' = (case\ (p1,\ p2,\ a = 0,\ p')\ of$
 $(Inl\ sp1,\ Inr\ _ ,\ True,\ Inl\ sp') \Rightarrow [(Inl\ sp') \oplus (Inl\ sp1)]$
 $| (Inl\ sp1,\ _ ,\ False,\ Inl\ sp') \Rightarrow [(Inl\ sp') \oplus (Inl\ sp1)]$
 $| (Inl\ sp1,\ Inl\ sp2,\ True,\ Inl\ sp') \Rightarrow [(Inl\ sp') \oplus (Inl\ sp1),\ Inl\ (SSince\ sp2\ \ \ \)]$
 $| (Inl\ _ ,\ Inr\ vp2,\ True,\ Inr\ (VSinceInf\ _ _ _)) \Rightarrow [p' \oplus (Inr\ vp2)]$
 $| (Inl\ _ ,\ _ ,\ False,\ Inr\ (VSinceInf\ _ li\ vp2s')) \Rightarrow [Inr\ (VSinceInf\ i\ li\ vp2s')]$
 $| (Inl\ _ ,\ Inr\ vp2,\ True,\ Inr\ (VSince\ _ _ _)) \Rightarrow [p' \oplus (Inr\ vp2)]$
 $| (Inl\ _ ,\ _ ,\ False,\ Inr\ (VSince\ _ vp1'\ vp2s')) \Rightarrow [Inr\ (VSince\ i\ vp1'\ vp2s')]$
 $| (Inr\ vp1,\ Inr\ vp2,\ True,\ Inl\ _) \Rightarrow [Inr\ (VSince\ i\ vp1\ [vp2])]$
 $| (Inr\ vp1,\ _ ,\ False,\ Inl\ _) \Rightarrow [Inr\ (VSince\ i\ vp1\ \ \ \)]$
 $| (Inr\ _ ,\ Inl\ sp2,\ True,\ Inl\ _) \Rightarrow [Inl\ (SSince\ sp2\ \ \ \)]$
 $| (Inr\ vp1,\ Inr\ vp2,\ True,\ Inr\ (VSinceInf\ _ _ _)) \Rightarrow [Inr\ (VSince\ i\ vp1\ [vp2]),\ p' \oplus (Inr\ vp2)]$
 $| (Inr\ vp1,\ _ ,\ False,\ Inr\ (VSinceInf\ _ li\ vp2s')) \Rightarrow [Inr\ (VSince\ i\ vp1\ \ \ \),\ Inr\ (VSinceInf\ i\ li\ vp2s')]$
 $| (\ _ ,\ Inl\ sp2,\ True,\ Inr\ (VSinceInf\ _ _ _)) \Rightarrow [Inl\ (SSince\ sp2\ \ \ \)]$
 $| (Inr\ vp1,\ Inr\ vp2,\ True,\ Inr\ (VSince\ _ _ _)) \Rightarrow [Inr\ (VSince\ i\ vp1\ [vp2]),\ p' \oplus (Inr\ vp2)]$
 $| (Inr\ vp1,\ _ ,\ False,\ Inr\ (VSince\ _ vp1'\ vp2s')) \Rightarrow [Inr\ (VSince\ i\ vp1\ \ \ \),\ Inr\ (VSince\ i\ vp1'\ vp2s')]$
 $| (\ _ ,\ Inl\ vp2,\ True,\ Inr\ (VSince\ _ _ _)) \Rightarrow [Inl\ (SSince\ vp2\ \ \ \)]$

definition $do_until_base :: nat \Rightarrow nat \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof list$ **where**

$do_until_base\ i\ a\ p1\ p2 = (case\ (p1,\ p2,\ a = 0)\ of$
 $(\ _ ,\ Inl\ sp2,\ True) \Rightarrow [Inl\ (SUntil\ \ \ \ sp2)]$
 $| (Inl\ sp1,\ _ ,\ False) \Rightarrow [Inr\ (VUntilInf\ i\ i\ \ \ \)]$
 $| (Inl\ sp1,\ Inr\ vp2,\ True) \Rightarrow [Inr\ (VUntilInf\ i\ i\ [vp2])]$
 $| (Inr\ vp1,\ _ ,\ False) \Rightarrow [Inr\ (VUntil\ i\ \ \ \ vp1),\ Inr\ (VUntilInf\ i\ i\ \ \ \)]$
 $| (Inr\ vp1,\ Inr\ vp2,\ True) \Rightarrow [Inr\ (VUntil\ i\ [vp2]\ vp1),\ Inr\ (VUntilInf\ i\ i\ [vp2])]$

definition $do_until :: nat \Rightarrow nat \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof list$ **where**

$do_until\ i\ a\ p1\ p2\ p' = (case\ (p1,\ p2,\ a = 0,\ p')\ of$
 $(Inl\ sp1,\ Inr\ _ ,\ True,\ Inl\ (SUntil\ _ _)) \Rightarrow [p' \oplus (Inl\ sp1)]$
 $| (Inl\ sp1,\ _ ,\ False,\ Inl\ (SUntil\ _ _)) \Rightarrow [p' \oplus (Inl\ sp1)]$
 $| (Inl\ sp1,\ Inl\ sp2,\ True,\ Inl\ (SUntil\ _ _)) \Rightarrow [p' \oplus (Inl\ sp1),\ Inl\ (SUntil\ \ \ \ sp2)]$
 $| (Inl\ _ ,\ Inr\ vp2,\ True,\ Inr\ (VUntilInf\ _ _ _)) \Rightarrow [p' \oplus (Inr\ vp2)]$
 $| (Inl\ _ ,\ _ ,\ False,\ Inr\ (VUntilInf\ _ hi\ vp2s')) \Rightarrow [Inr\ (VUntilInf\ i\ hi\ vp2s')]$
 $| (Inl\ _ ,\ Inr\ vp2,\ True,\ Inr\ (VUntil\ _ _ _)) \Rightarrow [p' \oplus (Inr\ vp2)]$
 $| (Inl\ _ ,\ _ ,\ False,\ Inr\ (VUntil\ _ vp2s'\ vp1')) \Rightarrow [Inr\ (VUntil\ i\ vp2s'\ vp1')]$

```

| (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 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  $\sigma :: ('n, 'd :: \{default, linorder\}) \text{ trace and}$ 
    cmp :: ('n, 'd) proof  $\Rightarrow$  ('n, 'd) proof  $\Rightarrow$  bool

```

begin

```

function (sequential) eval :: 'n list  $\Rightarrow$  nat  $\Rightarrow$  ('n, 'd) Formula.formula  $\Rightarrow$  ('n, 'd) expl where
    eval vs i Formula.TT = Leaf (Inl (STT i))
| eval vs i Formula.FF = Leaf (Inr (VFF i))
| eval vs i (Eq_Const x c) = Node x (tabulate [c] ( $\lambda c$ . 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 ( $\lambda x$ .  $x \in$  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  $\geq$  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  $\geq$  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 + 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)))
| eval vs i (Formula.MatchP I r) = undefined
| eval vs i (Formula.MatchF I r) = undefined
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: $\tau \sigma$ (Suc i) \leq b + $\tau \sigma$ i \implies i < LTP σ (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

13 Examples

definition monitor :: (('n :: linorder \times 'd :: {default, linorder} list) set \times nat) list \Rightarrow ('n, 'd) formula \Rightarrow ('n, 'd) expl list **where**

```

  monitor  $\pi$   $\varphi$  = map ( $\lambda i$ . eval (trace_of_list  $\pi$ ) ( $\lambda p q$ . size p  $\leq$  size q) (sorted_list_of_set (fv  $\varphi$ )) i  $\varphi$ )
  [0 ..< length  $\pi$ ]

```

definition check :: (('n :: linorder \times 'd :: {default, linorder} list) set \times nat) list \Rightarrow ('n, 'd) formula \Rightarrow

bool where
check $\pi \varphi = \text{list_all } (\text{check_all } (\text{trace_of_list } \pi) \varphi) (\text{monitor } \pi \varphi)$

13.1 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*

13.2 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.