

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	3
1.3	Earliest and Latest Time-Points	5
2	Regular expressions	6
3	Metric First-Order Temporal Logic	8
3.1	Syntax	8
3.2	Semantics	10
4	Valued Partitions	13
4.1	<i>size</i> setup	13
4.2	Functions on Valued Partitions	14
5	Partitioned Decision Trees	15
6	Proof System	18
6.1	Soundness and Completeness	20
7	Proof Objects	21
8	Auxiliary Lemmas	23
9	Proof Checker	28
9.1	Checker Soundness	31
9.2	Executable Variant of the Checker	33
9.3	Latest Relevant Time-Point	37
9.4	Active Domain	38

9.5	Congruence Modulo Active Domain	38
9.6	Checker Completeness	39
9.7	Lifting the Checker to PDTs	40
10	Type of Events	41
10.1	Code Adaptation for 8-bit strings	41
10.2	Event Parameters	42
11	Code Generation	44
11.1	Type Class Instances	44
11.2	Progress	45
11.3	Trace	46
11.4	Auxiliary results	47
11.5	<i>v_check_exec</i> setup	49
11.6	ETP/LTP setup	51
11.7	Exported functions	52
12	Unverified Explanation-Producing Monitoring Algorithm	53
13	Examples	62
13.1	Infinite Domain	62
13.2	Finite Domain	63

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* $f\ n$)
 <proof>

lemma *sortedD*: *sorted* $s \Longrightarrow$ $s\ !!\ i \leq$ *stl* $s\ !!\ i$
 <proof>

lemma *sorted_sdrop*: *sorted* $s \Longrightarrow$ *sorted* (*sdrop* $i\ s$)
 <proof>

lemma *sorted_monoD*: *sorted* $s \Longrightarrow$ $i \leq j \Longrightarrow$ $s\ !!\ i \leq$ $s\ !!\ j$
 <proof>

lemma *sorted_stake*: *sorted* $s \Longrightarrow$ *sorted* (*stake* $i\ s$)
 <proof>

lemma *sorted_monoI*: $\forall i\ j. i \leq j \longrightarrow$ $s\ !!\ i \leq$ $s\ !!\ j \Longrightarrow$ *sorted* s
 <proof>

lemma *sorted_iff_mono*: *sorted* $s \longleftrightarrow$ $(\forall i\ j. i \leq j \longrightarrow$ $s\ !!\ i \leq$ $s\ !!\ j)$
 <proof>

lemma *sorted_iff_le_Suc*: *sorted* $s \longleftrightarrow$ $(\forall i. s\ !!\ i \leq$ $s\ !!\ \text{Suc}\ i)$
 <proof>

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

lemma *sincreasingI*: $(\bigwedge x. \exists i. x <$ $s\ !!\ i) \Longrightarrow$ *sincreasing* s

<proof>

lemma *sincreasing_grD*:

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

assumes *sincreasing s*

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

<proof>

lemma *sincreasing_siterate_nat[simp]*:

fixes $n :: \text{nat}$

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

shows *sincreasing (siterate f n)*

<proof>

lemma *sincreasing_stl*: *sincreasing s* \implies *sincreasing (stl s)* **for** $s :: 'a :: \text{semilattice_sup}$ *stream*

<proof>

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

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

<proof>

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

<proof>

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **is**

$\lambda s i. \text{fst } (s !! i)$ *<proof>*

lift_definition $\tau :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **is**

$\lambda s i. \text{snd } (s !! i)$ *<proof>*

lemma *stream_eq_iff*: $s = s' \iff (\forall n. s !! n = s' !! n)$

<proof>

lemma *trace_eqI*: $(\bigwedge i. \Gamma \sigma i = \Gamma \sigma' i) \implies (\bigwedge i. \tau \sigma i = \tau \sigma' i) \implies \sigma = \sigma'$

<proof>

lemma *τ _mono[simp]*: $i \leq j \implies \tau s i \leq \tau s j$

<proof>

lemma *ex_le_ τ* : $\exists j \geq i. x \leq \tau s j$

<proof>

lemma *le_ τ _less*: $\tau \sigma i \leq \tau \sigma j \implies j < i \implies \tau \sigma i = \tau \sigma j$

<proof>

lemma *less_ τ D*: $\tau \sigma i < \tau \sigma j \implies i < j$

<proof>

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

1.2 Finite Trace Prefixes

typedef *'a prefix* = $\{p :: ('a \text{ set} \times \text{nat}) \text{ list. sorted } (\text{map snd } p)\}$

<proof>

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))
<proof>

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

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

lift_definition *pnil* :: 'a prefix **is** [] <proof>

lift_definition *plen* :: 'a prefix ⇒ nat **is** length <proof>

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>

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

lift_definition *less_eq_prefix* :: 'a prefix ⇒ 'a prefix ⇒ bool **is**
λp q. ∃ r. q = p @ r <proof>

definition *less_prefix* :: 'a prefix ⇒ 'a prefix ⇒ bool **where**
less_prefix x y = (x ≤ y ∧ ¬ y ≤ x)

instance
<proof>

end

lemma *psnoc_inject[simp]*:
last_ts p ≤ snd x ⇒ *last_ts* q ≤ snd y ⇒ *psnoc* p x = *psnoc* q y ↔ (p = q ∧ x = y)
<proof>

lift_definition *prefix_of* :: 'a prefix ⇒ 'a trace ⇒ bool **is** λp s. stake (length p) s = p <proof>

lemma *prefix_of_pnil[simp]*: *prefix_of* *pnil* σ
<proof>

lemma *plen_pnil[simp]*: *plen* *pnil* = 0
<proof>

lemma *plen_mono*: π ≤ π' ⇒ *plen* π ≤ *plen* π'
<proof>

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc* p x) s ⇒ *last_ts* p ≤ snd x ⇒
(*prefix_of* p s ⇒ Γ s (*plen* p) = fst x ⇒ τ s (*plen* p) = snd x ⇒ P) ⇒ P
<proof>

lemma *le_pnil[simp]*: *pnil* ≤ π
<proof>

lift_definition *take_prefix* :: nat ⇒ 'a trace ⇒ 'a prefix **is** stake
<proof>

lemma *plen_take_prefix[simp]*: $\text{plen } (\text{take_prefix } i \ \sigma) = i$
 ⟨proof⟩

lemma *plen_psnoc[simp]*: $\text{last_ts } \pi \leq \text{snd } x \implies \text{plen } (\text{psnoc } \pi \ x) = \text{plen } \pi + 1$
 ⟨proof⟩

lemma *prefix_of_take_prefix[simp]*: $\text{prefix_of } (\text{take_prefix } i \ \sigma) \ \sigma$
 ⟨proof⟩

lift_definition *pdrop* :: $\text{nat} \Rightarrow 'a \ \text{prefix} \Rightarrow 'a \ \text{prefix} \ \text{is } \text{drop}$
 ⟨proof⟩

lemma *pdrop_0[simp]*: $\text{pdrop } 0 \ \pi = \pi$
 ⟨proof⟩

lemma *prefix_of_antimono*: $\pi \leq \pi' \implies \text{prefix_of } \pi' \ s \implies \text{prefix_of } \pi \ s$
 ⟨proof⟩

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

lemma *τ _prefix_conv*: $\text{prefix_of } p \ s \implies \text{prefix_of } p \ s' \implies i < \text{plen } p \implies \tau \ s \ i = \tau \ s' \ i$
 ⟨proof⟩

lemma *Γ _prefix_conv*: $\text{prefix_of } p \ s \implies \text{prefix_of } p \ s' \implies i < \text{plen } p \implies \Gamma \ s \ i = \Gamma \ s' \ i$
 ⟨proof⟩

lemma *sincreasing_sdrop*:
fixes $s :: ('a :: \text{semilattice_sup}) \ \text{stream}$
assumes *sincreasing* s
shows *sincreasing* ($\text{sdrop } n \ s$)
 ⟨proof⟩

lemma *sorted_shift*:
 $\text{sorted } (xs \ @- \ s) = (\text{sorted } xs \ \wedge \ \text{sorted } s \ \wedge \ (\forall x \in \text{set } xs. \ \forall y \in \text{sset } s. \ x \leq y))$
 ⟨proof⟩

lemma *sincreasing_shift*:
assumes *sincreasing* s
shows *sincreasing* ($xs \ @- \ s$)
 ⟨proof⟩

lift_definition *pts* :: $'a \ \text{prefix} \Rightarrow \text{nat } \text{list} \ \text{is } \text{map } \text{snd}$ ⟨proof⟩

lemma *pts_pmap_Γ[simp]*: $\text{pts } (\text{pmap_}\Gamma \ f \ \pi) = \text{pts } \pi$
 ⟨proof⟩

1.3 Earliest and Latest Time-Points

definition *ETP*:: $'a \ \text{trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{ETP } \sigma \ t = (\text{LEAST } i. \ \tau \ \sigma \ i \geq t)$

definition *LTP*:: $'a \ \text{trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{LTP } \sigma \ t = \text{Max } \{i. \ (\tau \ \sigma \ i) \leq t\}$

abbreviation $\delta \ \sigma \ i \ j \equiv (\tau \ \sigma \ i - \tau \ \sigma \ j)$

abbreviation $\text{ETP_p } \sigma \ i \ b \equiv \text{ETP } \sigma \ ((\tau \ \sigma \ i) - b)$

abbreviation $LTP_p \sigma i I \equiv \min i (LTP \sigma ((\tau \sigma i) - \text{left } I))$
abbreviation $ETP_f \sigma i I \equiv \max i (ETP \sigma ((\tau \sigma i) + \text{left } I))$
abbreviation $LTP_f \sigma i b \equiv LTP \sigma ((\tau \sigma i) + b)$

definition max_opt **where**

$\text{max_opt } a b = (\text{case } (a,b) \text{ of } (\text{Some } x, \text{Some } y) \Rightarrow \text{Some } (\text{max } x y) \mid _ \Rightarrow \text{None})$

definition $LTP_p_safe \sigma i I = (\text{if } \tau \sigma i - \text{left } I \geq \tau \sigma 0 \text{ then } LTP_p \sigma i I \text{ else } 0)$

lemma $i_ETP_tau: i \geq ETP \sigma n \longleftrightarrow \tau \sigma i \geq n$
 $\langle \text{proof} \rangle$

lemma $tau_LTP_k:$
assumes $\tau \sigma 0 \leq n$ $LTP \sigma n < k$
shows $\tau \sigma k > n$
 $\langle \text{proof} \rangle$

lemma $i_LTP_tau:$
assumes $n_asm: n \geq \tau \sigma 0$
shows $(i \leq LTP \sigma n \longleftrightarrow \tau \sigma i \leq n)$
 $\langle \text{proof} \rangle$

lemma $ETP_delta: i \geq ETP \sigma (\tau \sigma l + n) \implies \delta \sigma i l \geq n$
 $\langle \text{proof} \rangle$

lemma $ETP_ge: ETP \sigma (\tau \sigma l + n + 1) > l$
 $\langle \text{proof} \rangle$

lemma $i_le_LTPi: i \leq LTP \sigma (\tau \sigma i)$
 $\langle \text{proof} \rangle$

lemma $i_le_LTPi_add: i \leq LTP \sigma (\tau \sigma i + n)$
 $\langle \text{proof} \rangle$

lemma $i_le_LTPi_minus:$
assumes $\tau \sigma 0 + n \leq \tau \sigma i$ $i > 0$ $n > 0$
shows $LTP \sigma (\tau \sigma i - n) < i$
 $\langle \text{proof} \rangle$

lemma $i_ge_etpi: ETP \sigma (\tau \sigma i) \leq i$
 $\langle \text{proof} \rangle$

lemma $etp_0[\text{simp}]: ETP \sigma 0 = 0$
 $\langle \text{proof} \rangle$

2 Regular expressions

context **begin**

qualified datatype $(\text{atms: 'a}) \text{ regex} = \text{Skip nat} \mid \text{Test 'a}$
 $\mid \text{Plus 'a regex 'a regex} \mid \text{Times 'a regex 'a regex} \mid \text{Star 'a regex}$

lemma $\text{finite_atms}[\text{simp}]: \text{finite } (\text{atms } r)$
 $\langle \text{proof} \rangle$

definition $\text{Wild} = \text{Skip } 1$

lemma $\text{size_regex_estimation}[\text{termination_simp}]: x \in \text{atms } r \implies y < f x \implies y < \text{size_regex } f r$

$\langle \text{proof} \rangle$

lemma *size_regec_estimation*^[termination_simp]: $x \in \text{atms } r \implies y \leq f x \implies y \leq \text{size_regec } f r$

$\langle \text{proof} \rangle$ **definition** *TimesL* $r S = \text{Times } r \text{ ' } S$

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

qualified primrec *collect* **where**

$\text{collect } f (\text{Skip } n) = \{\}$

| $\text{collect } f (\text{Test } \varphi) = f \varphi$

| $\text{collect } f (\text{Plus } r s) = \text{collect } f r \cup \text{collect } f s$

| $\text{collect } f (\text{Times } r s) = \text{collect } f r \cup \text{collect } f s$

| $\text{collect } f (\text{Star } r) = \text{collect } f r$

lemma *collect_cong*^[fundef_cong]:

$r = r' \implies (\bigwedge z. z \in \text{atms } r \implies f z = f' z) \implies \text{collect } f r = \text{collect } f' r'$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *collect_commute*:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$ **definition** *ncollect* **where**

$\text{ncollect } f r = \text{Max } (\text{insert } 0 (\text{Suc ' collect } f r))$

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

$\langle \text{proof} \rangle$

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

$\text{ncollect } f (\text{Skip } n) = 0$

$\text{ncollect } f (\text{Test } \varphi) = \text{Max } (\text{insert } 0 (\text{Suc ' } f \varphi))$

$\text{ncollect } f (\text{Plus } r s) = \text{max } (\text{ncollect } f r) (\text{ncollect } f s)$

$\text{ncollect } f (\text{Times } r s) = \text{max } (\text{ncollect } f r) (\text{ncollect } f s)$

$\text{ncollect } f (\text{Star } r) = \text{ncollect } f r$

$\langle \text{proof} \rangle$

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

qualified primrec *match* $:: (\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ regec} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$\text{match test } (\text{Skip } n) = (\lambda i j. j = i + n)$

| $\text{match test } (\text{Test } \varphi) = (\lambda i j. i = j \wedge \text{test } i \varphi)$

| $\text{match test } (\text{Plus } r s) = \text{match test } r \sqcup \text{match test } s$

| $\text{match test } (\text{Times } r s) = \text{match test } r \text{ OO } \text{match test } s$

| $\text{match test } (\text{Star } r) = (\text{match test } r)^{**}$

lemma *match_cong*^[fundef_cong]:

$r = r' \implies (\bigwedge i z. z \in \text{atms } r \implies \text{test } i z = \text{test } i z) \implies \text{match } t r = \text{match } t' r'$

$\langle \text{proof} \rangle$

lemma *match_le*: $\text{match test } r i j \implies i \leq j$

$\langle \text{proof} \rangle$

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

<proof>

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

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>

lemma *match_cong_strong*:

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

end

3 Metric First-Order Temporal Logic

3.1 Syntax

type_synonym (n, a) *event* = $(n \times a)$ *list*

type_synonym (n, a) *database* = (n, a) *event set*

type_synonym (n, a) *prefix* = $(n \times a)$ *list prefix*

type_synonym (n, a) *trace* = $(n \times a)$ *list trace*

type_synonym (n, a) *env* = $n \Rightarrow a$

type_synonym (n, a) *envset* = $n \Rightarrow a$ *set*

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

lemma *in_fv_trm_conv*: $x \in fv_trm\ t \iff t = v\ x$
<proof>

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 $(r \dagger ts) = \bigcup (fv_trm\ 'a\ set\ ts)$

| *fv* $\top = \{\}$

$| \text{fv } \perp = \{\}$
 $| \text{fv } (x \approx c) = \{x\}$
 $| \text{fv } (\neg_F \varphi) = \text{fv } \varphi$
 $| \text{fv } (\varphi \vee_F \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\varphi \wedge_F \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\varphi \rightarrow_F \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\varphi \leftarrow_F \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\exists_F x. \varphi) = \text{fv } \varphi - \{x\}$
 $| \text{fv } (\forall_F x. \varphi) = \text{fv } \varphi - \{x\}$
 $| \text{fv } (\mathbf{Y} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\mathbf{X} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\mathbf{P} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\mathbf{H} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\mathbf{F} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\mathbf{G} I \varphi) = \text{fv } \varphi$
 $| \text{fv } (\varphi \mathbf{S} I \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\varphi \mathbf{U} I \psi) = \text{fv } \varphi \cup \text{fv } \psi$
 $| \text{fv } (\triangleleft I r) = \text{Regex.collect fv } r$
 $| \text{fv } (\triangleright I r) = \text{Regex.collect fv } r$

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)

$| \text{consts } \top = \{\}$
 $| \text{consts } \perp = \{\}$
 $| \text{consts } (x \approx c) = \{c\}$
 $| \text{consts } (\neg_F \varphi) = \text{consts } \varphi$
 $| \text{consts } (\varphi \vee_F \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\varphi \wedge_F \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\varphi \rightarrow_F \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\varphi \leftarrow_F \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\exists_F x. \varphi) = \text{consts } \varphi$
 $| \text{consts } (\forall_F x. \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{Y} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{X} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{P} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{H} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{F} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\mathbf{G} I \varphi) = \text{consts } \varphi$
 $| \text{consts } (\varphi \mathbf{S} I \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\varphi \mathbf{U} I \psi) = \text{consts } \varphi \cup \text{consts } \psi$
 $| \text{consts } (\triangleleft I r) = \text{Regex.collect consts } r$
 $| \text{consts } (\triangleright I r) = \text{Regex.collect consts } r$

lemma *finite_fv_trm[simp]*: finite (fv_trm t)
 <proof>

lemma *finite_fv[simp]*: finite (fv φ)
 <proof>

lemma *finite_consts[simp]*: finite (consts φ)
 <proof>

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

fun *future_bounded* :: ('n, 'a) formula \Rightarrow bool **where**
future_bounded $\top = \text{True}$

$| \text{future_bounded } \perp = \text{True}$
 $| \text{future_bounded } (_ \dagger _) = \text{True}$
 $| \text{future_bounded } (_ \approx _) = \text{True}$
 $| \text{future_bounded } (\neg_F \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\varphi \vee_F \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\varphi \wedge_F \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\varphi \rightarrow_F \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\varphi \leftrightarrow_F \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\exists_F _ . \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\forall_F _ . \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{Y} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{X} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{P} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{H} I \varphi) = \text{future_bounded } \varphi$
 $| \text{future_bounded } (\mathbf{F} I \varphi) = (\text{future_bounded } \varphi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\mathbf{G} I \varphi) = (\text{future_bounded } \varphi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\varphi \mathbf{S} I \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi)$
 $| \text{future_bounded } (\varphi \mathbf{U} I \psi) = (\text{future_bounded } \varphi \wedge \text{future_bounded } \psi \wedge \text{right } I \neq \infty)$
 $| \text{future_bounded } (\triangleleft I r) = \text{Regex.pred_regex future_bounded } r$
 $| \text{future_bounded } (\triangleright I r) = (\text{Regex.pred_regex future_bounded } r \wedge \text{right } I \neq \infty)$

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 $\text{eval_trm_fv_cong}: \forall x \in \text{fv_trm } t. v x = v' x \implies v[t] = v'[t]$
 $\langle \text{proof} \rangle$

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 $\text{eval_trms_fv_cong}: \forall t \in \text{set } ts. \forall x \in \text{fv_trm } t. v x = v' x \implies v[ts] = v'[ts]$
 $\langle \text{proof} \rangle$

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 $\text{eval_trms_set_Nil}: vs\{\}\{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma $\text{eval_trms_set_Cons}: vs\{t \# ts\}\{\} = vs\{t\}\{\} \# vs\{ts\}\{\}$
 $\langle \text{proof} \rangle$

fun $\text{sat} :: ('n, 'a) \text{trace} \Rightarrow ('n, 'a) \text{env} \Rightarrow \text{nat} \Rightarrow ('n, 'a) \text{formula} \Rightarrow \text{bool} (\langle _ _ _ \rangle \models _ [56, 56, 56, 56] 55)$ **where**
 $\langle \sigma, v, i \rangle \models \top = \text{True}$
 $\langle \sigma, v, i \rangle \models \perp = \text{False}$

$\mid \langle \sigma, v, i \rangle \models r \dagger ts = ((r, v[[ts]]) \in \Gamma \sigma i)$
 $\mid \langle \sigma, v, i \rangle \models x \approx c = (v x = c)$
 $\mid \langle \sigma, v, i \rangle \models \neg_F \varphi = (\neg \langle \sigma, v, i \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \varphi \vee_F \psi = (\langle \sigma, v, i \rangle \models \varphi \vee \langle \sigma, v, i \rangle \models \psi)$
 $\mid \langle \sigma, v, i \rangle \models \varphi \wedge_F \psi = (\langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i \rangle \models \psi)$
 $\mid \langle \sigma, v, i \rangle \models \varphi \rightarrow_F \psi = (\langle \sigma, v, i \rangle \models \varphi \rightarrow \langle \sigma, v, i \rangle \models \psi)$
 $\mid \langle \sigma, v, i \rangle \models \varphi \leftarrow_F \psi = (\langle \sigma, v, i \rangle \models \varphi \leftarrow \langle \sigma, v, i \rangle \models \psi)$
 $\mid \langle \sigma, v, i \rangle \models \exists_F x. \varphi = (\exists z. \langle \sigma, v(x := z), i \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \forall_F x. \varphi = (\forall z. \langle \sigma, v(x := z), i \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{Y} I \varphi = (\text{case } i \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } j \Rightarrow \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{X} I \varphi = (\text{mem } (\tau \sigma (\text{Suc } i) - \tau \sigma i) I \wedge \langle \sigma, v, \text{Suc } i \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{P} I \varphi = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{H} I \varphi = (\forall j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \rightarrow \langle \sigma, v, j \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{F} I \varphi = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \langle \sigma, v, j \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \mathbf{G} I \varphi = (\forall j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \rightarrow \langle \sigma, v, j \rangle \models \varphi)$
 $\mid \langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{j..i\}. \langle \sigma, v, k \rangle \models \varphi))$
 $\mid \langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \langle \sigma, v, j \rangle \models \psi \wedge (\forall k \in \{i..j\}. \langle \sigma, v, k \rangle \models \varphi))$
 $\mid \langle \sigma, v, i \rangle \models \triangleleft I r = (\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge \text{Regex.match } (\lambda k \varphi. \langle \sigma, v, k \rangle \models \varphi) r j i)$
 $\mid \langle \sigma, v, i \rangle \models \triangleright I r = (\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \text{Regex.match } (\lambda k \varphi. \langle \sigma, v, k \rangle \models \varphi) r i j)$

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>

lemma *sat_Until_rec*: $\langle \sigma, v, i \rangle \models \varphi \mathbf{U} I \psi \longleftrightarrow$
 $(\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \psi \vee$
 $\Delta \sigma (i + 1) \leq \text{right } I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i + 1 \rangle \models \varphi \mathbf{U} (\text{subtract } (\Delta \sigma (i + 1)) I) \psi)$
 $(\text{is } ?L \longleftrightarrow ?R)$
<proof>

lemma *sat_Since_rec*: $\langle \sigma, v, i \rangle \models \varphi \mathbf{S} I \psi \longleftrightarrow$
 $\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \psi \vee$
 $(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \langle \sigma, v, i \rangle \models \varphi \wedge \langle \sigma, v, i - 1 \rangle \models \varphi \mathbf{S} (\text{subtract } (\Delta \sigma i) I) \psi)$
 $(\text{is } ?L \longleftrightarrow ?R)$
<proof>

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

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

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

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

lemma *sat_Once_rec*: $\langle \sigma, v, i \rangle \models \mathbf{P} I \varphi \longleftrightarrow$
 $\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$
 $(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \langle \sigma, v, i - 1 \rangle \models \mathbf{P} (\text{subtract } (\Delta \sigma i) I) \varphi)$
<proof>

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

lemma *sat_Historically_rec*: $\langle \sigma, v, i \rangle \models \mathbf{H} I \varphi \longleftrightarrow$
 $(\text{mem } 0 I \rightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$

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

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

lemma *sat_Eventually_rec*: $\langle \sigma, v, i \rangle \models \mathbf{F} I \varphi \longleftrightarrow$
 $\text{mem } 0 I \wedge \langle \sigma, v, i \rangle \models \varphi \vee$
 $(\Delta \sigma (i + 1) \leq \text{right } I \wedge \langle \sigma, v, i + 1 \rangle \models \mathbf{F} (\text{subtract } (\Delta \sigma (i + 1)) I) \varphi)$
 $\langle \text{proof} \rangle$

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

lemma *sat_Always_rec*: $\langle \sigma, v, i \rangle \models \mathbf{G} I \varphi \longleftrightarrow$
 $(\text{mem } 0 I \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$
 $(\Delta \sigma (i + 1) \leq \text{right } I \longrightarrow \langle \sigma, v, i + 1 \rangle \models \mathbf{G} (\text{subtract } (\Delta \sigma (i + 1)) I) \varphi)$
 $\langle \text{proof} \rangle$

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

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>

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>

lemma *part_list_eq*:

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

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

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

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

<proof>

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

typedef $('d, 'a) \text{ part} = \{ xs :: ('d \text{ set } \times 'a) \text{ list. } \text{partition_on } \text{UNIV } (\text{set } (\text{map } \text{fst } xs)) \wedge \text{distinct } (\text{map } \text{fst } xs) \}$

<proof>

setup_lifting *type_definition_part*

lift_bnf $(\text{no_warn_wits}, \text{no_warn_transfer}) (\text{dead } 'd, \text{Vals: } 'a) \text{ part}$

<proof>

4.1 size setup

lift_definition *subs* :: $('d, 'a) \text{ part} \Rightarrow 'd \text{ set list is map fst } \langle \text{proof} \rangle$

lift_definition *Subs* :: $('d, 'a) \text{ part} \Rightarrow 'd \text{ set set is set o map fst } \langle \text{proof} \rangle$

lift_definition *vals* :: $('d, 'a) \text{ part} \Rightarrow 'a \text{ list is map snd } \langle \text{proof} \rangle$

lift_definition *SubsVals* :: $('d, 'a) \text{ part} \Rightarrow ('d \text{ set } \times 'a) \text{ set is set } \langle \text{proof} \rangle$

lift_definition *subsvals* :: $('d, 'a) \text{ part} \Rightarrow ('d \text{ set } \times 'a) \text{ list is id } \langle \text{proof} \rangle$

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

instantiation *part* :: $(\text{type}, \text{type}) \text{ size begin}$

definition *size_part where*

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

instance *<proof>*

end

lemma *size_part_overloaded_simps*[simp]: $\text{size } x = \text{size } (\text{vals } x)$
 ⟨proof⟩

lemma *part_size_o_map*: $\text{inj } h \implies \text{size_part } f \, g \circ \text{map_part } h = \text{size_part } f \, (g \circ h)$
 ⟨proof⟩

⟨ML⟩

lemma *is_measure_size_part*[measure_function]: $\text{is_measure } f \implies \text{is_measure } g \implies \text{is_measure } (\text{size_part } f \, g)$
 ⟨proof⟩

lemma *size_part_estimation*[termination_simp]: $x \in \text{Vals } xs \implies y < g \, x \implies y < \text{size_part } f \, g \, xs$
 ⟨proof⟩

lemma *size_part_estimation'*[termination_simp]: $x \in \text{Vals } xs \implies y \leq g \, x \implies y \leq \text{size_part } f \, g \, xs$
 ⟨proof⟩

lemma *size_part_pointwise*[termination_simp]: $(\bigwedge x. x \in \text{Vals } xs \implies f \, x \leq g \, x) \implies \text{size_part } h \, f \, xs \leq \text{size_part } h \, g \, xs$
 ⟨proof⟩

4.2 Functions on Valued Partitions

lemma *Vals_code*[code]: $\text{Vals } x = \text{set } (\text{map } \text{snd } (\text{Rep_part } x))$
 ⟨proof⟩

lemma *Vals_transfer*[transfer_rule]: $\text{rel_fun } (\text{pcr_part } (=) (=)) (=) (\text{set } \circ \text{map } \text{snd}) \text{ Vals}$
 ⟨proof⟩

lemma *set_vals*[simp]: $\text{set } (\text{vals } xs) = \text{Vals } xs$
 ⟨proof⟩

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

lift_definition *tabulate* :: $'d \text{ list} \Rightarrow ('d \Rightarrow 'n) \Rightarrow 'n \Rightarrow ('d, 'n) \text{ part is}$
 $\lambda ds \, f \, z. \text{if distinct } ds \text{ then if set } ds = \text{UNIV then map } (\lambda d. (\{d\}, f \, d)) \, ds \text{ else } (- \text{set } ds, z) \# \text{map } (\lambda d. (\{d\}, f \, d)) \, ds \text{ else } [(UNIV, z)]$
 ⟨proof⟩

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

lemma *Vals_tabulate*[simp]: $\text{Vals } (\text{tabulate } xs \, f \, z) =$
 $(\text{if distinct } xs \text{ then if set } xs = \text{UNIV then } f \, ' \text{set } xs \text{ else } \{z\} \cup f \, ' \text{set } xs \text{ else } \{z\})$
 ⟨proof⟩

lemma *lookup_part_tabulate*[simp]: $\text{lookup_part } (\text{tabulate } xs \, f \, z) \, x =$
 $(\text{if distinct } xs \wedge x \in \text{set } xs \text{ then } f \, x \text{ else } z)$
 ⟨proof⟩

lemma *part_hd_Vals*[simp]: $\text{part_hd } \text{part} \in \text{Vals } \text{part}$
 ⟨proof⟩

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

lemma *lookup_part_SubVals*: $\exists D. d \in D \wedge (D, \text{lookup_part } \text{part } d) \in \text{SubsVals } \text{part}$
 <proof>

lemma *lookup_part_from_subvals*: $(D, e) \in \text{set } (\text{subsvals } \text{part}) \implies d \in D \implies \text{lookup_part } \text{part } d = e$
 <proof>

lemma *size_lookup_part_estimation*[*termination_simp*]: $\text{size } (\text{lookup_part } \text{part } d) < \text{Suc } (\text{size_part } (\lambda_. 0) \text{ size } \text{part})$
 <proof>

lemma *subsvals_part_estimation*[*termination_simp*]: $(D, e) \in \text{set } (\text{subsvals } \text{part}) \implies \text{size } e < \text{Suc } (\text{size_part } (\lambda_. 0) \text{ size } \text{part})$
 <proof>

lemma *size_part_hd_estimation*[*termination_simp*]: $\text{size } (\text{part_hd } \text{part}) < \text{Suc } (\text{size_part } (\lambda_. 0) \text{ size } \text{part})$
 <proof>

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

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

lemma *snd_lookup*[*simp*]: $\text{snd } (\text{lookup } \text{part } d) = \text{lookup_part } \text{part } d$
 <proof>

lemma *distinct_disjoint_uniq*: $\text{distinct } xs \implies \text{disjoint } (\text{set } xs) \implies i < j \implies j < \text{length } xs \implies d \in xs ! i \implies d \in xs ! j \implies \text{False}$
 <proof>

lemma *partition_on_UNIV_find_Some*:
 $\text{partition_on } \text{UNIV } (\text{set } (\text{map } \text{fst } \text{part})) \implies \text{distinct } (\text{map } \text{fst } \text{part}) \implies \exists y. \text{find } (\lambda(D, _). d \in D) \ \text{part} = \text{Some } y$
 <proof>

lemma *fst_lookup*: $d \in \text{fst } (\text{lookup } \text{part } d)$
 <proof>

lemma *lookup_subsvals*: $\text{lookup } \text{part } d \in \text{set } (\text{subsvals } \text{part})$
 <proof>

lift_definition *trivial_part* :: $'pt \Rightarrow ('d, 'pt) \text{ part} \text{ is } \lambda pt. [(UNIV, pt)]$
 <proof>

lemma *part_hd_trivial*[*simp*]: $\text{part_hd } (\text{trivial_part } pt) = pt$
 <proof>

lemma *SubsVals_trivial*[*simp*]: $\text{SubsVals } (\text{trivial_part } pt) = \{(UNIV, pt)\}$
 <proof>

5 Partitioned Decision Trees

datatype $(\text{dead } 'd, \text{leaves: } 'l, \text{vars: } 'n) \text{ pdt} = \text{Leaf } (\text{unleaf: } 'l) \mid \text{Node } 'n \ ('d, ('d, 'l, 'n) \text{ pdt}) \ \text{part}$

inductive *vars_order* :: $'n \text{ list} \Rightarrow ('d, 'l, 'n) \text{ pdt} \Rightarrow \text{bool}$ **where**
 $\text{vars_order } vs \ (\text{Leaf } _)$
 $\mid \forall \text{expl} \in \text{Vals } \text{part1}. \text{vars_order } vs \ \text{expl} \implies \text{vars_order } (x \# vs) \ (\text{Node } x \ \text{part1})$

| $\text{vars_order } vs \text{ (Node } x \text{ part1)} \implies x \neq z \implies \text{vars_order } (z \# vs) \text{ (Node } x \text{ part1)}$

lemma vars_order_Node :

assumes $\text{distinct } xs$

shows $\text{vars_order } xs \text{ (Node } x \text{ part)} = (\exists ys \ zs. xs = ys @ x \# zs \wedge (\forall e \in \text{Vals part. vars_order } zs \ e))$

$\langle \text{proof} \rangle$

fun distinct_paths **where**

$\text{distinct_paths (Leaf } _ \text{)} = \text{True}$

| $\text{distinct_paths (Node } x \text{ part)} = (\forall e \in \text{Vals part. } x \notin \text{vars } e \wedge \text{distinct_paths } e)$

fun eval_pdt **where**

$\text{eval_pdt } v \text{ (Leaf } l \text{)} = l$

| $\text{eval_pdt } v \text{ (Node } x \text{ part)} = \text{eval_pdt } v \text{ (lookup_part part (v } x \text{))}$

lemma eval_pdt_cong : $\forall x \in \text{vars } e. v \ x = v' \ x \implies \text{eval_pdt } v \ e = \text{eval_pdt } v' \ e$

$\langle \text{proof} \rangle$

lemma vars_order_vars : $\text{vars_order } vs \ e \implies \text{vars } e \subseteq \text{set } vs$

$\langle \text{proof} \rangle$

lemma $\text{vars_order_distinct_paths}$: $\text{vars_order } vs \ e \implies \text{distinct } vs \implies \text{distinct_paths } e$

$\langle \text{proof} \rangle$

lemma eval_pdt_fun_upd : $\text{vars_order } vs \ e \implies x \notin \text{set } vs \implies \text{eval_pdt } (v(x := d)) \ e = \text{eval_pdt } v \ e$

$\langle \text{proof} \rangle$

context **begin**

qualified inductive

$\text{SAT} :: (\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ \text{Regex.regex} \Rightarrow \text{bool}$

for sat **where**

$\text{STest}: i = j \implies \text{sat } i \ x \implies \text{SAT } sat \ i \ j \ (\text{Regex.Test } x)$

| $\text{SSkip}: j = i + n \implies \text{SAT } sat \ i \ j \ (\text{Regex.Skip } n)$

| $\text{SPlusL}: \text{SAT } sat \ i \ j \ r \implies \text{SAT } sat \ i \ j \ (\text{Regex.Plus } r \ s)$

| $\text{SPlusR}: \text{SAT } sat \ i \ j \ s \implies \text{SAT } sat \ i \ j \ (\text{Regex.Plus } r \ s)$

| $\text{STimes}: \text{SAT } sat \ i \ k \ r \implies \text{SAT } sat \ k \ j \ s \implies \text{SAT } sat \ i \ j \ (\text{Regex.Times } r \ s)$

| $\text{SStar_eps}: i = j \implies \text{SAT } sat \ i \ j \ (\text{Regex.Star } r)$

| $\text{SStar}: i < j \implies (\exists zs. xs = i \# zs @ [j]) \implies$

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

$\forall k \in \{0 \ .. < \text{length } xs - 1\}. \text{SAT } sat \ (xs ! k) \ (xs ! (\text{Suc } k)) \ r \implies$

$\text{SAT } sat \ i \ j \ (\text{Regex.Star } r)$

lemma SAT_mono [mono]:

assumes $X \leq Y$

shows $\text{SAT } X \leq \text{SAT } Y$

$\langle \text{proof} \rangle$

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

qualified inductive

$\text{VIO} :: (\text{nat} \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \ \text{Regex.regex} \Rightarrow \text{bool}$

for vio **where**

$\text{VSkip}: j \neq i + n \implies \text{VIO } vio \ i \ j \ (\text{Regex.Skip } n)$

| $\text{VTest}: i = j \implies \text{vio } i \ x \implies \text{VIO } vio \ i \ j \ (\text{Regex.Test } x)$

| $\text{VTest_neg}: i \neq j \implies \text{VIO } vio \ i \ j \ (\text{Regex.Test } x)$

| $\text{VPlus}: \text{VIO } vio \ i \ j \ r \implies \text{VIO } vio \ i \ j \ s \implies \text{VIO } vio \ i \ j \ (\text{Regex.Plus } r \ s)$

| $\text{VTimes}: \forall k \in \{i \ .. j\}. \text{VIO } vio \ i \ k \ r \vee \text{VIO } vio \ k \ j \ s \implies \text{VIO } vio \ i \ j \ (\text{Regex.Times } r \ s)$

| *VStar*: $i < j \implies i \in S \implies j \in T \implies S \cup T = \{i .. j\} \implies S \cap T = \{\} \implies$
 $\forall (s, t) \in \text{rm } (S \times T). \text{VIO } \text{vio } s \ t \ r \implies \text{VIO } \text{vio } i \ j \ (\text{Regex.Star } r)$
| *VStar_gt*: $i > j \implies \text{VIO } \text{vio } i \ j \ (\text{Regex.Star } r)$

lemma *VIO_mono*[*mono*]:
assumes $X \leq Y$
shows $\text{VIO } X \leq \text{VIO } Y$
<proof>

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

lemma
chain_Nil[*simp*]: $\neg \text{chain } R \ []$ **and**
chain_not_Nil: *chain* *R* xs \implies xs \neq []
<proof>

lemma *chain_rtranclp*: *chain* *R* xs \implies *R*** (hd xs) (last xs)
<proof>

lemma *chain_append*:
assumes *chain* *R* xs *chain* *R* ys *R* (last xs) (hd ys)
shows *chain* *R* (xs @ ys)
<proof>

lemma *tranclp_imp_exists_finite_chain_list*:
 $R^{++} \ x \ y \implies \exists \text{xs. } \text{chain } R \ (x \ \# \ \text{xs} \ @ \ [y])$
<proof>

lemma *chain_pairwise*:
chain *R* xs \implies Suc *i* < length xs \implies *R* (xs ! *i*) (xs ! (Suc *i*))
<proof>

lemma *chain_sorted_remdups*:
chain *R* xs \implies ($\bigwedge x \ y. \ R \ x \ y \implies x \leq y$) \implies sorted xs \wedge *chain* *R* (remdups xs)
<proof>

lemma *sorted_remdups*: sorted xs \implies sorted_wrt (<) (remdups xs)
<proof>

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]
<proof>

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]
 $\forall k \in \{0 ..< \text{length } xs - 1\}. \text{xs} \ ! \ k < \text{xs} \ ! \ (\text{Suc } k) \ \wedge \ R \ (\text{xs} \ ! \ k) \ (\text{xs} \ ! \ (\text{Suc } k))$
<proof>

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*:

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

lemma *match_rel_list_to_tranclp*:

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

lemma *completeness_SAT*:

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

lemma *completeness_VIO*:

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

lemma *soundness_SAT*:

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

lemma *soundness_VIO*:

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

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*: $(r, \text{eval_trms } v \ ts) \in \Gamma \ \sigma \ i \implies SAT \ \sigma \ v \ i \ (\text{Pred } r \ ts)$
 | *VPred*: $(r, \text{eval_trms } v \ ts) \notin \Gamma \ \sigma \ i \implies VIO \ \sigma \ v \ i \ (\text{Pred } r \ ts)$
 | *SEq_Const*: $v \ x = c \implies SAT \ \sigma \ v \ i \ (\text{Eq_Const } x \ c)$
 | *VEq_Const*: $v \ x \neq c \implies VIO \ \sigma \ v \ i \ (\text{Eq_Const } x \ c)$
 | *SNeg*: $VIO \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ (\text{Neg } \varphi)$
 | *VNeg*: $SAT \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ (\text{Neg } \varphi)$
 | *SOrL*: $SAT \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ (\text{Or } \varphi \ \psi)$
 | *SOrR*: $SAT \ \sigma \ v \ i \ \psi \implies SAT \ \sigma \ v \ i \ (\text{Or } \varphi \ \psi)$
 | *VOr*: $VIO \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ \psi \implies VIO \ \sigma \ v \ i \ (\text{Or } \varphi \ \psi)$
 | *SAnd*: $SAT \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ \psi \implies SAT \ \sigma \ v \ i \ (\text{And } \varphi \ \psi)$
 | *VAndL*: $VIO \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ (\text{And } \varphi \ \psi)$
 | *VAndR*: $VIO \ \sigma \ v \ i \ \psi \implies VIO \ \sigma \ v \ i \ (\text{And } \varphi \ \psi)$
 | *SImpl*: $SAT \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ (\text{Imp } \varphi \ \psi)$
 | *SImpR*: $SAT \ \sigma \ v \ i \ \psi \implies SAT \ \sigma \ v \ i \ (\text{Imp } \varphi \ \psi)$
 | *VImp*: $SAT \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ \psi \implies VIO \ \sigma \ v \ i \ (\text{Imp } \varphi \ \psi)$
 | *SIffSS*: $SAT \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ \psi \implies SAT \ \sigma \ v \ i \ (\text{Iff } \varphi \ \psi)$
 | *SIffVV*: $VIO \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ \psi \implies SAT \ \sigma \ v \ i \ (\text{Iff } \varphi \ \psi)$
 | *VIffSV*: $SAT \ \sigma \ v \ i \ \varphi \implies VIO \ \sigma \ v \ i \ \psi \implies VIO \ \sigma \ v \ i \ (\text{Iff } \varphi \ \psi)$
 | *VIffVS*: $VIO \ \sigma \ v \ i \ \varphi \implies SAT \ \sigma \ v \ i \ \psi \implies VIO \ \sigma \ v \ i \ (\text{Iff } \varphi \ \psi)$
 | *SExists*: $\exists z. SAT \ \sigma \ (v \ (x := z)) \ i \ \varphi \implies SAT \ \sigma \ v \ i \ (\text{Exists } x \ \varphi)$
 | *VExists*: $\forall z. VIO \ \sigma \ (v \ (x := z)) \ i \ \varphi \implies VIO \ \sigma \ v \ i \ (\text{Exists } x \ \varphi)$
 | *SForall*: $\forall z. SAT \ \sigma \ (v \ (x := z)) \ i \ \varphi \implies SAT \ \sigma \ v \ i \ (\text{Forall } x \ \varphi)$

$| VForall: \exists z. VIO \sigma (v (x := z)) i \varphi \implies VIO \sigma v i (Forall x \varphi)$
 $| SPrev: i > 0 \implies mem (\Delta \sigma i) I \implies SAT \sigma v (i-1) \varphi \implies SAT \sigma v i (\mathbf{Y} I \varphi)$
 $| VPrev: i > 0 \implies VIO \sigma v (i-1) \varphi \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $| VPrevZ: i = 0 \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $| VPrevOutL: i > 0 \implies (\Delta \sigma i) < (left I) \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $| VPrevOutR: i > 0 \implies enat (\Delta \sigma i) > (right I) \implies VIO \sigma v i (\mathbf{Y} I \varphi)$
 $| SNext: mem (\Delta \sigma (i+1)) I \implies SAT \sigma v (i+1) \varphi \implies SAT \sigma v i (\mathbf{X} I \varphi)$
 $| VNext: VIO \sigma v (i+1) \varphi \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $| VNextOutL: (\Delta \sigma (i+1)) < (left I) \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $| VNextOutR: enat (\Delta \sigma (i+1)) > (right I) \implies VIO \sigma v i (\mathbf{X} I \varphi)$
 $| SOnce: j \leq i \implies mem (\delta \sigma i j) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{P} I \varphi)$
 $| VOnceOut: \tau \sigma i < \tau \sigma 0 + left I \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $| VOnce: j = (case right I of \infty \Rightarrow 0$
 $\quad | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + left I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \varphi) \implies VIO \sigma v i (\mathbf{P} I \varphi)$
 $| SEventually: j \geq i \implies mem (\delta \sigma j i) I \implies SAT \sigma v j \varphi \implies SAT \sigma v i (\mathbf{F} I \varphi)$
 $| VEventually: (\bigwedge k. k \in (case right I of \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | enat b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies VIO \sigma v k \varphi) \implies$
 $\quad VIO \sigma v i (\mathbf{F} I \varphi)$
 $| SHistorically: j = (case right I of \infty \Rightarrow 0$
 $\quad | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + left I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies SAT \sigma v k \varphi) \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $| SHistoricallyOut: \tau \sigma i < \tau \sigma 0 + left I \implies SAT \sigma v i (\mathbf{H} I \varphi)$
 $| VHistorically: j \leq i \implies mem (\delta \sigma i j) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{H} I \varphi)$
 $| SAlways: (\bigwedge k. k \in (case right I of \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | enat b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies SAT \sigma v k \varphi) \implies$
 $\quad SAT \sigma v i (\mathbf{G} I \varphi)$
 $| VAlways: j \geq i \implies mem (\delta \sigma j i) I \implies VIO \sigma v j \varphi \implies VIO \sigma v i (\mathbf{G} I \varphi)$
 $| SSince: j \leq i \implies mem (\delta \sigma i j) I \implies SAT \sigma v j \psi \implies (\bigwedge k. k \in \{j <.. i\} \implies$
 $\quad SAT \sigma v k \varphi) \implies SAT \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSinceOut: \tau \sigma i < \tau \sigma 0 + left I \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSince: (case right I of \infty \Rightarrow True$
 $\quad | enat b \Rightarrow ETP \sigma ((\tau \sigma i) - b) \leq j) \implies$
 $\quad j \leq i \implies (\tau \sigma 0) + left I \leq (\tau \sigma i) \implies VIO \sigma v j \varphi \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \psi) \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| VSinceInf: j = (case right I of \infty \Rightarrow 0$
 $\quad | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad (\tau \sigma i) \geq (\tau \sigma 0) + left I \implies$
 $\quad (\bigwedge k. k \in \{j .. LTP_p \sigma i I\} \implies VIO \sigma v k \psi) \implies VIO \sigma v i (\varphi \mathbf{S} I \psi)$
 $| SUntil: j \geq i \implies mem (\delta \sigma j i) I \implies SAT \sigma v j \psi \implies (\bigwedge k. k \in \{i ..< j\} \implies SAT \sigma v k \varphi) \implies$
 $\quad SAT \sigma v i (\varphi \mathbf{U} I \psi)$
 $| VUntil: (case right I of \infty \Rightarrow True$
 $\quad | enat b \Rightarrow j < LTP_f \sigma i b) \implies$
 $\quad j \geq i \implies VIO \sigma v j \varphi \implies (\bigwedge k. k \in \{ETP_f \sigma i I .. j\} \implies VIO \sigma v k \psi) \implies$
 $\quad VIO \sigma v i (\varphi \mathbf{U} I \psi)$
 $| VUntilInf: (\bigwedge k. k \in (case right I of \infty \Rightarrow \{ETP_f \sigma i I ..\}$
 $\quad | enat b \Rightarrow \{ETP_f \sigma i I .. LTP_f \sigma i b\}) \implies VIO \sigma v k \psi) \implies$
 $\quad VIO \sigma v i (\varphi \mathbf{U} I \psi)$
 $| SMatchP: j \leq i \implies mem (\delta \sigma i j) I \implies Regex_Proof_System.SAT (SAT \sigma v) j i r \implies$
 $\quad SAT \sigma v i (MatchP I r)$
 $| VMatchPOut: \tau \sigma i < \tau \sigma 0 + left I \implies VIO \sigma v i (MatchP I r)$
 $| VMatchP: k = (case right I of \infty \Rightarrow 0 | enat b \Rightarrow ETP_p \sigma i b) \implies$
 $\quad \tau \sigma i \geq \tau \sigma 0 + left I \implies (\bigwedge j. j \in \{k .. LTP_p \sigma i I\} \implies Regex_Proof_System.VIO (VIO$
 $\sigma v) j i r) \implies$
 $\quad VIO \sigma v i (MatchP I r)$
 $| SMatchF: i \leq j \implies mem (\delta \sigma j i) I \implies Regex_Proof_System.SAT (SAT \sigma v) i j r \implies$

$SAT \sigma v i (MatchF I r)$
 $| VMatchF: (\bigwedge j. j \in (case\ right\ I\ of\ \infty \Rightarrow \{ETP_f\ \sigma\ i\ I\ ..\})$
 $\quad | enat\ b \Rightarrow \{ETP_f\ \sigma\ i\ I\ ..\ LTP_f\ \sigma\ i\ b\}) \Longrightarrow Regex_Proof_System.VIO (VIO\ \sigma\ v)$
 $i\ j\ r) \Longrightarrow$
 $\quad VIO\ \sigma\ v\ i\ (MatchF\ I\ r)$

6.1 Soundness and Completeness

lemma *not_sat_SinceD*:

assumes *unsat*: $\neg \langle \sigma, v, i \rangle \models \varphi$ **S** *I* ψ **and**

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

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

<proof>

lemma *not_sat_UntilD*:

assumes *unsat*: $\neg \langle \sigma, v, i \rangle \models \varphi$ **U** *I* ψ

and *witness*: $\exists j \geq i. mem\ (\delta\ \sigma\ j\ i)\ I \wedge \langle \sigma, v, j \rangle \models \psi$

shows $\exists j \geq i. (case\ right\ I\ of\ \infty \Rightarrow True\ | enat\ n \Rightarrow j < LTP\ \sigma\ (\tau\ \sigma\ i + n))$

$\wedge \neg (\langle \sigma, v, j \rangle \models \varphi) \wedge (\forall k \in \{(max\ i\ (ETP\ \sigma\ (\tau\ \sigma\ i + left\ I))) .. j\}.$

$\neg \langle \sigma, v, k \rangle \models \psi)$

<proof>

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>

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

lemma *completeness_raw*: $(\langle \sigma, v, i \rangle \models \varphi \longrightarrow SAT\ \sigma\ v\ i\ \varphi) \wedge (\neg \langle \sigma, v, i \rangle \models \varphi \longrightarrow VIO\ \sigma\ v\ i\ \varphi)$

<proof>

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

lemma *SAT_or_VIO*: $SAT\ \sigma\ v\ i\ \varphi \vee VIO\ \sigma\ v\ i\ \varphi$

<proof>

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*]: $xs \neq [] \Longrightarrow size\ (hd\ xs) < size_list\ size\ xs$

<proof>

lemma *size_last_estimation*[*termination_simp*]: $xs \neq [] \Longrightarrow size\ (last\ xs) < size_list\ size\ xs$

<proof>

lemma *size_rsproof_estimation*[*termination_simp*]: $x \in spatms\ p \Longrightarrow y < f\ x \Longrightarrow y < size_rsproof\ f\ p$

<proof>

lemma *size_rsproof_estimation'*[*termination_simp*]: $x \in spatms\ p \Longrightarrow y \leq f\ x \Longrightarrow y \leq size_rsproof\ f\ p$

<proof>

lemma *size_rvproof_estimation*[*termination_simp*]: $x \in vpatms\ p \Longrightarrow y < f\ x \Longrightarrow y < size_rvproof\ f\ p$

⟨proof⟩
lemma *size_rvproof_estimation*[*termination_simp*]: $x \in \text{vpatms } p \implies y \leq f x \implies y \leq \text{size_rvproof } f p$
 ⟨proof⟩

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⟩

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')
 ⟨proof⟩

termination ⟨proof⟩

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⟩

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 [] => s_at sp2 | _ => s_at (last sp1s))
| s_at (SUntil sp1s sp2) = (case sp1s of [] => s_at sp2 | sp1 # _ => 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 $\text{Cons_eq_upt_conv}: x \# xs = [m ..< n] \longleftrightarrow m < n \wedge x = m \wedge xs = [\text{Suc } m ..< n]$
 <proof>

lemma $\text{map_setE}[\text{elim_format}]: \text{map } f\ xs = ys \Longrightarrow y \in \text{set } ys \Longrightarrow \exists x \in \text{set } xs. f\ x = y$
 <proof>

lemma $\text{set_Cons_eq}: \text{set_Cons } X\ XS = (\bigcup xs \in XS. (\lambda x. x \# xs) ' X)$
 <proof>

lemma $\text{set_Cons_empty_iff}: \text{set_Cons } X\ XS = \{\} \longleftrightarrow (X = \{\} \vee XS = \{\})$
 <proof>

lemma $\text{infinite_set_ConsI}$:

$XS \neq \{\}$ \implies *infinite* $X \implies$ *infinite* (*set_Cons* X XS)
 $X \neq \{\}$ \implies *infinite* $XS \implies$ *infinite* (*set_Cons* X XS)
 <proof>

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
 <proof>

lemma *nth_fst_pos*: $x \in$ set $xs \implies$ $xs !$ (*the* (*fst_pos* xs x)) = x
 <proof>

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

lemma *eq_positions_iff*: length $xs =$ length ys
 \implies *positions* xs $x =$ *positions* ys $y \longleftrightarrow$ ($\forall n <$ length $xs.$ $xs ! n = x \longleftrightarrow$ $ys ! n = y$)
 <proof>

lemma *positions_eq_nil_iff*: *positions* xs $x =$ [] \longleftrightarrow $x \notin$ set xs
 <proof>

lemma *positions_nth*: $n \in$ set (*positions* xs x) \implies $xs ! n = x$
 <proof>

lemma *set_positions_eq*: set (*positions* xs x) = { $n.$ $xs ! n = x \wedge n <$ length xs }
 <proof>

lemma *positions_length*: $n \in$ set (*positions* xs x) \implies $n <$ length xs
 <proof>

lemma *positions_nth_cong*:
 $m \in$ set (*positions* xs x) \implies $n \in$ set (*positions* xs x) \implies $xs ! n = xs ! m$
 <proof>

lemma *fst_pos_in_positions*: $x \in$ set $xs \implies$ *the* (*fst_pos* xs x) \in set (*positions* xs x)
 <proof>

lemma *hd_positions_eq_fst_pos*: $x \in$ set $xs \implies$ *hd* (*positions* xs x) = *the* (*fst_pos* xs x)
 <proof>

lemma *sorted_positions*: *sorted* (*positions* xs x)
 <proof>

lemma *Min_sorted_list*: *sorted* $xs \implies$ $xs \neq$ [] \implies *Min* (set xs) = *hd* xs
 <proof>

lemma *Min_positions*: $x \in$ set $xs \implies$ *Min* (set (*positions* xs x)) = *the* (*fst_pos* xs x)
 <proof>

lemma *subset_positions_map_fst*: set (*positions* tXs tX) \subseteq set (*positions* (*map* *fst* tXs) (*fst* tX))
 <proof>

lemma *subset_positions_map_snd*: set (*positions* tXs tX) \subseteq set (*positions* (*map* *snd* tXs) (*snd* tX))

<proof>

lemma *Max_eqI*: $\text{finite } A \implies A \neq \{\} \implies (\bigwedge a. a \in A \implies a \leq b) \implies \exists a \in A. b \leq a \implies \text{Max } A = b$
<proof>

lemma *Max_Suc*: $X \neq \{\} \implies \text{finite } X \implies \text{Max } (\text{Suc } 'X) = \text{Suc } (\text{Max } X)$
<proof>

lemma *Max_insert0*: $X \neq \{\} \implies \text{finite } X \implies \text{Max } (\text{insert } (0::\text{nat}) X) = \text{Max } X$
<proof>

lemma *positions_Cons_notin_tail*: $x \notin \text{set } xs \implies \text{positions } (x \# xs) x = [0::\text{nat}]$
<proof>

lemma *Max_set_positions_Cons_hd*:
 $x \notin \text{set } xs \implies \text{Max } (\text{set } (\text{positions } (x \# xs) x)) = 0$
<proof>

lemma *Max_set_positions_Cons_tl*:
 $y \in \text{set } xs \implies \text{Max } (\text{set } (\text{positions } (x \# xs) y)) = \text{Suc } (\text{Max } (\text{set } (\text{positions } xs y)))$
<proof>

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

lemma *ball_swap*: $(\forall x \in A. \forall y \in B. P x y) = (\forall y \in B. \forall x \in A. P x y)$
<proof>

lemma *ball_triv_nonempty*: $A \neq \{\} \implies (\forall x \in A. P) = P$
<proof>

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

context fixes *test* :: 'a ⇒ 'b ⇒ bool **and** *testi* :: 'b ⇒ 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) ∧ *rs_check* r p1 ∧ *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]*:

$p = p' \implies (\bigwedge x \text{ sp}. x \in \text{regex.atms } r \implies \text{sp} \in \text{spatms } p \implies \text{t } x \text{ sp} = \text{t}' x \text{ sp})$
 $\implies (\bigwedge x. x \in \text{spatms } p \implies \text{ti } x = \text{ti}' x) \implies \text{rs_check } t \text{ ti } r \text{ p} = \text{rs_check } t' \text{ ti}' r \text{ p}'$
<proof>

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

$| \text{rv_check } (\text{Regex.Test } x) (\text{VTest_neq } i \ j) = (i < j)$
 $| \text{rv_check } (\text{Regex.Plus } r \ r') (\text{VPlus } p1 \ p2) =$
 $\quad (\text{rv_check } r \ p1 \wedge \text{rv_check } r' \ p2 \wedge \text{rv_at } \text{testi } p1 = \text{rv_at } \text{testi } p2)$
 $| \text{rv_check } (\text{Regex.Times } r \ r') (\text{VTimes } ps) = (ps \neq [] \wedge$
 $\quad (\exists i \ j. i = \text{fst } (\text{rv_at } \text{testi } (\text{snd } (\text{hd } ps))) \wedge j = \text{snd } (\text{rv_at } \text{testi } (\text{snd } (\text{last } ps)))) \wedge$
 $\quad i + \text{length } ps - 1 = j \wedge (\forall k \in \{0 \dots \text{length } ps\}. \text{let } (b, p) = ps ! k \text{ in}$
 $\quad \text{if } b \text{ then } \text{rv_check } r \ p \wedge \text{rv_at } \text{testi } p = (i, i + k)$
 $\quad \text{else } \text{rv_check } r' \ p \wedge \text{rv_at } \text{testi } p = (i + k, j))))$
 $| \text{rv_check } (\text{Regex.Star } r) (\text{VStar } ps) =$
 $\quad (\exists S \ T \ i \ j. S = \text{set } (\text{map } (\text{fst} \circ \text{rv_at } \text{testi}) \ ps) \wedge T = \text{set } (\text{map } (\text{snd} \circ \text{rv_at } \text{testi}) \ ps)$
 $\quad \wedge i = \text{Min } S \wedge j = \text{Max } T \wedge i \leq j \wedge S \cap T = \{\} \wedge S \cup T = \{i \dots j\}$
 $\quad \wedge \text{map } (\text{rv_at } \text{testi}) \ ps = \text{sorted_list_of_set } (\text{rm } (S \times T))$
 $\quad \wedge (\forall k \in \{0 \dots \text{length } ps\}. \text{rv_check } r \ (ps ! k)))$
 $| \text{rv_check } (\text{Regex.Star } r) (\text{VStar_gt } n \ n') = (n > n')$
 $| \text{rv_check } _ _ = \text{False}$

lemma *rv_check_code_Times*:

$\text{rv_check } (\text{Regex.Times } r \ r') (\text{VTimes } ps) = (ps \neq [] \wedge$
 $\quad (\text{let } i = \text{fst } (\text{rv_at } \text{testi } (\text{snd } (\text{hd } ps))); j = \text{snd } (\text{rv_at } \text{testi } (\text{snd } (\text{last } ps))) \text{ in}$
 $\quad i + \text{length } ps - 1 = j \wedge (\forall k \in \{0 \dots \text{length } ps\}. \text{let } (b, p) = ps ! k \text{ in}$
 $\quad \text{if } b \text{ then } \text{rv_check } r \ p \wedge \text{rv_at } \text{testi } p = (i, i + k)$
 $\quad \text{else } \text{rv_check } r' \ p \wedge \text{rv_at } \text{testi } p = (i + k, j))))$
 $\langle \text{proof} \rangle$

lemma *rv_check_code_Star*:

$\text{rv_check } (\text{Regex.Star } r) (\text{VStar } ps) =$
 $\quad (\text{let } S = \text{set } (\text{map } (\text{fst} \circ \text{rv_at } \text{testi}) \ ps); T = \text{set } (\text{map } (\text{snd} \circ \text{rv_at } \text{testi}) \ ps);$
 $\quad i = \text{Min } S; j = \text{Max } T \text{ in } i \leq j \wedge S \cap T = \{\} \wedge S \cup T = \{i \dots j\}$
 $\quad \wedge \text{map } (\text{rv_at } \text{testi}) \ ps = \text{sorted_list_of_set } (\text{rm } (S \times T))$
 $\quad \wedge (\forall k \in \{0 \dots \text{length } ps\}. \text{rv_check } r \ (ps ! k)))$
 $\langle \text{proof} \rangle$

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*]:

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

lemma *Cons_eq_upt_conv*: $x \# xs = [m \dots n] \iff m < n \wedge x = m \wedge xs = [\text{Suc } m \dots n]$
 $\langle \text{proof} \rangle$

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

lemma *rs_check_sound*:

$\forall x \in \text{Regex.atms } r. \forall p' \in \text{spatms } p. \text{test } x \ p' \longrightarrow \text{sat } (\text{testi } p') \ x \implies$
 $\text{rs_check } \text{test } \text{testi } r \ p \implies \text{Regex_Proof_System.SAT } \text{sat } (\text{fst } (\text{rs_at } \text{testi } p)) (\text{snd } (\text{rs_at } \text{testi } p)) \ r$
 $\langle \text{proof} \rangle$

lemma *rs_check_complete*:

$(\forall x \in \text{Regex.atms } r. \forall i. \text{sat } i \ x \longrightarrow (\exists p'. \text{testi } p' = i \wedge \text{test } x \ p')) \implies$
 $\text{Regex_Proof_System.SAT } \text{sat } i \ j \ r \implies \exists p. \text{rs_check } \text{test } \text{testi } r \ p \wedge \text{rs_at } \text{testi } p = (i, j)$
 $\langle \text{proof} \rangle$

lemma *rv_check_sound*:

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

$rv_check\ test\ testi\ r\ p \implies Regex_Proof_System.VIO\ vio\ (fst\ (rv_at\ testi\ p))\ (snd\ (rv_at\ testi\ p))\ r$
 <proof>

lemma *rv_check_complete*:

$(\forall x \in Regex.atms\ r. \forall i. vio\ i\ x \longrightarrow (\exists p'. testi\ p' = i \wedge test\ x\ p')) \implies$
 $Regex_Proof_System.VIO\ vio\ i\ j\ r \implies i \leq j \implies \exists p. rv_check\ test\ testi\ r\ p \wedge rv_at\ testi\ p = (i, j)$
 <proof>

lemma *rs_check_exec_rs_check*:

fixes *test* :: 'a \Rightarrow 'b \Rightarrow bool
and *testi* :: 'b \Rightarrow nat
and *test'* :: ('n \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool
and *FV* :: 'a \Rightarrow 'n set
and *C* :: 'n set \Rightarrow ('n \Rightarrow '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 $(\bigwedge x\ sp. x \in regex.atms\ r \implies test\ x\ sp = (\forall v \in C\ (FV\ x). test'\ v\ x\ sp)) \implies$
 $rs_check\ test\ testi\ r\ rsp = (\forall v \in \bigcap_{x \in regex.atms\ r}. C\ (FV\ x). rs_check\ (test'\ v)\ testi\ r\ rsp)$
 <proof>

lemma *rv_check_exec_rv_check*:

fixes *test* :: 'a \Rightarrow 'b \Rightarrow bool
and *testi* :: 'b \Rightarrow nat
and *test'* :: ('n \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool
and *FV* :: 'a \Rightarrow 'n set
and *C* :: 'n set \Rightarrow ('n \Rightarrow '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>

lemma *chain_sorted1*:

fixes *f* :: $_ \Rightarrow$ nat \times nat
assumes $\forall k \in \{Suc\ 0..<length\ ps\}. fst\ (f\ (ps\ !\ k)) = snd\ (f\ (ps\ !\ (k - Suc\ 0)))$
and $\forall k \in \{0..<length\ ps\}. fst\ (f\ (ps\ !\ k)) < snd\ (f\ (ps\ !\ k))$
and $j \leq k\ k < length\ ps$
shows $fst\ (f\ (ps\ !\ j)) \leq fst\ (f\ (ps\ !\ k))$
 <proof>

lemma *chain_sorted2*:

fixes *f* :: $_ \Rightarrow$ nat \times nat
assumes $\forall k \in \{Suc\ 0..<length\ ps\}. fst\ (f\ (ps\ !\ k)) = snd\ (f\ (ps\ !\ (k - Suc\ 0)))$
and $\forall k \in \{0..<length\ ps\}. fst\ (f\ (ps\ !\ k)) < snd\ (f\ (ps\ !\ k))$
and $j \leq k\ k < length\ ps$
shows $snd\ (f\ (ps\ !\ j)) \leq snd\ (f\ (ps\ !\ k))$
 <proof>

context

fixes *test* :: 'a \Rightarrow 'b \Rightarrow bool **and** *testi* :: 'b \Rightarrow nat **and** *SAT sat*
assumes *test_sound*: $\forall x \in regex.atms\ r. \forall p' \in spatms\ rsp. test\ x\ p' \longrightarrow SAT\ (testi\ p')\ x$
and *SAT_sound*: $\forall x \in regex.atms\ r. \forall i. SAT\ i\ x \longrightarrow sat\ i\ x$

begin

lemma *rs_check_le*:

$rs_check\ test\ testi\ r\ rsp \implies fst\ (rs_at\ testi\ rsp) \leq snd\ (rs_at\ testi\ rsp)$
(*proof*)

lemma *rs_check_le1*:

$rs_check\ test\ testi\ r\ rsp \implies sp \in spatms\ rsp \implies fst\ (rs_at\ testi\ rsp) \leq testi\ sp$
(*proof*)

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*)

end

lemma *rv_check_le*:

$rv_check\ test\ testi\ r\ rvp \implies vp \in vpatms\ rvp \implies fst\ (rv_at\ testi\ rvp) \leq snd\ (rv_at\ testi\ rvp)$
(*proof*)

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*)

9 Proof Checker

unbundle *MFOTL_syntax*

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

begin

fun *s_check* :: ('n, 'd) env \Rightarrow ('n, 'd) formula \Rightarrow ('n, 'd) sproof \Rightarrow bool

and *v_check* :: ('n, 'd) env \Rightarrow ('n, 'd) formula \Rightarrow ('n, 'd) vproof \Rightarrow bool **where**

s_check *v* *f* *p* = (case (f, p) of
| (\top , *STT* *i*) \Rightarrow True
| ($r \dagger ts$, *SPred* *i* *s* *ts'*) \Rightarrow
| ($r = s \wedge ts = ts' \wedge (r, v[[ts]]) \in \Gamma\ \sigma\ i$)
| ($x \approx c$, *SEq_Const* *i* *x'* *c'*) \Rightarrow
| ($c = c' \wedge x = x' \wedge v\ x = c$)
| ($\neg_F\ \varphi$, *SNeg* *vp*) \Rightarrow *v_check* *v* φ *vp*
| ($\varphi \vee_F\ \psi$, *SOrL* *sp1*) \Rightarrow *s_check* *v* φ *sp1*
| ($\varphi \vee_F\ \psi$, *SOrR* *sp2*) \Rightarrow *s_check* *v* ψ *sp2*
| ($\varphi \wedge_F\ \psi$, *SAnd* *sp1* *sp2*) \Rightarrow *s_check* *v* φ *sp1* \wedge *s_check* *v* ψ *sp2* \wedge *s_at* *sp1* = *s_at* *sp2*
| ($\varphi \longrightarrow_F\ \psi$, *SImpL* *vp1*) \Rightarrow *v_check* *v* φ *vp1*
| ($\varphi \longrightarrow_F\ \psi$, *SImpR* *sp2*) \Rightarrow *s_check* *v* ψ *sp2*
| ($\varphi \longleftrightarrow_F\ \psi$, *SIfSS* *sp1* *sp2*) \Rightarrow *s_check* *v* φ *sp1* \wedge *s_check* *v* ψ *sp2* \wedge *s_at* *sp1* = *s_at* *sp2*
| ($\varphi \longleftrightarrow_F\ \psi$, *SIfVV* *vp1* *vp2*) \Rightarrow *v_check* *v* φ *vp1* \wedge *v_check* *v* ψ *vp2* \wedge *v_at* *vp1* = *v_at* *vp2*
| ($\exists_F x.$ φ , *SExists* *y* *val* *sp*) \Rightarrow ($x = y \wedge s_check\ (v\ (x := val))\ \varphi\ sp$)
| ($\forall_F x.$ φ , *SForall* *y* *sp_part*) \Rightarrow (let *i* = *s_at* (*part_hd* *sp_part*)
in $x = y \wedge (\forall (sub, sp) \in SubsVals\ sp_part. s_at\ sp = i \wedge (\forall z \in sub. s_check\ (v\ (x := z))\ \varphi\ sp))$)
| (**Y** *I* φ , *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* φ , *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* φ , *SOnce* *i* *sp*) \Rightarrow

$(let\ j = s_at\ sp\ in$
 $j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{F}\ I\ \varphi, SEventually\ i\ sp) \Rightarrow$
 $(let\ j = s_at\ sp\ in$
 $j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge s_check\ v\ \varphi\ sp)$
 $| (\mathbf{H}\ I\ \varphi, SHistoricallyOut\ i) \Rightarrow$
 $\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{H}\ I\ \varphi, SHistorically\ i\ li\ sps) \Rightarrow$
 $(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - b))$
 $\wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge map\ s_at\ sps = [li\ ..<\ (LTP_p\ \sigma\ i\ I) + 1]$
 $\wedge (\forall\ sp \in set\ sps.\ s_check\ v\ \varphi\ sp))$
 $| (\mathbf{G}\ I\ \varphi, SAlways\ i\ hi\ sps) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b)$
 $\wedge right\ I \neq \infty$
 $\wedge map\ s_at\ sps = [(ETP_f\ \sigma\ i\ I) ..<\ hi + 1]$
 $\wedge (\forall\ sp \in set\ sps.\ s_check\ v\ \varphi\ sp))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, SSince\ sp2\ sp1s) \Rightarrow$
 $(let\ i = s_at\ (SSince\ sp2\ sp1s); j = s_at\ sp2\ in$
 $j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
 $\wedge map\ s_at\ sp1s = [j+1\ ..<\ i+1]$
 $\wedge s_check\ v\ \psi\ sp2$
 $\wedge (\forall\ sp1 \in set\ sp1s.\ s_check\ v\ \varphi\ sp1))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, SUntil\ sp1s\ sp2) \Rightarrow$
 $(let\ i = s_at\ (SUntil\ sp1s\ sp2); j = s_at\ sp2\ in$
 $j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$
 $\wedge map\ s_at\ sp1s = [i\ ..<\ j] \wedge s_check\ v\ \psi\ sp2$
 $\wedge (\forall\ sp1 \in set\ sp1s.\ s_check\ v\ \varphi\ sp1))$
 $| (\triangleleft\ I\ r, SMatchP\ rsp) \Rightarrow$
 $(let\ (j, i) = rs_at\ s_at\ rsp\ in\ j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge rs_check\ (s_check\ v)\ s_at\ r\ rsp)$
 $| (\triangleright\ I\ r, SMatchF\ rsp) \Rightarrow$
 $(let\ (i, j) = rs_at\ s_at\ rsp\ in\ i \leq j \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge rs_check\ (s_check\ v)\ s_at\ r\ rsp)$
 $| (_ , _) \Rightarrow False)$
 $| v_check\ v\ f\ p = (case\ (f, p)\ of$
 $(\perp, VFF\ i) \Rightarrow True$
 $| (r\ \dagger\ ts, VPred\ i\ pred\ ts') \Rightarrow$
 $(r = pred \wedge ts = ts' \wedge (r, v[[ts]]) \notin \Gamma\ \sigma\ i)$
 $| (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge v\ x \neq c)$
 $| (\neg_F\ \varphi, VNeg\ sp) \Rightarrow s_check\ v\ \varphi\ sp$
 $| (\varphi \vee_F\ \psi, VOr\ vp1\ vp2) \Rightarrow v_check\ v\ \varphi\ vp1 \wedge v_check\ v\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\varphi \wedge_F\ \psi, VAndL\ vp1) \Rightarrow v_check\ v\ \varphi\ vp1$
 $| (\varphi \wedge_F\ \psi, VAndR\ vp2) \Rightarrow v_check\ v\ \psi\ vp2$
 $| (\varphi \rightarrow_F\ \psi, VImp\ sp1\ vp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge v_check\ v\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $| (\varphi \leftarrow_F\ \psi, VIffSV\ sp1\ vp2) \Rightarrow s_check\ v\ \varphi\ sp1 \wedge v_check\ v\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $| (\varphi \leftarrow_F\ \psi, VIffVS\ vp1\ sp2) \Rightarrow v_check\ v\ \varphi\ vp1 \wedge s_check\ v\ \psi\ sp2 \wedge v_at\ vp1 = s_at\ sp2$
 $| (\exists_F x.\ \varphi, VExists\ y\ vp_part) \Rightarrow (let\ i = v_at\ (part_hd\ vp_part)$
 $in\ x = y \wedge (\forall\ (sub, vp) \in SubsVals\ vp_part.\ v_at\ vp = i \wedge (\forall\ z \in sub.\ v_check\ (v\ (x := z))\ \varphi\ vp)))$
 $| (\forall_F x.\ \varphi, VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v_check\ (v\ (x := val))\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrev\ vp) \Rightarrow$
 $(let\ j = v_at\ vp; i = v_at\ (VPrev\ vp)\ in$
 $i = j+1 \wedge v_check\ v\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrevZ) \Rightarrow True$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutL\ i) \Rightarrow$
 $i > 0 \wedge \Delta\ \sigma\ i < left\ I$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutR\ i) \Rightarrow$
 $i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I$
 $| (\mathbf{X}\ I\ \varphi, VNext\ vp) \Rightarrow$

```

    (let j = v_at vp; i = v_at (VNext vp) in
    j = i+1 ∧ v_check v φ vp)
| (X I φ, VNextOutL i) ⇒
  Δ σ (i+1) < left I
| (X I φ, VNextOutR i) ⇒
  enat (Δ σ (i+1)) > right I
| (P I φ, VOnceOut i) ⇒
  τ σ i < τ σ 0 + left I
| (P I φ, VOnce i li vps) ⇒
  (li = (case right I of ∞ ⇒ 0 | enat b ⇒ ETP_p σ i b)
  ∧ τ σ 0 + left I ≤ τ σ i
  ∧ map v_at vps = [li ..< (LTP_p σ i I) + 1]
  ∧ (∀ vp ∈ set vps. v_check v φ vp))
| (F I φ, VEventually i hi vps) ⇒
  (hi = (case right I of enat b ⇒ LTP_f σ i b) ∧ right I ≠ ∞
  ∧ map v_at vps = [(ETP_f σ i I) ..< hi + 1]
  ∧ (∀ vp ∈ set vps. v_check v φ vp))
| (H I φ, VHistorically i vp) ⇒
  (let j = v_at vp in
  j ≤ i ∧ mem (τ σ i - τ σ j) I ∧ v_check v φ vp)
| (G I φ, VAlways i vp) ⇒
  (let j = v_at vp
  in j ≥ i ∧ mem (τ σ j - τ σ i) I ∧ v_check v φ vp)
| (φ S I ψ, VSinceOut i) ⇒
  τ σ i < τ σ 0 + left I
| (φ S I ψ, VSince i vp1 vp2s) ⇒
  (let j = v_at vp1 in
  (case right I of ∞ ⇒ True | enat b ⇒ ETP_p σ i b ≤ j) ∧ j ≤ i
  ∧ τ σ 0 + left I ≤ τ σ i
  ∧ map v_at vp2s = [j ..< (LTP_p σ i I) + 1] ∧ v_check v φ vp1
  ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
| (φ S I ψ, VSinceInf i li vp2s) ⇒
  (li = (case right I of ∞ ⇒ 0 | enat b ⇒ ETP_p σ i b)
  ∧ τ σ 0 + left I ≤ τ σ i
  ∧ map v_at vp2s = [li ..< (LTP_p σ i I) + 1]
  ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
| (φ U I ψ, VUntil i vp2s vp1) ⇒
  (let j = v_at vp1 in
  (case right I of ∞ ⇒ True | enat b ⇒ j < LTP_f σ i b) ∧ i ≤ j
  ∧ map v_at vp2s = [ETP_f σ i I ..< j + 1] ∧ v_check v φ vp1
  ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
| (φ U I ψ, VUntilInf i hi vp2s) ⇒
  (hi = (case right I of enat b ⇒ LTP_f σ i b) ∧ right I ≠ ∞
  ∧ map v_at vp2s = [ETP_f σ i I ..< hi + 1]
  ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
| (◁ I r, VMatchPOut i) ⇒ τ σ i < τ σ 0 + left I
| (◁ I r, VMatchP i rvps) ⇒
  (let j = ETP σ (case right I of ∞ ⇒ 0 | enat n ⇒ τ σ i - n)
  in τ σ i ≥ τ σ 0 + left I ∧ map (fst ∘ rv_at v_at) rvps = [j ..< Suc (LTP_p σ i I)] ∧
  (∀ rvp ∈ set rvps. rv_check (v_check v) v_at r rvp ∧ snd (rv_at v_at rvp) = i))
| (▷ I r, VMatchF i rvps) ⇒
  (let j = LTP σ (case right I of ∞ ⇒ 0 | enat n ⇒ τ σ i + n)
  in map (snd ∘ rv_at v_at) rvps = [ETP_f σ i I ..< Suc j] ∧ right I ≠ ∞ ∧
  (∀ rvp ∈ set rvps. rv_check (v_check v) v_at r rvp ∧ fst (rv_at v_at rvp) = i))
| ( _ , _ ) ⇒ 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*:

$s_check\ v\ \varphi\ sp \implies SAT\ \sigma\ v\ (s_at\ sp)\ \varphi$
 $v_check\ v\ \varphi\ vp \implies VIO\ \sigma\ v\ (v_at\ vp)\ \varphi$

<proof>

definition *compatible* $X\ vs\ v \longleftrightarrow (\forall x \in X. v\ x \in vs\ x)$

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

lemma *compatible_alt*:

$compatible\ X\ vs\ v \longleftrightarrow v \in compatible_vals\ X\ vs$

<proof>

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

<proof>

lemma *compatible_vals_empty_eq*: $compatible_vals\ \{\}\ vs = UNIV$

<proof>

lemma *compatible_union_iff*:

$compatible\ (X \cup Y)\ vs\ v \longleftrightarrow compatible\ X\ vs\ v \wedge compatible\ Y\ vs\ v$

<proof>

lemma *compatible_vals_union_eq*:

$compatible_vals\ (X \cup Y)\ vs = compatible_vals\ X\ vs \cap compatible_vals\ Y\ vs$

<proof>

lemma *compatible_vals_Union_eq*:

$compatible_vals\ (\bigcup x \in X. Y\ x)\ vs = (\bigcap x \in X. compatible_vals\ (Y\ x)\ vs)$

<proof>

lemma *compatible_antimono*:

$compatible\ X\ vs\ v \implies Y \subseteq X \implies compatible\ Y\ vs\ v$

<proof>

lemma *compatible_vals_antimono*:

$Y \subseteq X \implies compatible_vals\ X\ vs \subseteq compatible_vals\ Y\ vs$

<proof>

lemma *compatible_extensible*:

$(\forall x. vs\ x \neq \{\}) \implies compatible\ X\ vs\ v \implies X \subseteq Y \implies \exists v'. compatible\ Y\ vs\ v' \wedge (\forall x \in X. v\ x = v'\ x)$

<proof>

lemmas $compatible_vals_extensible = compatible_extensible[unfolded\ compatible_alt]$

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

where $mk_values\ [] = \{\}\}$

| $mk_values\ (T \# Ts) = (case\ T\ of$

$(\mathbf{v}\ x, X) \Rightarrow$

$let\ terms = map\ fst\ Ts\ in$

$if\ \mathbf{v}\ x \in set\ terms\ then$

$let\ fst_pos = hd\ (positions\ terms\ (\mathbf{v}\ x))\ in\ (\lambda xs. (xs\ !\ fst_pos)\ \# xs)\ ' (mk_values\ Ts)$

$else\ set_Cons\ X\ (mk_values\ Ts)$

| $(\mathbf{c}\ a, X) \Rightarrow set_Cons\ X\ (mk_values\ Ts)$)

lemma *mk_values_nempty*:

$\{\} \notin \text{set } (\text{map } \text{snd } tXs) \implies \text{mk_values } tXs \neq \{\}$
 ⟨proof⟩

lemma *mk_values_not_Nil*:

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

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

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

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

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

$vs ! n = vs ! m$

⟨proof⟩

definition *mk_values_subset* $p \ tXs \ X$

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

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

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

$\text{if } \text{inf_dups} = [] \text{ then (if finite } X \text{ then False else Code.abort STR "subset on infinite subset" } (\lambda_. \{p\} \times \text{mk_values } tXs \subseteq X))$

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

inf_dups

$\text{then } \{p\} \times \text{mk_values } tXs \subseteq X$

$\text{else (if finite } X \text{ then False else Code.abort STR "subset on infinite subset" } (\lambda_. \{p\} \times \text{mk_values } tXs \subseteq X))$

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

⟨proof⟩

lemma *infinite_mk_values1*: $\forall tX \in \text{set } tXs. \text{snd } tX \neq \{\} \implies tY \in \text{set } tXs \implies$

$\forall Y. (\text{fst } tY, Y) \in \text{set } tXs \longrightarrow \text{infinite } Y \implies \text{infinite } (\text{mk_values } tXs)$

⟨proof⟩

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⟩

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

$\text{mk_values_subset } p \ tXs \ X \longleftrightarrow \{p\} \times \text{mk_values } tXs \subseteq X$

⟨proof⟩

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⟩

lemma *fst_eval_trm_set[simp]*:

$\text{fst } (vs \llbracket t \rrbracket) = t$

⟨proof⟩

lemma *mk_values_complete*: $cs = v \llbracket ts \rrbracket \implies$

$v \in \text{compatible_vals } (fv \ (r \dagger \ ts)) \ vs \implies$

$cs \in \text{mk_values } (vs \llbracket ts \rrbracket)$

⟨proof⟩

definition $mk_values_subset_Compl\ r\ vs\ ts\ i = (\{r\} \times mk_values\ (vs\ \{\!\{ts\}\!\}) \subseteq -\ \Gamma\ \sigma\ i)$

fun $check_values$ **where**

$check_values\ _\ _\ _\ _\ None = None$
 $| check_values\ vs\ (\mathbf{c}\ c\ \#\ ts)\ (u\ \#)\ us\ f = (if\ c = u\ then\ check_values\ vs\ ts\ us\ f\ else\ None)$
 $| check_values\ vs\ (\mathbf{v}\ x\ \#)\ ts)\ (u\ \#)\ us\ (Some\ v) = (if\ u \in vs\ x \wedge (v\ x = Some\ u \vee v\ x = None)\ then\ check_values\ vs\ ts\ us\ (Some\ (v(x \mapsto u)))\ else\ None)$
 $| check_values\ vs\ []\ []\ f = f$
 $| check_values\ _\ _\ _\ _\ = None$

lemma mk_values_alt :

$mk_values\ (vs\ \{\!\{ts\}\!\}) =$
 $\{cs.\ \exists v \in compatible_vals\ (\bigcup (fv_trm\ 'set\ ts))\ vs.\ cs = v\ \{\!\{ts\}\!\}\}$
 $\langle proof \rangle$

lemma $check_values_neq_NoneI$:

assumes $v \in compatible_vals\ (\bigcup (fv_trm\ 'set\ ts) - dom\ f)\ vs\ \wedge x\ y.\ f\ x = Some\ y \implies y \in vs\ x$
shows $check_values\ vs\ ts\ ((\lambda x.\ case\ f\ x\ of\ None \Rightarrow v\ x \mid Some\ y \Rightarrow y)\ \{\!\{ts\}\!\})\ (Some\ f) \neq None$
 $\langle proof \rangle$

lemma $check_values_eq_NoneI$:

$\forall v \in compatible_vals\ (\bigcup (fv_trm\ 'set\ ts) - dom\ f)\ vs.\ us \neq (\lambda x.\ case\ f\ x\ of\ None \Rightarrow v\ x \mid Some\ y \Rightarrow y)\ \{\!\{ts\}\!\} \implies$
 $check_values\ vs\ ts\ us\ (Some\ f) = None$
 $\langle proof \rangle$

lemma $mk_values_subset_Compl_code[code]$:

$mk_values_subset_Compl\ r\ vs\ ts\ i = (\forall (q,\ us) \in \Gamma\ \sigma\ i.\ q \neq r \vee check_values\ vs\ ts\ us\ (Some\ Map.empty) = None)$
 $\langle proof \rangle$

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**

$s_check_exec\ vs\ f\ p = (case\ (f,\ p)\ of$
 $(\top,\ STT\ i) \Rightarrow True$
 $| (r\ \dagger\ ts,\ SPred\ i\ s\ ts') \Rightarrow$
 $(r = s \wedge ts = ts' \wedge mk_values_subset\ r\ (vs\ \{\!\{ts\}\!\})\ (\Gamma\ \sigma\ i))$
 $| (x \approx c,\ SEq_Const\ i\ x'\ c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge vs\ x = \{c\})$
 $| (\neg_F\ \varphi,\ SNeg\ vp) \Rightarrow v_check_exec\ vs\ \varphi\ vp$
 $| (\varphi \vee_F\ \psi,\ SOrL\ sp1) \Rightarrow s_check_exec\ vs\ \varphi\ sp1$
 $| (\varphi \vee_F\ \psi,\ SOrR\ sp2) \Rightarrow s_check_exec\ vs\ \psi\ sp2$
 $| (\varphi \wedge_F\ \psi,\ SAnd\ sp1\ sp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \longrightarrow_F\ \psi,\ SImpL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1$
 $| (\varphi \longrightarrow_F\ \psi,\ SImpR\ sp2) \Rightarrow s_check_exec\ vs\ \psi\ sp2$
 $| (\varphi \longleftrightarrow_F\ \psi,\ SIffSS\ sp1\ sp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge s_at\ sp1 = s_at\ sp2$
 $| (\varphi \longleftrightarrow_F\ \psi,\ SIffVV\ vp1\ vp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\exists_F x.\ \varphi,\ SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s_check_exec\ (vs\ (x := \{val\}))\ \varphi\ sp)$
 $| (\forall_F x.\ \varphi,\ SForall\ y\ sp_part) \Rightarrow (let\ i = s_at\ (part_hd\ sp_part)$
 $in\ x = y \wedge (\forall (sub,\ sp) \in SubsVals\ sp_part.\ s_at\ sp = i \wedge s_check_exec\ (vs\ (x := sub))\ \varphi\ sp))$
 $| (\mathbf{Y}\ I\ \varphi,\ SPrev\ sp) \Rightarrow$
 $(let\ j = s_at\ sp;\ i = s_at\ (SPrev\ sp)\ in$
 $i = j+1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{X}\ I\ \varphi,\ SNext\ sp) \Rightarrow$

$(let\ j = s_at\ sp; i = s_at\ (SNext\ sp)\ in$
 $j = i+1 \wedge mem\ (\Delta\ \sigma\ j)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{P}\ I\ \varphi, SOnce\ i\ sp) \Rightarrow$
 $(let\ j = s_at\ sp\ in$
 $j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{F}\ I\ \varphi, SEventually\ i\ sp) \Rightarrow$
 $(let\ j = s_at\ sp\ in$
 $j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge s_check_exec\ vs\ \varphi\ sp)$
 $| (\mathbf{H}\ I\ \varphi, SHistoricallyOut\ i) \Rightarrow$
 $\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{H}\ I\ \varphi, SHistorically\ i\ li\ sps) \Rightarrow$
 $(li = (case\ right\ I\ of\ \infty \Rightarrow 0\ |\ enat\ b \Rightarrow ETP\ \sigma\ (\tau\ \sigma\ i - b))$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ map\ s_at\ sps = [li\ ..<\ (LTP_p\ \sigma\ i\ I) + 1]$
 $\wedge\ (\forall\ sp \in set\ sps.\ s_check_exec\ vs\ \varphi\ sp))$
 $| (\mathbf{G}\ I\ \varphi, SAlways\ i\ hi\ sps) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b)$
 $\wedge\ right\ I \neq \infty$
 $\wedge\ map\ s_at\ sps = [(ETP_f\ \sigma\ i\ I)\ ..<\ hi + 1]$
 $\wedge\ (\forall\ sp \in set\ sps.\ s_check_exec\ vs\ \varphi\ sp))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, SSince\ sp2\ sp1s) \Rightarrow$
 $(let\ i = s_at\ (SSince\ sp2\ sp1s); j = s_at\ sp2\ in$
 $j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
 $\wedge\ map\ s_at\ sp1s = [j+1\ ..<\ i+1]$
 $\wedge\ s_check_exec\ vs\ \psi\ sp2$
 $\wedge\ (\forall\ sp1 \in set\ sp1s.\ s_check_exec\ vs\ \varphi\ sp1))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, SUntil\ sp1s\ sp2) \Rightarrow$
 $(let\ i = s_at\ (SUntil\ sp1s\ sp2); j = s_at\ sp2\ in$
 $j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$
 $\wedge\ map\ s_at\ sp1s = [i\ ..<\ j] \wedge s_check_exec\ vs\ \psi\ sp2$
 $\wedge\ (\forall\ sp1 \in set\ sp1s.\ s_check_exec\ vs\ \varphi\ sp1))$
 $| (\triangleleft\ I\ r, SMatchP\ rsp) \Rightarrow$
 $(let\ (j, i) = rs_at\ s_at\ rsp\ in\ j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge rs_check\ (s_check_exec\ vs)\ s_at\ r$
 $rsp)$
 $| (\triangleright\ I\ r, SMatchF\ rsp) \Rightarrow$
 $(let\ (i, j) = rs_at\ s_at\ rsp\ in\ i \leq j \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge rs_check\ (s_check_exec\ vs)\ s_at\ r$
 $rsp)$
 $| (_ , _) \Rightarrow False)$
 $| v_check_exec\ vs\ f\ p = (case\ (f, p)\ of$
 $(\perp, VFF\ i) \Rightarrow True$
 $| (r\ \dagger\ ts, VPred\ i\ pred\ ts') \Rightarrow$
 $(r = pred \wedge ts = ts' \wedge mk_values_subset_Compl\ r\ vs\ ts\ i)$
 $| (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge c \notin vs\ x)$
 $| (\neg_F\ \varphi, VNeg\ sp) \Rightarrow s_check_exec\ vs\ \varphi\ sp$
 $| (\varphi\ \vee_F\ \psi, VOr\ vp1\ vp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\varphi\ \wedge_F\ \psi, VAndL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1$
 $| (\varphi\ \wedge_F\ \psi, VAndR\ vp2) \Rightarrow v_check_exec\ vs\ \psi\ vp2$
 $| (\varphi \longrightarrow_F\ \psi, VImp\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at$
 $vp2$
 $| (\varphi \longleftarrow_F\ \psi, VIffSV\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at$
 $vp2$
 $| (\varphi \longleftrightarrow_F\ \psi, VIffVS\ vp1\ sp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge v_at\ vp1 = s_at$
 $sp2$
 $| (\exists_Fx.\ \varphi, VExists\ y\ vp_part) \Rightarrow (let\ i = v_at\ (part_hd\ vp_part)$
 $in\ x = y \wedge (\forall\ (sub, vp) \in SubsVals\ vp_part.\ v_at\ vp = i \wedge v_check_exec\ (vs\ (x := sub))\ \varphi\ vp))$
 $| (\forall_Fx.\ \varphi, VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v_check_exec\ (vs\ (x := \{val\}))\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrev\ vp) \Rightarrow$

$(let\ j = v_at\ vp; i = v_at\ (VPrev\ vp)\ in$
 $i = j+1 \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrevZ) \Rightarrow True$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutL\ i) \Rightarrow$
 $i > 0 \wedge \Delta\ \sigma\ i < left\ I$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutR\ i) \Rightarrow$
 $i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I$
 $| (\mathbf{X}\ I\ \varphi, VNext\ vp) \Rightarrow$
 $(let\ j = v_at\ vp; i = v_at\ (VNext\ vp)\ in$
 $j = i+1 \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{X}\ I\ \varphi, VNextOutL\ i) \Rightarrow$
 $\Delta\ \sigma\ (i+1) < left\ I$
 $| (\mathbf{X}\ I\ \varphi, VNextOutR\ i) \Rightarrow$
 $enat\ (\Delta\ \sigma\ (i+1)) > right\ I$
 $| (\mathbf{P}\ I\ \varphi, VOnceOut\ i) \Rightarrow$
 $\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{P}\ I\ \varphi, VOnce\ i\ li\ vps) \Rightarrow$
 $(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ map\ v_at\ vps = [li\ ..< (LTP_p\ \sigma\ i\ I) + 1]$
 $\wedge\ (\forall\ vp \in set\ vps.\ v_check_exec\ vs\ \varphi\ vp))$
 $| (\mathbf{F}\ I\ \varphi, VEventually\ i\ hi\ vps) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
 $\wedge\ map\ v_at\ vps = [(ETP_f\ \sigma\ i\ I) ..< hi + 1]$
 $\wedge\ (\forall\ vp \in set\ vps.\ v_check_exec\ vs\ \varphi\ vp))$
 $| (\mathbf{H}\ I\ \varphi, VHistorically\ i\ vp) \Rightarrow$
 $(let\ j = v_at\ vp\ in$
 $j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{G}\ I\ \varphi, VAlways\ i\ vp) \Rightarrow$
 $(let\ j = v_at\ vp$
 $in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSinceOut\ i) \Rightarrow$
 $\tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSince\ i\ vp1\ vp2s) \Rightarrow$
 $(let\ j = v_at\ vp1\ in$
 $(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b \leq j) \wedge j \leq i$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ map\ v_at\ vp2s = [j\ ..< (LTP_p\ \sigma\ i\ I) + 1] \wedge v_check_exec\ vs\ \varphi\ vp1$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSinceInf\ i\ li\ vp2s) \Rightarrow$
 $(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ map\ v_at\ vp2s = [li\ ..< (LTP_p\ \sigma\ i\ I) + 1]$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, VUntil\ i\ vp2s\ vp1) \Rightarrow$
 $(let\ j = v_at\ vp1\ in$
 $(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP_f\ \sigma\ i\ b) \wedge i \leq j$
 $\wedge\ map\ v_at\ vp2s = [ETP_f\ \sigma\ i\ I\ ..< j + 1] \wedge v_check_exec\ vs\ \varphi\ vp1$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $| (\varphi\ \mathbf{U}\ I\ \psi, VUntilInf\ i\ hi\ vp2s) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
 $\wedge\ map\ v_at\ vp2s = [ETP_f\ \sigma\ i\ I\ ..< hi + 1]$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $| (\triangleleft\ I\ r, VMatchPOut\ i) \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\triangleleft\ I\ r, VMatchP\ i\ rvps) \Rightarrow$
 $(let\ j = ETP\ \sigma\ (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ n \Rightarrow \tau\ \sigma\ i - n)$
 $in\ \tau\ \sigma\ i \geq \tau\ \sigma\ 0 + left\ I \wedge map\ (fst \circ rv_at\ v_at)\ rvps = [j\ ..< Suc\ (LTP_p\ \sigma\ i\ I)] \wedge$
 $(\forall\ rvp \in set\ rvps.\ rv_check\ (v_check_exec\ vs)\ v_at\ r\ rvp \wedge snd\ (rv_at\ v_at\ rvp) = i))$

| ($\triangleright I r, VMatchF i rvps$) \Rightarrow
 (let $j = LTP \sigma$ (case right I of $\infty \Rightarrow 0$ | $enat n \Rightarrow \tau \sigma i + n$)
 in map (snd $\circ rv_at v_at$) $rvps = [ETP_f \sigma i I ..< Suc j] \wedge right I \neq \infty \wedge$
 ($\forall rvp \in set rvps. rv_check (v_check_exec vs) v_at r rvp \wedge fst (rv_at v_at rvp) = i$)
 | ($_ , _$) $\Rightarrow False$)

declare $s_check_exec.simps[simp del]$ $v_check_exec.simps[simp del]$
simps_of_case $s_check_exec_simps[simp]: s_check_exec.simps[unfolded prod.case]$ (*splits: formula.split sproof.split*)
simps_of_case $v_check_exec_simps[simp]: v_check_exec.simps[unfolded prod.case]$ (*splits: formula.split vproof.split*)

lemma $check_fv_cong$:
assumes $\forall x \in fv \varphi. v x = v' x$
shows $s_check v \varphi sp \longleftrightarrow s_check v' \varphi sp$ $v_check v \varphi vp \longleftrightarrow v_check v' \varphi vp$
<proof>

lemma $s_check_fun_upd_notin[simp]$:
 $x \notin fv \varphi \Longrightarrow s_check (v(x := t)) \varphi sp = s_check v \varphi sp$
<proof>

lemma $v_check_fun_upd_notin[simp]$:
 $x \notin fv \varphi \Longrightarrow v_check (v(x := t)) \varphi sp = v_check v \varphi sp$
<proof>

lemma $SubsVals_nonempty$: $(X, t) \in SubsVals part \Longrightarrow X \neq \{\}$
<proof>

lemma $compatible_vals_nonemptyI$: $\forall x. vs x \neq \{\} \Longrightarrow compatible_vals A vs \neq \{\}$
<proof>

lemma $compatible_vals_fun_upd$: $compatible_vals A (vs(x := X)) =$
(if $x \in A$ then $\{v \in compatible_vals (A - \{x\}) vs. v x \in X\}$ else $compatible_vals A vs$)
<proof>

lemma $fun_upd_in_compatible_vals$: $v \in compatible_vals (A - \{x\}) vs \Longrightarrow v(x := t) \in compatible_vals (A - \{x\}) vs$
<proof>

lemma $fun_upd_in_compatible_vals_in$: $v \in compatible_vals (A - \{x\}) vs \Longrightarrow t \in vs x \Longrightarrow v(x := t) \in compatible_vals A vs$
<proof>

lemma $fun_upd_in_compatible_vals_notin$: $x \notin A \Longrightarrow v \in compatible_vals A vs \Longrightarrow v(x := t) \in compatible_vals A vs$
<proof>

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

lemma $s_check_code[code]$: $s_check v \varphi sp = s_check_exec (\lambda x. \{v x\}) \varphi sp$
<proof>

lemma $v_check_code[code]$: $v_check v \varphi vp = v_check_exec (\lambda x. \{v x\}) \varphi vp$
<proof>

9.3 Latest Relevant Time-Point

fun $rLRTP :: ('a \Rightarrow nat \Rightarrow nat\ option) \Rightarrow 'a\ Regex.regex \Rightarrow nat \Rightarrow nat\ option$ **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 $rLRTP_cong[fundef_cong]$:

$(\bigwedge x. x \in regex.atms\ r \implies LRTP\ x\ i = LRTP'\ x\ i) \implies rLRTP\ LRTP\ r\ i = rLRTP\ LRTP'\ r\ i$
 $\langle proof \rangle$

lemma fb_rLRTP :

assumes $\forall \varphi \in regex.atms\ r. future_bounded\ \varphi \wedge \neg Option.is_none\ (LRTP\ \varphi\ i)$
shows $\neg Option.is_none\ (rLRTP\ LRTP\ r\ i)$
 $\langle proof \rangle$

fun $LRTP :: ('n, 'd)\ formula \Rightarrow nat \Rightarrow nat\ option$ **where**

$LRTP\ \top\ i = Some\ i$
 $| LRTP\ \perp\ i = Some\ i$
 $| LRTP\ (_ \dagger _) i = Some\ i$
 $| LRTP\ (_ \approx _) i = Some\ i$
 $| LRTP\ (\neg_F\ \varphi)\ i = LRTP\ \varphi\ i$
 $| LRTP\ (\varphi \vee_F\ \psi)\ i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \wedge_F\ \psi)\ i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \rightarrow_F\ \psi)\ i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \longleftrightarrow_F\ \psi)\ i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\exists_{F_}.\ \varphi)\ i = LRTP\ \varphi\ i$
 $| LRTP\ (\forall_{F_}.\ \varphi)\ i = LRTP\ \varphi\ i$
 $| LRTP\ (\mathbf{Y}\ I\ \varphi)\ i = LRTP\ \varphi\ (i-1)$
 $| LRTP\ (\mathbf{X}\ I\ \varphi)\ i = LRTP\ \varphi\ (i+1)$
 $| LRTP\ (\mathbf{P}\ I\ \varphi)\ i = LRTP\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $| LRTP\ (\mathbf{H}\ I\ \varphi)\ i = LRTP\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $| LRTP\ (\mathbf{F}\ I\ \varphi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP_f\ \sigma\ i\ b))$
 $| LRTP\ (\mathbf{G}\ I\ \varphi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP_f\ \sigma\ i\ b))$
 $| LRTP\ (\varphi\ \mathbf{S}\ I\ \psi)\ i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ (LTP_p_safe\ \sigma\ i\ I))$
 $| LRTP\ (\varphi\ \mathbf{U}\ I\ \psi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow max_opt\ (LRTP\ \varphi\ ((LTP_f\ \sigma\ i\ b)-1))\ (LRTP\ \psi\ (LTP_f\ \sigma\ i\ b)))$
 $| LRTP\ (\triangleleft I\ r)\ i =$
 $(let\ X = (\lambda\varphi. LRTP\ \varphi\ i)\ ' regex.atms\ r\ in$
 $if\ X = \{\} then\ Some\ i\ else\ if\ None \in X then\ None\ else\ Some\ (Max\ (the\ ' X)))$
 $| LRTP\ (\triangleright I\ r)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow$
 $let\ X = (\lambda\varphi. LRTP\ \varphi\ (LTP_f\ \sigma\ i\ b))\ ' regex.atms\ r\ in$
 $if\ X = \{\} then\ Some\ (LTP_f\ \sigma\ i\ b)\ else\ if\ None \in X then\ None\ else\ Some\ (Max\ (the\ ' X)))$

lemma fb_LRTP :

assumes $future_bounded\ \varphi$
shows $\neg Option.is_none\ (LRTP\ \varphi\ i)$
 $\langle proof \rangle$

lemma $not_none_fb_LRTP$:

assumes $future_bounded\ \varphi$
shows $LRTP\ \varphi\ i \neq None$
 $\langle proof \rangle$

lemma $is_some_fb_LRTP$:

assumes $future_bounded\ \varphi$
shows $\exists j. LRTP\ \varphi\ i = Some\ j$

<proof>

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

9.4 Active Domain

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

lemma *val_in_AD_iff*:
 $x \in fv\ \varphi \implies v\ x \in AD\ \varphi\ i \longleftrightarrow v\ x \in consts\ \varphi \vee$
 $(\exists r\ ts\ k. k \leq the\ (LRTP\ \varphi\ i) \wedge (r, v[[ts]]) \in \Gamma\ \sigma\ k \wedge x \in \bigcup (set\ (map\ fv_trm\ ts)))$
<proof>

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

lemma *finite_values*: $finite\ (\bigcup (set\ 'snd\ ' \Gamma\ \sigma\ k))$
<proof>

lemma *finite_tps*: $future_bounded\ \varphi \implies finite\ (\bigcup k < the\ (LRTP\ \varphi\ i). \{k\})$
<proof>

lemma *finite_AD [simp]*: $future_bounded\ \varphi \implies finite\ (AD\ \varphi\ i)$
<proof>

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

9.5 Congruence Modulo Active Domain

lemma *AD_simps[simp]*:
 $AD\ (\neg_F\ \varphi)\ i = AD\ \varphi\ i$
 $future_bounded\ (\varphi \vee_F\ \psi) \implies AD\ (\varphi \vee_F\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
 $future_bounded\ (\varphi \wedge_F\ \psi) \implies AD\ (\varphi \wedge_F\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
 $future_bounded\ (\varphi \longrightarrow_F\ \psi) \implies AD\ (\varphi \longrightarrow_F\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
 $future_bounded\ (\varphi \longleftrightarrow_F\ \psi) \implies AD\ (\varphi \longleftrightarrow_F\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
 $AD\ (\exists_F x. \varphi)\ i = AD\ \varphi\ i$
 $AD\ (\forall_F x. \varphi)\ i = AD\ \varphi\ i$
 $AD\ (\mathbf{Y}\ I\ \varphi)\ i = AD\ \varphi\ (i - 1)$
 $AD\ (\mathbf{X}\ I\ \varphi)\ i = AD\ \varphi\ (i + 1)$
 $future_bounded\ (\mathbf{F}\ I\ \varphi) \implies AD\ (\mathbf{F}\ I\ \varphi)\ i = AD\ \varphi\ (LTP_f\ \sigma\ i\ (the_enat\ (right\ I)))$
 $future_bounded\ (\mathbf{G}\ I\ \varphi) \implies AD\ (\mathbf{G}\ I\ \varphi)\ i = AD\ \varphi\ (LTP_f\ \sigma\ i\ (the_enat\ (right\ I)))$
 $AD\ (\mathbf{P}\ I\ \varphi)\ i = AD\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $AD\ (\mathbf{H}\ I\ \varphi)\ i = AD\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $future_bounded\ (\varphi\ \mathbf{S}\ I\ \psi) \implies AD\ (\varphi\ \mathbf{S}\ I\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ (LTP_p_safe\ \sigma\ i\ I)$
 $future_bounded\ (\varphi\ \mathbf{U}\ I\ \psi) \implies AD\ (\varphi\ \mathbf{U}\ I\ \psi)\ i = AD\ \varphi\ (LTP_f\ \sigma\ i\ (the_enat\ (right\ I))) - 1 \cup AD\ \psi\ (LTP_f\ \sigma\ i\ (the_enat\ (right\ I)))$
<proof>

lemma *AD_simps_regex[simp]*:
 $future_bounded\ (\triangleleft I\ r) \implies regex.atms\ r \neq \{\}\ \implies AD\ (\triangleleft I\ r)\ i = (\bigcup \varphi \in regex.atms\ r. AD\ \varphi\ i)$
 $future_bounded\ (\triangleright I\ r) \implies regex.atms\ r \neq \{\}\ \implies AD\ (\triangleright I\ r)\ i = (\bigcup \varphi \in regex.atms\ r. AD\ \varphi\ (LTP_f\ \sigma\ i\ (the_enat\ (right\ I))))$

<proof>

lemma *LTP_p_mono*: $i \leq j \implies LTP_p_safe \sigma i I \leq LTP_p_safe \sigma j I$
<proof>

lemma *LTP_τ_mono*:
assumes $\tau \sigma i \leq u$
shows $LTP \sigma (\tau \sigma i) \leq LTP \sigma u$
<proof>

lemma *LTP_f_mono*:
assumes $i \leq j$
shows $LTP_f \sigma i b \leq LTP_f \sigma j b$
<proof>

lemma *LRTP_mono*: $future_bounded \varphi \implies i \leq j \implies the (LRTP \varphi i) \leq the (LRTP \varphi j)$
<proof>

lemma *AD_mono*: $future_bounded \varphi \implies i \leq j \implies AD \varphi i \subseteq AD \varphi j$
<proof>

lemma *LTP_p_safe_le[simp]*: $LTP_p_safe \sigma i I \leq i$
<proof>

lemma *check_AD_cong*:
assumes $future_bounded \varphi$
and $(\forall x \in fv \varphi. v x = v' x \vee (v x \notin AD \varphi i \wedge v' x \notin AD \varphi i))$
shows $(s_at sp = i \implies s_check v \varphi sp \longleftrightarrow s_check v' \varphi sp)$
 $(v_at vp = i \implies v_check v \varphi vp \longleftrightarrow v_check v' \varphi vp)$
<proof>

9.6 Checker Completeness

lemma *part_hd_tabulate*: $distinct xs \implies part_hd (tabulate xs f z) = (case xs of [] \Rightarrow z \mid (x \# _) \Rightarrow (if set xs = UNIV then f x else z))$
<proof>

lemma *s_at_tabulate*:
assumes $\forall z. s_at (mypick z) = i$
and $mypart = tabulate (sorted_list_of_set (AD \varphi i)) mypick (mypick (SOME z. z \notin AD \varphi i))$
shows $\forall (sub, vp) \in SubsVals mypart. s_at vp = i$
<proof>

lemma *v_at_tabulate*:
assumes $\forall z. v_at (mypick z) = i$
and $mypart = tabulate (sorted_list_of_set (AD \varphi i)) mypick (mypick (SOME z. z \notin AD \varphi i))$
shows $\forall (sub, vp) \in SubsVals mypart. v_at vp = i$
<proof>

lemma *s_check_tabulate*:
assumes $future_bounded \varphi$
and $\forall z. s_at (mypick z) = i$
and $\forall z. s_check (v(x:=z)) \varphi (mypick z)$
and $mypart = tabulate (sorted_list_of_set (AD \varphi i)) mypick (mypick (SOME z. z \notin AD \varphi i))$
shows $\forall (sub, vp) \in SubsVals mypart. \forall z \in sub. s_check (v(x := z)) \varphi vp$
<proof>

lemma *v_check_tabulate*:

assumes *future_bounded* φ
and $\forall z. v_at (mypick\ z) = i$
and $\forall z. v_check (v(x:=z))\ \varphi (mypick\ z)$
and $mypart = tabulate\ (sorted_list_of_set\ (AD\ \varphi\ i))\ mypick\ (mypick\ (SOME\ z. z \notin AD\ \varphi\ i))$
shows $\forall (sub, vp) \in SubsVals\ mypart. \forall z \in sub. v_check (v(x := z))\ \varphi\ vp$
<proof>

lemma *s_at_part_hd_tabulate*:
assumes *future_bounded* φ
and $\forall z. s_at (f\ z) = i$
and $mypart = tabulate\ (sorted_list_of_set\ (AD\ \varphi\ i))\ f\ (f\ (SOME\ z. z \notin AD\ \varphi\ i))$
shows $s_at (part_hd\ mypart) = i$
<proof>

lemma *v_at_part_hd_tabulate*:
assumes *future_bounded* φ
and $\forall z. v_at (f\ z) = i$
and $mypart = tabulate\ (sorted_list_of_set\ (AD\ \varphi\ i))\ f\ (f\ (SOME\ z. z \notin AD\ \varphi\ i))$
shows $v_at (part_hd\ mypart) = i$
<proof>

lemma *check_completeness_aux*:
 $(SAT\ \sigma\ v\ i\ \varphi \longrightarrow future_bounded\ \varphi \longrightarrow (\exists\ sp. s_at\ sp = i \wedge s_check\ v\ \varphi\ sp)) \wedge$
 $(VIO\ \sigma\ v\ i\ \varphi \longrightarrow future_bounded\ \varphi \longrightarrow (\exists\ vp. v_at\ vp = i \wedge v_check\ v\ \varphi\ vp))$
<proof>

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

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

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

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

end

9.7 Lifting the Checker to PDTs

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

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

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

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>

lemma *check_all_aux_check_one*: $\forall x. \text{vs } x \neq \{\} \implies \text{distinct_paths } e \implies (\forall x \in \text{vars } e. \text{vs } x = \text{UNIV}) \implies$
 $\text{check_all_aux } \sigma \text{vs } \varphi e \longleftrightarrow (\forall v \in \text{compatible_vals } (\text{fv } \varphi) \text{vs}. \text{check_one } \sigma v \varphi e)$
 <proof>

definition *check_all* :: $(\text{'n}, \text{'d} :: \{\text{default}, \text{linorder}\}) \text{ trace} \Rightarrow (\text{'n}, \text{'d}) \text{ formula} \Rightarrow (\text{'n}, \text{'d}) \text{ expl} \Rightarrow \text{bool}$
where

check_all $\sigma \varphi e = (\text{distinct_paths } e \wedge \text{check_all_aux } \sigma (\lambda_. \text{UNIV}) \varphi e)$

lemma *check_one_alt*: $\text{check_one } \sigma v \varphi e = \text{p_check } \sigma v \varphi (\text{eval_pdt } v e)$
 <proof>

lemma *check_all_alt*: $\text{check_all } \sigma \varphi e = (\text{distinct_paths } e \wedge (\forall v. \text{p_check } \sigma v \varphi (\text{eval_pdt } v e)))$
 <proof>

fun *pdt_at where*

pdt_at i (*Leaf* l) = (*p_at* $l = i$)

| *pdt_at* i (*Node* x *part*) = $(\forall \text{pdt} \in \text{Vals } \text{part}. \text{pdt_at } i \text{pdt})$

lemma *pdt_at_p_at_eval_pdt*: $\text{pdt_at } i e \implies \text{p_at } (\text{eval_pdt } v e) = i$
 <proof>

lemma *check_all_completeness_aux*:

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

shows $\text{set } \text{vs} \subseteq \text{fv } \varphi \implies \text{future_bounded } \varphi \implies \text{distinct } \text{vs} \implies$

$\exists e. \text{pdt_at } i e \wedge \text{vars_order } \text{vs } e \wedge (\forall v. (\forall x. x \notin \text{set } \text{vs} \longrightarrow v x = w x) \longrightarrow \text{p_check } \sigma v \varphi (\text{eval_pdt } v e))$

<proof>

lemma *check_all_completeness*:

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

assumes *future_bounded* φ

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

<proof>

lemma *check_all_soundness_aux*: $\text{check_all } \sigma \varphi e \implies p = \text{eval_pdt } v e \implies \text{isl } p \longleftrightarrow \text{sat } \sigma v (\text{p_at } p) \varphi$
 <proof>

lemma *check_all_soundness*: $\text{check_all } \sigma \varphi e \implies \text{pdt_at } i e \implies \text{isl } (\text{eval_pdt } v e) \longleftrightarrow \text{sat } \sigma v i \varphi$
 <proof>

unbundle *no MFOTL_syntax*

10 Type of Events

10.1 Code Adaptation for 8-bit strings

typedef *string8* = *UNIV* :: *char list set* <proof>

setup_lifting *type_definition_string8*

lift_definition *empty_string* :: *string8* **is** [] <proof>

lift_definition *string8_literal* :: *String.literal* \Rightarrow *string8* **is** *String.explode* <proof>

```

lift_definition literal_string8 :: string8 ⇒ String.literal is String.Abs_literal ⟨proof⟩
declare [[coercion string8_literal]]

instantiation string8 :: {equal, linorder}
begin

lift_definition equal_string8 :: string8 ⇒ string8 ⇒ bool is HOL.equal ⟨proof⟩
lift_definition less_eq_string8 :: string8 ⇒ string8 ⇒ bool is ord_class.lexordp_eq ⟨proof⟩
lift_definition less_string8 :: string8 ⇒ string8 ⇒ bool is ord_class.lexordp ⟨proof⟩

instance ⟨proof⟩

end

lifting_forget string8.lifting

declare [[code drop: literal_string8 string8_literal HOL.equal :: string8 ⇒ _
(≤) :: string8 ⇒ _ (<) :: string8 ⇒ _
Code_Evaluation.term_of :: string8 ⇒ _]]

code_printing
type_constructor string8 → (OCaml) string
| constant HOL.equal :: string8 ⇒ string8 ⇒ bool → (OCaml) Stdlib.(=)
| constant (≤) :: string8 ⇒ string8 ⇒ bool → (OCaml) Stdlib.(≤)
| constant (<) :: string8 ⇒ string8 ⇒ bool → (OCaml) Stdlib.<
| constant empty_string :: string8 → (OCaml)
| constant string8_literal :: String.literal ⇒ string8 → (OCaml) id
| constant literal_string8 :: string8 ⇒ String.literal → (OCaml) id

⟨ML⟩

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

⟨ML⟩

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  $b \neq 0 \implies \text{div\_to\_zero } a \ b * b + \text{mod\_to\_zero } a \ b = a$ 
  <proof>

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  $x \leq$  EInt  $y \longleftrightarrow x \leq y$ 
| EString  $x \leq$  EString  $y \longleftrightarrow x \leq y$ 
| EInt  $\_ \leq$  EString  $\_ \longleftrightarrow$  True
| ( $\_ ::$  event_data)  $\leq$   $\_ \longleftrightarrow$  False

definition less_event_data :: event_data  $\Rightarrow$  event_data  $\Rightarrow$  bool where
  less_event_data  $x \ y \longleftrightarrow x \leq y \wedge \neg y \leq x$ 

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  $x \text{ div } EInt \ y =$  EInt  $(\text{div\_to\_zero } x \ y)$ 
| ( $\_ ::$  event_data)  $\text{div } \_ =$  undefined

fun modulo_event_data where
  EInt  $x \text{ mod } EInt \ y =$  EInt  $(\text{mod\_to\_zero } x \ y)$ 
| ( $\_ ::$  event_data)  $\text{mod } \_ =$  undefined

instance <proof>

end

lemma infinite_UNIV_event_data:
   $\neg$ finite (UNIV :: event_data set)
  <proof>

primrec integer_of_event_data :: event_data  $\Rightarrow$  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⟩

end

instantiation event_data :: linorder begin
instance
⟨proof⟩

end

```

11 Code Generation

11.1 Type Class Instances

```

class universe =
  fixes universe :: 'a list option
  assumes infinite: universe = None  $\implies$  infinite (UNIV :: 'a set)
  and finite: universe = Some xs  $\implies$  distinct xs  $\wedge$  set xs = UNIV
begin

lemma finite_coset: finite (List.coset (xs :: 'a list)) = (case universe of None  $\implies$  False | _  $\implies$  True)
  ⟨proof⟩

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 ⟨proof⟩
end
instantiation char :: universe begin
definition universe_char :: char list option where universe_char = Some (map char_of [0::nat.. $256$ ])
instance ⟨proof⟩
end
instantiation nat :: universe begin
definition universe_nat :: nat list option where universe_nat = None
instance ⟨proof⟩
end
instantiation list :: (type) universe begin
definition universe_list :: 'a list list option where universe_list = None
instance ⟨proof⟩
end
instantiation String.literal :: universe begin
definition universe_literal :: String.literal list option where universe_literal = None
instance ⟨proof⟩
end
instantiation string8 :: universe begin
definition universe_string8 :: string8 list option where universe_string8 = None
lemma UNIV_string8: UNIV = Abs_string8 ' UNIV
  ⟨proof⟩
instance ⟨proof⟩
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) ⇒ Some (List.product xs ys) | _ ⇒ None)
instance ⟨proof⟩
end
instantiation sum :: (universe, universe) universe begin
definition universe_sum :: ('a + 'b) list option where universe_sum =
    (case (universe, universe) of (Some xs, Some ys) ⇒ Some (map Inl xs @ map Inr ys) | _ ⇒ None)
instance ⟨proof⟩
end
instantiation option :: (universe) universe begin
definition universe_option = (case universe of Some xs ⇒ Some (None # map Some xs) | _ ⇒ None)
instance ⟨proof⟩
end
instantiation fun :: (universe, universe) universe begin
definition universe_fun :: ('a ⇒ 'b) list option where universe_fun =
    (case (universe, universe) of
      (Some xs, Some ys) ⇒ Some (map (λzs. the ∘ map_of (zip xs zs)) (List.n_lists (length xs) ys))
    | (_, Some [x]) ⇒ Some [λ_. x]
    | _ ⇒ None)
instance
  ⟨proof⟩
end
instantiation event_data :: universe begin
definition universe_event_data :: event_data list option where universe_event_data = None
instance ⟨proof⟩
end

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

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

instance event_data :: equal ⟨proof⟩

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

instantiation event_data :: card_UNIV begin
definition finite_UNIV = Phantom(event_data) False
definition card_UNIV = Phantom(event_data) 0
instance ⟨proof⟩
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)

```

$| \text{progress } \sigma \text{ (Formula.And } \varphi \ \psi) \ j = \min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $| \text{progress } \sigma \text{ (Formula.Imp } \varphi \ \psi) \ j = \min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $| \text{progress } \sigma \text{ (Formula.Iff } \varphi \ \psi) \ j = \min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $| \text{progress } \sigma \text{ (Formula.Exists } _ \ \varphi) \ j = \text{progress } \sigma \ \varphi \ j$
 $| \text{progress } \sigma \text{ (Formula.Forall } _ \ \varphi) \ j = \text{progress } \sigma \ \varphi \ j$
 $| \text{progress } \sigma \text{ (Formula.Prev } I \ \varphi) \ j = (\text{if } j = 0 \text{ then } 0 \text{ else } \min (\text{Suc } (\text{progress } \sigma \ \varphi \ j)) \ j)$
 $| \text{progress } \sigma \text{ (Formula.Next } I \ \varphi) \ j = \text{progress } \sigma \ \varphi \ j - 1$
 $| \text{progress } \sigma \text{ (Formula.Once } I \ \varphi) \ j = \text{progress } \sigma \ \varphi \ j$
 $| \text{progress } \sigma \text{ (Formula.Historically } I \ \varphi) \ j = \text{progress } \sigma \ \varphi \ j$
 $| \text{progress } \sigma \text{ (Formula.Eventually } I \ \varphi) \ j =$
 $\quad \text{Inf } \{i. \forall k. k < j \wedge k \leq (\text{progress } \sigma \ \varphi \ j) \longrightarrow (\tau \ \sigma \ k - \tau \ \sigma \ i) \leq \text{right } I\}$
 $| \text{progress } \sigma \text{ (Formula.Always } I \ \varphi) \ j =$
 $\quad \text{Inf } \{i. \forall k. k < j \wedge k \leq (\text{progress } \sigma \ \varphi \ j) \longrightarrow (\tau \ \sigma \ k - \tau \ \sigma \ i) \leq \text{right } I\}$
 $| \text{progress } \sigma \text{ (Formula.Since } \varphi \ I \ \psi) \ j = \min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $| \text{progress } \sigma \text{ (Formula.Until } \varphi \ I \ \psi) \ j =$
 $\quad \text{Inf } \{i. \forall k. k < j \wedge k \leq \min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j) \longrightarrow (\tau \ \sigma \ k - \tau \ \sigma \ i) \leq \text{right } I\}$
 $| \text{progress } \sigma \text{ (Formula.MatchP } I \ r) \ j = \min_regex_default (\text{progress } \sigma) \ r \ j$
 $| \text{progress } \sigma \text{ (Formula.MatchF } I \ r) \ j = \text{Inf } \{i. \forall k. k < j \wedge k \leq \min_regex_default (\text{progress } \sigma) \ r \ j \longrightarrow$
 $\tau \ \sigma \ i + \text{right } I \geq \tau \ \sigma \ k\}$

lemma *Inf_Min*:

fixes $P :: \text{nat} \Rightarrow \text{bool}$

assumes $P \ j$

shows $\text{Inf } (\text{Collect } P) = \text{Min } (\text{Set.filter } P \ \{..j\})$

<proof>

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

$(\text{let } m = \min j \ (\text{Suc } (\text{progress } \sigma \ \varphi \ j)) - 1 \text{ in } \text{Min } (\text{Set.filter } (\lambda i. \text{enat } (\delta \ \sigma \ m \ i) \leq \text{right } I) \ \{..j\}))$

<proof>

lemma *progress_Always_code*: $\text{progress } \sigma \text{ (Formula.Always } I \ \varphi) \ j =$

$(\text{let } m = \min j \ (\text{Suc } (\text{progress } \sigma \ \varphi \ j)) - 1 \text{ in } \text{Min } (\text{Set.filter } (\lambda i. \text{enat } (\delta \ \sigma \ m \ i) \leq \text{right } I) \ \{..j\}))$

<proof>

lemma *progress_Until_code*: $\text{progress } \sigma \text{ (Formula.Until } \varphi \ I \ \psi) \ j =$

$(\text{let } m = \min j \ (\text{Suc } (\min (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j))) - 1 \text{ in } \text{Min } (\text{Set.filter } (\lambda i. \text{enat } (\delta \ \sigma \ m \ i) \leq \text{right } I) \ \{..j\}))$

<proof>

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*: $(x \ \#\# \ s) \ !! \ n = (\text{case } n \text{ of } 0 \Rightarrow x \mid \text{Suc } m \Rightarrow s \ !! \ m)$

<proof>

lemma *extend_is_stream*:

assumes *sorted* $(\text{map } \text{snd } \text{list})$

and $\bigwedge x. x \in \text{set } \text{list} \Longrightarrow \text{snd } x \leq m$

and $\bigwedge x. x \in \text{set } \text{list} \Longrightarrow \text{finite } (\text{fst } x)$

shows *ssorted* $(\text{smap } \text{snd } (\text{list } @- \text{smap } (\lambda n. (\{\}, n + m)) \ \text{nats})) \wedge$

sincreasing $(\text{smap } \text{snd } (\text{list } @- \text{smap } (\lambda n. (\{\}, n + m)) \ \text{nats})) \wedge$

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

<proof>

typedef *'a trace_mapping* = $\{(n, m, t) :: (\text{nat} \times \text{nat} \times (\text{nat}, \text{'a set} \times \text{nat}) \text{ mapping}) \mid$
 $n \ m \ t. \text{Mapping.keys } t = \{..<n\} \wedge$

$sorted (map (snd \circ (the \circ Mapping.lookup\ t)) [0..<n]) \wedge$
 $(case\ n\ of\ 0 \Rightarrow True \mid Suc\ n' \Rightarrow (case\ Mapping.lookup\ t\ n'\ of\ Some\ (X',\ t') \Rightarrow t' \leq m \mid None \Rightarrow False))$
 \wedge
 $(\forall\ n' < n.\ case\ Mapping.lookup\ t\ n'\ of\ Some\ (X',\ t') \Rightarrow finite\ X' \mid None \Rightarrow False)$
 $\langle proof \rangle$

setup_lifting *type_definition_trace_mapping*

lemma *lookup_bulkload_Some*: $i < length\ list \implies$
 $Mapping.lookup\ (Mapping.bulkload\ list)\ i = Some\ (list\ !\ i)$
 $\langle proof \rangle$

lift_definition *trace_mapping_of_list* :: $('a\ set \times nat)\ list \Rightarrow 'a\ trace_mapping$ **is**
 $\lambda xs.\ if\ sorted\ (map\ snd\ xs) \wedge (\forall x \in set\ xs.\ finite\ (fst\ x))\ then\ (if\ xs = []\ then\ (0,\ 0,\ Mapping.empty)$
 $else\ (length\ xs,\ snd\ (last\ xs),\ Mapping.bulkload\ xs))$
 $else\ (0,\ 0,\ Mapping.empty)$
 $\langle proof \rangle$

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

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

code_datatype *Trace_Mapping*

definition *trace_of_list* $xs = Trace_Mapping\ (trace_mapping_of_list\ xs)$

lemma $\Gamma_rbt_code[code]$: $\Gamma\ (Trace_Mapping\ t)\ i = fst\ (trace_mapping_nth\ t\ i)$
 $\langle proof \rangle$

lemma $\tau_rbt_code[code]$: $\tau\ (Trace_Mapping\ t)\ i = snd\ (trace_mapping_nth\ t\ i)$
 $\langle proof \rangle$

lemma *trace_mapping_of_list_sound*: $sorted\ (map\ snd\ xs) \wedge (\forall x \in set\ xs.\ finite\ (fst\ x)) \implies i < length\ xs \implies$
 $xs\ !\ i = (\Gamma\ (trace_of_list\ xs)\ i,\ \tau\ (trace_of_list\ xs)\ i)$
 $\langle proof \rangle$

11.4 Auxiliary results

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

lemma *sum_proofs_empty[simp]*: $sum_proofs\ f\ [] = 0$
 $\langle proof \rangle$

lemma *sum_proofs_fundef_cong[fundef_cong]*: $(\bigwedge x.\ x \in set\ xs \implies f\ x = f'\ x) \implies$
 $sum_proofs\ f\ xs = sum_proofs\ f'\ xs$
 $\langle proof \rangle$

lemma *sum_proofs_Cons*:
fixes $f :: 'a \Rightarrow nat$
shows $sum_proofs\ f\ (p \# qs) = f\ p + sum_proofs\ f\ qs$
 $\langle proof \rangle$

lemma *sum_proofs_app*:
fixes $f :: 'a \Rightarrow nat$

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

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**
 $s_pred \text{ (STT _)} = 1$
 $| s_pred \text{ (SEq_Const _ _ _)} = 1$
 $| s_pred \text{ (SPred _ r _)} = w \text{ r}$
 $| s_pred \text{ (SNeg vp)} = (v_pred \text{ vp}) + 1$
 $| s_pred \text{ (SOOrL sp1)} = (s_pred \text{ sp1}) + 1$
 $| s_pred \text{ (SOOrR sp2)} = (s_pred \text{ sp2}) + 1$
 $| s_pred \text{ (SAnd sp1 sp2)} = (s_pred \text{ sp1}) + (s_pred \text{ sp2}) + 1$
 $| s_pred \text{ (SImpL vp1)} = (v_pred \text{ vp1}) + 1$
 $| s_pred \text{ (SImpR sp2)} = (s_pred \text{ sp2}) + 1$
 $| s_pred \text{ (SIffSS sp1 sp2)} = (s_pred \text{ sp1}) + (s_pred \text{ sp2}) + 1$
 $| s_pred \text{ (SIffVV vp1 vp2)} = (v_pred \text{ vp1}) + (v_pred \text{ vp2}) + 1$
 $| s_pred \text{ (SExists _ _ sp)} = (s_pred \text{ sp}) + 1$
 $| s_pred \text{ (SForall _ _ part)} = (\text{sum_proofs } s_pred \text{ (vals part)}) + 1$
 $| s_pred \text{ (SPrev sp)} = (s_pred \text{ sp}) + 1$
 $| s_pred \text{ (SNext sp)} = (s_pred \text{ sp}) + 1$
 $| s_pred \text{ (SOnce _ sp)} = (s_pred \text{ sp}) + 1$
 $| s_pred \text{ (SEventually _ sp)} = (s_pred \text{ sp}) + 1$
 $| s_pred \text{ (SHistorically _ _ sps)} = (\text{sum_proofs } s_pred \text{ sps}) + 1$
 $| s_pred \text{ (SHistoricallyOut _)} = 1$
 $| s_pred \text{ (SAlways _ _ sps)} = (\text{sum_proofs } s_pred \text{ sps}) + 1$
 $| s_pred \text{ (SSince sp2 sp1s)} = (\text{sum_proofs } s_pred \text{ (sp2 \# sp1s)}) + 1$
 $| s_pred \text{ (SUntil sp1s sp2)} = (\text{sum_proofs } s_pred \text{ (sp1s @ [sp2])}) + 1$
 $v_pred \text{ (VFF _)} = 1$
 $| v_pred \text{ (VEq_Const _ _ _)} = 1$
 $| v_pred \text{ (VPred _ r _)} = w \text{ r}$
 $| v_pred \text{ (VNeg sp)} = (s_pred \text{ sp}) + 1$
 $| v_pred \text{ (VOr vp1 vp2)} = ((v_pred \text{ vp1}) + (v_pred \text{ vp2})) + 1$
 $| v_pred \text{ (VAndL vp1)} = (v_pred \text{ vp1}) + 1$
 $| v_pred \text{ (VAndR vp2)} = (v_pred \text{ vp2}) + 1$
 $| v_pred \text{ (VImp sp1 vp2)} = ((s_pred \text{ sp1}) + (v_pred \text{ vp2})) + 1$
 $| v_pred \text{ (VIffSV sp1 vp2)} = ((s_pred \text{ sp1}) + (v_pred \text{ vp2})) + 1$
 $| v_pred \text{ (VIffVS vp1 sp2)} = ((v_pred \text{ vp1}) + (s_pred \text{ sp2})) + 1$
 $| v_pred \text{ (VExists _ _ part)} = (\text{sum_proofs } v_pred \text{ (vals part)}) + 1$
 $| v_pred \text{ (VForall _ _ vp)} = (v_pred \text{ vp}) + 1$
 $| v_pred \text{ (VPrev vp)} = (v_pred \text{ vp}) + 1$
 $| v_pred \text{ (VPrevZ)} = 1$
 $| v_pred \text{ (VPrevOutL _)} = 1$
 $| v_pred \text{ (VPrevOutR _)} = 1$
 $| v_pred \text{ (VNext vp)} = (v_pred \text{ vp}) + 1$
 $| v_pred \text{ (VNextOutL _)} = 1$
 $| v_pred \text{ (VNextOutR _)} = 1$
 $| v_pred \text{ (VOnceOut _)} = 1$
 $| v_pred \text{ (VOnce _ _ vps)} = (\text{sum_proofs } v_pred \text{ vps}) + 1$
 $| v_pred \text{ (VEventually _ _ vps)} = (\text{sum_proofs } v_pred \text{ vps}) + 1$
 $| v_pred \text{ (VHistorically _ vp)} = (v_pred \text{ vp}) + 1$
 $| v_pred \text{ (VAlways _ vp)} = (v_pred \text{ vp}) + 1$
 $| v_pred \text{ (VSinceOut _)} = 1$
 $| v_pred \text{ (VSince _ vp1 vp2s)} = (\text{sum_proofs } v_pred \text{ (vp1 \# vp2s)}) + 1$
 $| v_pred \text{ (VSinceInf _ _ vp2s)} = (\text{sum_proofs } v_pred \text{ vp2s}) + 1$

| $v_pred (VUntil_vp2s\ vp1) = (sum_proofs\ v_pred\ (vp2s\ @\ [vp1])) + 1$
| $v_pred (VUntilInf_ _ _ vp2s) = (sum_proofs\ v_pred\ vp2s) + 1$
⟨proof⟩

termination

⟨proof⟩

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

$p_pred = case_sum\ s_pred\ v_pred$

end

11.5 v_check_exec setup

lemma $ETP_minus_le_iff: ETP\ \sigma\ (\tau\ \sigma\ i - n) \leq j \longleftrightarrow \delta\ \sigma\ i\ j \leq n$
⟨proof⟩

lemma $ETP_minus_gt_iff: j < ETP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow \delta\ \sigma\ i\ j > n$
⟨proof⟩

lemma $nat_le_iff_less:$

fixes $n :: nat$

shows $(j \leq n) \longleftrightarrow (j = 0 \vee j - 1 < n)$

⟨proof⟩

lemma $ETP_minus_eq_iff: j = ETP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow ((j = 0 \vee n < \delta\ \sigma\ i\ (j - 1)) \wedge \delta\ \sigma\ i\ j \leq n)$
⟨proof⟩

lemma $LTP_minus_ge_iff: \tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i \implies j \leq LTP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow$
 $(case\ n\ of\ 0 \Rightarrow \delta\ \sigma\ j\ i = 0 \mid _ \Rightarrow j \leq i \wedge \delta\ \sigma\ i\ j \geq n)$
⟨proof⟩

lemma $LTP_plus_ge_iff: j \leq LTP\ \sigma\ (\tau\ \sigma\ i + n) \longleftrightarrow \delta\ \sigma\ j\ i \leq n$
⟨proof⟩

lemma $LTP_minus_lt_iff:$

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⟩

lemma $LTP_minus_lt_iff:$

assumes $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i$

shows $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j \longleftrightarrow (if\ \neg j \leq i \wedge n = 0\ then\ \delta\ \sigma\ j\ i > 0\ else\ \delta\ \sigma\ i\ j < n)$

⟨proof⟩

lemma $LTP_minus_eq_iff:$

assumes $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i$

shows $j = LTP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow$

$(case\ n\ of\ 0 \Rightarrow i \leq j \wedge \delta\ \sigma\ j\ i = 0 \wedge \delta\ \sigma\ (Suc\ j)\ j > 0$

$\mid _ \Rightarrow j \leq i \wedge n \leq \delta\ \sigma\ i\ j \wedge \delta\ \sigma\ i\ (Suc\ j) < n)$

⟨proof⟩

lemma $LTP_plus_eq_iff:$

shows $j = LTP\ \sigma\ (\tau\ \sigma\ i + n) \longleftrightarrow (\delta\ \sigma\ j\ i \leq n \wedge \delta\ \sigma\ (Suc\ j)\ i > n)$

⟨proof⟩

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

⟨proof⟩

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

lemma $check_upt_l_cong$:

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

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

$\langle proof \rangle$

lemma $check_upt_LTP_p_eq$:

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

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

$\langle proof \rangle$

declare $v_check_exec_simps\ [code]\ s_check_exec_simps\ [code]$

lemma $v_check_exec_Once_code[code]$: $v_check_exec\ \sigma\ vs\ (Formula.Once\ I\ \varphi)\ vp = (case\ vp\ of$
 $VOnce\ i\ li\ vps \Rightarrow$

$(case\ right\ I\ of\ \infty \Rightarrow li = 0 \mid enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b))$

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

$\wedge check_upt_LTP_p\ \sigma\ I\ li\ (map\ v_at\ vps)\ i \wedge Ball\ (set\ vps)\ (v_check_exec\ \sigma\ vs\ \varphi)$

$\mid VOnceOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$

$\mid _ \Rightarrow False$)

$\langle proof \rangle$

lemma $s_check_exec_Historically_code[code]$: $s_check_exec\ \sigma\ vs\ (Formula.Historically\ I\ \varphi)\ vp = (case$
 $vp\ of$

$SHistorically\ i\ li\ vps \Rightarrow$

$(case\ right\ I\ of\ \infty \Rightarrow li = 0 \mid enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b))$

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

$\wedge check_upt_LTP_p\ \sigma\ I\ li\ (map\ s_at\ vps)\ i \wedge Ball\ (set\ vps)\ (s_check_exec\ \sigma\ vs\ \varphi)$

$\mid SHistoricallyOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$

$\mid _ \Rightarrow False$)

$\langle proof \rangle$

lemma $v_check_exec_Since_code[code]$: $v_check_exec\ \sigma\ vs\ (Formula.Since\ \varphi\ I\ \psi)\ vp = (case\ vp\ of$
 $VSince\ i\ vp1\ vp2s \Rightarrow$

$let\ j = v_at\ vp1\ in$

$(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow \delta\ \sigma\ i\ j \leq b) \wedge j \leq i$

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

$\wedge check_upt_LTP_p\ \sigma\ I\ j\ (map\ v_at\ vp2s)\ i$

$\wedge v_check_exec\ \sigma\ vs\ \varphi\ vp1 \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$

$\mid VSinceInf\ i\ li\ vp2s \Rightarrow$

$(case\ right\ I\ of\ \infty \Rightarrow li = 0 \mid enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b)) \wedge$

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

$check_upt_LTP_p\ \sigma\ I\ li\ (map\ v_at\ vp2s)\ i \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$

$\mid VSinceOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$

$\mid _ \Rightarrow False$)

$\langle proof \rangle$

lemma $ETP_f_le_iff$: $\max\ i\ (ETP\ \sigma\ (\tau\ \sigma\ i + a)) \leq j \longleftrightarrow i \leq j \wedge \delta\ \sigma\ j\ i \geq a$

$\langle proof \rangle$

lemma $ETP_f_ge_iff$: $j \leq \max\ i\ (ETP\ \sigma\ (\tau\ \sigma\ i + n)) \longleftrightarrow (case\ n\ of\ 0 \Rightarrow j \leq i$

| $Suc\ n' \Rightarrow (case\ j\ of\ 0 \Rightarrow True \mid Suc\ j' \Rightarrow \delta\ \sigma\ j'\ i < n)$
 <proof>

definition $check_upt_ETP_f\ \sigma\ I\ i\ xs\ hi \longleftrightarrow (let\ j = Suc\ hi - length\ xs\ in$
 (case xs of $[] \Rightarrow (case\ left\ I\ of\ 0 \Rightarrow Suc\ hi \leq i \mid Suc\ n' \Rightarrow \delta\ \sigma\ hi\ i < left\ I)$
 | $_ \Rightarrow (xs = [j..<Suc\ hi] \wedge$
 (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 $\bigwedge j. \min\ i\ hi \leq j \wedge j \leq \max\ i\ hi \Longrightarrow \tau\ \sigma\ j = \tau\ \sigma'\ j$
shows $check_upt_ETP_f\ \sigma\ I\ i\ xs\ hi = check_upt_ETP_f\ \sigma'\ I\ i\ xs\ hi$
 <proof>

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>

lemma $v_check_exec_Eventually_code[code]$: $v_check_exec\ \sigma\ vs\ (Formula.Eventually\ I\ \varphi)\ vp = (case\ vp\ of$

$VEventually\ i\ hi\ vps \Rightarrow$
 (case $right\ I$ of $\infty \Rightarrow False \mid enat\ b \Rightarrow (\delta\ \sigma\ hi\ i \leq b \wedge b < \delta\ \sigma\ (Suc\ hi)\ i)) \wedge$
 $check_upt_ETP_f\ \sigma\ I\ i\ (map\ v_at\ vps)\ hi \wedge Ball\ (set\ vps)\ (v_check_exec\ \sigma\ vs\ \varphi)$
 | $_ \Rightarrow False$)
 <proof>

lemma $s_check_exec_Always_code[code]$: $s_check_exec\ \sigma\ vs\ (Formula.Always\ I\ \varphi)\ sp = (case\ sp\ of$

$SAlways\ i\ hi\ sps \Rightarrow$
 (case $right\ I$ of $\infty \Rightarrow False \mid enat\ b \Rightarrow (\delta\ \sigma\ hi\ i \leq b \wedge b < \delta\ \sigma\ (Suc\ hi)\ i))$
 $\wedge check_upt_ETP_f\ \sigma\ I\ i\ (map\ s_at\ sps)\ hi \wedge Ball\ (set\ sps)\ (s_check_exec\ \sigma\ vs\ \varphi)$
 | $_ \Rightarrow False$)
 <proof>

lemma $v_check_exec_Until_code[code]$: $v_check_exec\ \sigma\ vs\ (Formula.Until\ \varphi\ I\ \psi)\ vp = (case\ vp\ of$

$VUntil\ i\ vp2s\ vp1 \Rightarrow$
 let $j = v_at\ vp1$ in
 (case $right\ I$ of $\infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP_f\ \sigma\ i\ b$)
 $\wedge i \leq j \wedge check_upt_ETP_f\ \sigma\ I\ i\ (map\ v_at\ vp2s)\ j$
 $\wedge v_check_exec\ \sigma\ vs\ \varphi\ vp1 \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
 | $VUntilInf\ i\ hi\ vp2s \Rightarrow$
 (case $right\ I$ of $\infty \Rightarrow False \mid enat\ b \Rightarrow (\delta\ \sigma\ hi\ i \leq b \wedge b < \delta\ \sigma\ (Suc\ hi)\ i)) \wedge$
 $check_upt_ETP_f\ \sigma\ I\ i\ (map\ v_at\ vp2s)\ hi \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
 | $_ \Rightarrow False$)
 <proof>

11.6 ETP/LTP setup

lemma ETP_aux : $\neg t \leq \tau\ \sigma\ i \Longrightarrow i \leq (LEAST\ i. t \leq \tau\ \sigma\ i)$
 <proof>

function ETP_rec where

$ETP_rec\ \sigma\ t\ i = (if\ \tau\ \sigma\ i \geq t\ then\ i\ else\ ETP_rec\ \sigma\ t\ (i + 1))$
 <proof>

termination

<proof>

lemma ETP_rec_sound : $ETP_rec\ \sigma\ t\ j = (LEAST\ i. i \geq j \wedge t \leq \tau\ \sigma\ i)$
 <proof>

lemma *ETP_code*[code]: $ETP\ \sigma\ t = ETP_rec\ \sigma\ t\ 0$
 ⟨proof⟩

lemma *LTP_aux*:
assumes $\tau\ \sigma\ (Suc\ i) \leq t$
shows $i \leq Max\ \{i.\ \tau\ \sigma\ i \leq t\}$
 ⟨proof⟩

function (*sequential*) *LTP_rec* **where**
 $LTP_rec\ \sigma\ t\ i = (if\ \tau\ \sigma\ (Suc\ i) \leq t\ then\ LTP_rec\ \sigma\ t\ (i + 1)\ else\ i)$
 ⟨proof⟩

termination
 ⟨proof⟩

lemma *LTP_rec_sound*: $LTP_rec\ \sigma\ t\ j = Max\ (\{i.\ i \geq j \wedge (\tau\ \sigma\ i) \leq t\} \cup \{j\})$
 ⟨proof⟩

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* $\sigma\ t\ 0$)
 ⟨proof⟩

lemma *map_part_code*[code]: $Rep_part\ (map_part\ f\ xs) = map\ (map_prod\ id\ f)\ (Rep_part\ xs)$
 ⟨proof⟩

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* $zs \Rightarrow \forall z \in set\ zs.\ z \in set\ xs \vee z \in set\ ys)$
 ⟨proof⟩

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* $zs \Rightarrow \forall z \in set\ zs.\ z \in set\ xs)$
 ⟨proof⟩

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

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 × event_data list) set × nat) list ⇒ (name, event_data) trace **where**

trace_of_list_specialized xs = trace_of_list xs

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

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

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

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

lift_definition *abs_part* :: (event_data set × 'a) list ⇒ (event_data, 'a) part **is**

λxs.

let Ds = map fst xs in

if {} ∈ set Ds

∨ (∃ D ∈ set Ds. ∃ E ∈ set Ds. D ≠ E ∧ D ∩ E ≠ {})

∨ ¬ distinct Ds

∨ (∪ D ∈ set Ds. D) ≠ UNIV then [(UNIV, undefined)] else xs

⟨proof⟩

lemma *rm_code*[code_unfold]: *rm S = Set.filter (λ(i,j). i < j) S*
⟨proof⟩

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*:

$\text{partition_on } X \mathcal{P} \implies \mathcal{P} = \{\} \longleftrightarrow X = \{\}$
 $\text{partition_on } X \mathcal{P} \implies \mathcal{P} \neq \{\} \longleftrightarrow X \neq \{\}$
 <proof>

lemma *wf_part_list_filter_inter:*

defines *inP1 P1 f v1 part2*
 $\equiv \text{List.map_filter } (\lambda(P2, v2). \text{if } P1 \cap P2 \neq \{\} \text{ then Some}(P1 \cap P2, f v1 v2) \text{ else None}) \text{ part2}$
assumes *partition_on X (set (map fst ((P1, v1) # part1)))*
and *partition_on X (set (map fst part2))*
shows *partition_on P1 (set (map fst (inP1 P1 f v1 part2)))*
and *distinct (map fst ((P1, v1) # part1)) \implies distinct (map fst (part2)) \implies*
distinct (map fst (inP1 P1 f v1 part2))
 <proof>

lemma *wf_part_list_filter_minus:*

defines *notinP2 P1 f v1 part2*
 $\equiv \text{List.map_filter } (\lambda(P2, v2). \text{if } P2 - P1 \neq \{\} \text{ then Some}(P2 - P1, v2) \text{ else None}) \text{ part2}$
assumes *partition_on X (set (map fst ((P1, v1) # part1)))*
and *partition_on X (set (map fst part2))*
shows *partition_on (X - P1) (set (map fst (notinP2 P1 f v1 part2)))*
and *distinct (map fst ((P1, v1) # part1)) \implies distinct (map fst (part2)) \implies*
distinct (map fst (notinP2 P1 f v1 part2))
 <proof>

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>

lemma *partition_on_append: partition_on X (set xs) \implies partition_on Y (set ys) \implies X \cap Y = { } \implies*

partition_on (X \cup Y) (set (xs @ ys))
 <proof>

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>

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>

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

<proof>

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*

<proof>

definition *proof_app* :: ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof (**infixl** ⋈ 65) **where**

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

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

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

definition *min_list_wrt* :: (('n, 'd) proof ⇒ ('n, 'd) proof ⇒ bool) ⇒ ('n, 'd) proof list ⇒ ('n, 'd) proof **where**

min_list_wrt r xs = hd [x ← xs. ∀ y ∈ set xs. r x y]

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

do_neg p = (case *p* of
 Inl *sp* ⇒ [Inr (VNeg *sp*)]
 | Inr *vp* ⇒ [Inl (SNeg *vp*)])

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

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

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

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

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

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

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

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

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

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

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

```
do_forall x p_part = (case p_part of
  Inl p => (case p of
    Inl sp => [Inl (SForall x (trivial_part sp))]
  | Inr vp => [Inr (VForall x default vp)])
| Inr part => (if (\forall x \in Vals part. isl x) then
  [Inl (SForall x (map_part projl part))]
  else
  map (\lambda(D,p). map_sum id (VForall x (Min D)) p) (filter (\lambda(_, p). \isl p) (subvals 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) => [Inr (VPrevOutL i)]
| (Inl sp, False) => (if mem t I then [Inl (SPrev sp)] else [Inr (VPrevOutR i)])
| (Inr vp, True) => [Inr (VPrev vp), Inr (VPrevOutL i)]
| (Inr vp, False) => (if mem t I then [Inr (VPrev vp)] else [Inr (VPrev vp), Inr (VPrevOutR i)]))
```

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

```
do_next i I t p = (case (p, t < left I) of
  (Inl _, True) => [Inr (VNextOutL i)]
| (Inl sp, False) => (if mem t I then [Inl (SNext sp)] else [Inr (VNextOutR i)])
| (Inr vp, True) => [Inr (VNext vp), Inr (VNextOutL i)]
| (Inr vp, False) => (if mem t I then [Inr (VNext vp)] else [Inr (VNext vp), Inr (VNextOutR i)]))
```

definition $do_once_base :: nat \Rightarrow nat \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof\ list$ **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 $do_once :: nat \Rightarrow nat \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof \Rightarrow ('n, 'd) proof\ list$ **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') \oplus (Inr vp)]
| (Inr _, False, Inl (SONce _ sp')) => [Inl (SONce i sp')]
```

| (*Inr* _ , *False*, *Inr* (*VOnce* _ *li vps'*)) \Rightarrow [*Inr* (*VOnce* *i li vps'*)]

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

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

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

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

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

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

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

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

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

do_always_base *i a p'* = (*case* (*p'*, *a* = 0) of
 (*Inl* *sp'*, *True*) \Rightarrow [*Inl* (*SAlways* *i i [sp']*)]
 | (*Inr* *vp'*, *True*) \Rightarrow [*Inr* (*VAlways* *i vp'*)]
 | (_ , *False*) \Rightarrow [*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) ⇒ [Inr (VSince i vp1 []), Inr (VSinceInf i i [])]
| (Inr vp1, Inr sp2, True) ⇒ [Inr (VSince i vp1 [sp2]), Inr (VSinceInf i i [sp2])]

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

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

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

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

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

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

fun match :: ('n, 'd) Formula.trm list ⇒ 'd list ⇒ ('n → 'd) option where

match [] [] = Some Map.empty
| match (Formula.Const x # ts) (y # ys) = (if x = y then match ts ys else None)
| match (Formula.Var x # ts) (y # ys) = (case match ts ys of
None ⇒ None
| Some f ⇒ (case f x of
None ⇒ Some (f(x ↦ y))

| Some z ⇒ if y = z then Some f else None))
| match _ _ = None

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

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

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

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

```

    Node x (map_part (λexpl1. apply_pdt2 vs (λpt1 pt2. f pt1 pt2 pt3) expl1 (Node y part2)) part1)
  else if y = w then
    Node y (map_part (λexpl2. apply_pdt2 vs (λpt1 pt2. f pt1 pt2 pt3) (Node x part1) expl2) part2)
  else
    apply_pdt3 vs f (Node x part1) (Node y part2) (Leaf pt3)
| apply_pdt3 (w # vs) f (Node x part1) (Leaf pt2) (Node z part3) =
  (if x = w ∧ z = w then
    Node w (merge_part2 (apply_pdt2 vs (λpt1. f pt1 pt2)) part1 part3)
  else if x = w then
    Node x (map_part (λexpl1. apply_pdt2 vs (λpt1. f pt1 pt2) expl1 (Node z part3)) part1)
  else if z = w then
    Node z (map_part (λexpl3. apply_pdt2 vs (λpt1. f pt1 pt2) (Node x part1) expl3) part3)
  else
    apply_pdt3 vs f (Node x part1) (Leaf pt2) (Node z part3))
| apply_pdt3 (w # vs) f (Node x part1) (Node y part2) (Node z part3) =
  (if x = w ∧ y = w ∧ z = w then
    Node z (merge_part3 (apply_pdt3 vs f) part1 part2 part3)
  else if x = w ∧ y = w then
    Node w (merge_part2 (apply_pdt3 vs (λpt3 pt1 pt2. f pt1 pt2 pt3) (Node z part3)) part1 part2)
  else if x = w ∧ z = w then
    Node w (merge_part2 (apply_pdt3 vs (λpt2 pt1 pt3. f pt1 pt2 pt3) (Node y part2)) part1 part3)
  else if y = w ∧ z = w then
    Node w (merge_part2 (apply_pdt3 vs (λpt1. f pt1) (Node x part1)) part2 part3)
  else if x = w then
    Node x (map_part (λexpl1. apply_pdt3 vs f expl1 (Node y part2) (Node z part3)) part1)
  else if y = w then
    Node y (map_part (λexpl2. apply_pdt3 vs f (Node x part1) expl2 (Node z part3)) part2)
  else if z = w then
    Node z (map_part (λexpl3. apply_pdt3 vs f (Node x part1) (Node y part2) expl3) part3)
  else
    apply_pdt3 vs f (Node x part1) (Node y part2) (Node z part3))
| apply_pdt3 [] _ _ _ = undefined

```

```

fun hide_pdt :: 'n list ⇒ (('n, 'd) proof + ('d, ('n, 'd) proof) part ⇒ ('n, 'd) proof) ⇒ ('n, 'd) expl ⇒
('n, 'd) expl where
  hide_pdt vs f (Leaf pt) = Leaf (f (Inl pt))
| hide_pdt [x] f (Node y part) = Leaf (f (Inr (map_part unleaf part)))
| hide_pdt (x # xs) f (Node y part) =
  (if x = y then
    Node y (map_part (hide_pdt xs f) part)
  else
    hide_pdt xs f (Node y part))
| hide_pdt [] _ _ = undefined

```

context

```

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

```

begin

```

function (sequential) eval :: 'n list ⇒ nat ⇒ ('n, 'd) Formula.formula ⇒ ('n, 'd) expl where
  eval vs i Formula.TT = Leaf (Inl (STT i))
| eval vs i Formula.FF = Leaf (Inr (VFF i))
| eval vs i (Eq_Const x c) = Node x (tabulate [c] (λc. Leaf (Inl (SEq_Const i x c))) (Leaf (Inr
(VEq_Const i x c))))
| eval vs i (Formula.Pred r ts) =
  (pdt_of i r ts (filter (λx. x ∈ Formula.fv (Formula.Pred r ts)) vs) (List.map_filter (match ts) (sorted_list_of_set
(snd ' {rd ∈ Γ σ i. fst rd = r}))))
| eval vs i (Formula.Neg φ) = apply_pdt1 vs (λp. min_list_wrt cmp (do_neg p)) (eval vs i φ)

```

$| \text{eval vs } i \text{ (Formula.Or } \varphi \psi) = \text{apply_pdt2 vs } (\lambda p1 p2. \text{min_list_wrt cmp (do_or } p1 p2)) \text{ (eval vs } i \varphi)$
 $(\text{eval vs } i \psi)$
 $| \text{eval vs } i \text{ (Formula.And } \varphi \psi) = \text{apply_pdt2 vs } (\lambda p1 p2. \text{min_list_wrt cmp (do_and } p1 p2)) \text{ (eval vs } i$
 $\varphi) \text{ (eval vs } i \psi)$
 $| \text{eval vs } i \text{ (Formula.Imp } \varphi \psi) = \text{apply_pdt2 vs } (\lambda p1 p2. \text{min_list_wrt cmp (do_imp } p1 p2)) \text{ (eval vs } i$
 $\varphi) \text{ (eval vs } i \psi)$
 $| \text{eval vs } i \text{ (Formula.Iff } \varphi \psi) = \text{apply_pdt2 vs } (\lambda p1 p2. \text{min_list_wrt cmp (do_iff } p1 p2)) \text{ (eval vs } i \varphi)$
 $(\text{eval vs } i \psi)$
 $| \text{eval vs } i \text{ (Formula.Exists } x \varphi) = \text{hide_pdt (vs @ [x]) } (\lambda p. \text{min_list_wrt cmp (do_exists } x p)) \text{ (eval (vs$
 $@ [x]) } i \varphi)$
 $| \text{eval vs } i \text{ (Formula.Forall } x \varphi) = \text{hide_pdt (vs @ [x]) } (\lambda p. \text{min_list_wrt cmp (do_forall } x p)) \text{ (eval (vs$
 $@ [x]) } i \varphi)$
 $| \text{eval vs } i \text{ (Formula.Prev } I \varphi) = (\text{if } i = 0 \text{ then Leaf (Inr VPrevZ)}$
 $\text{else apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_prev } i I (\Delta \sigma i) p)) \text{ (eval vs$
 $(i-1) \varphi)$
 $| \text{eval vs } i \text{ (Formula.Next } I \varphi) = \text{apply_pdt1 vs } (\lambda l. \text{min_list_wrt cmp (do_next } i I (\Delta \sigma (i+1)) l)) \text{ (eval$
 $\text{vs } (i+1) \varphi)$
 $| \text{eval vs } i \text{ (Formula.Once } I \varphi) =$
 $(\text{if } \tau \sigma i < \tau \sigma 0 + \text{left } I \text{ then Leaf (Inr (VOnceOut } i))$
 $\text{else (let expl = eval vs } i \varphi \text{ in}$
 $(\text{if } i = 0 \text{ then}$
 $\text{apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_once_base } 0 0 p)) \text{ expl}$
 $\text{else (if right } I \geq \text{enat } (\Delta \sigma i) \text{ then}$
 $\text{apply_pdt2 vs } (\lambda p p'. \text{min_list_wrt cmp (do_once } i \text{ (left } I) p p')) \text{ expl}$
 $(\text{eval vs } (i-1) \text{ (Formula.Once (subtract } (\Delta \sigma i) I) \varphi))$
 $\text{else apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_once_base } i \text{ (left } I) p)) \text{ expl))))$
 $| \text{eval vs } i \text{ (Formula.Historically } I \varphi) =$
 $(\text{if } \tau \sigma i < \tau \sigma 0 + \text{left } I \text{ then Leaf (Inl (SHistoricallyOut } i))$
 $\text{else (let expl = eval vs } i \varphi \text{ in}$
 $(\text{if } i = 0 \text{ then}$
 $\text{apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_historically_base } 0 0 p)) \text{ expl}$
 $\text{else (if right } I \geq \text{enat } (\Delta \sigma i) \text{ then}$
 $\text{apply_pdt2 vs } (\lambda p p'. \text{min_list_wrt cmp (do_historically } i \text{ (left } I) p p')) \text{ expl}$
 $(\text{eval vs } (i-1) \text{ (Formula.Historically (subtract } (\Delta \sigma i) I) \varphi))$
 $\text{else apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_historically_base } i \text{ (left } I) p)) \text{ expl))))$
 $| \text{eval vs } i \text{ (Formula.Eventually } I \varphi) =$
 $(\text{let expl = eval vs } i \varphi \text{ in}$
 $(\text{if right } I = \infty \text{ then undefined}$
 $\text{else (if right } I \geq \text{enat } (\Delta \sigma (i+1)) \text{ then}$
 $\text{apply_pdt2 vs } (\lambda p p'. \text{min_list_wrt cmp (do_eventually } i \text{ (left } I) p p')) \text{ expl}$
 $(\text{eval vs } (i+1) \text{ (Formula.Eventually (subtract } (\Delta \sigma (i+1)) I) \varphi))$
 $\text{else apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_eventually_base } i \text{ (left } I) p)) \text{ expl))))$
 $| \text{eval vs } i \text{ (Formula.Always } I \varphi) =$
 $(\text{let expl = eval vs } i \varphi \text{ in}$
 $(\text{if right } I = \infty \text{ then undefined}$
 $\text{else (if right } I \geq \text{enat } (\Delta \sigma (i+1)) \text{ then}$
 $\text{apply_pdt2 vs } (\lambda p p'. \text{min_list_wrt cmp (do_always } i \text{ (left } I) p p')) \text{ expl}$
 $(\text{eval vs } (i+1) \text{ (Formula.Always (subtract } (\Delta \sigma (i+1)) I) \varphi))$
 $\text{else apply_pdt1 vs } (\lambda p. \text{min_list_wrt cmp (do_always_base } i \text{ (left } I) p)) \text{ expl))))$
 $| \text{eval vs } i \text{ (Formula.Since } \varphi I \psi) =$
 $(\text{if } \tau \sigma i < \tau \sigma 0 + \text{left } I \text{ then Leaf (Inr (VSinceOut } i))$
 $\text{else (let expl1 = eval vs } i \varphi \text{ in}$
 $\text{let expl2 = eval vs } i \psi \text{ in}$
 $(\text{if } i = 0 \text{ then}$
 $\text{apply_pdt2 vs } (\lambda p1 p2. \text{min_list_wrt cmp (do_since_base } 0 0 p1 p2)) \text{ expl1 expl2}$
 $\text{else (if right } I \geq \text{enat } (\Delta \sigma i) \text{ then}$
 $\text{apply_pdt3 vs } (\lambda p1 p2 p'. \text{min_list_wrt cmp (do_since } i \text{ (left } I) p1 p2 p')) \text{ expl1 expl2}$
 $(\text{eval vs } (i-1) \text{ (Formula.Since } \varphi \text{ (subtract } (\Delta \sigma i) I) \psi))$

```

else apply_pdt2 vs ( $\lambda p1 p2. \text{min\_list\_wrt\_cmp } (\text{do\_since\_base } i \text{ (left } I \text{) } p1 p2)) \text{ expl1}
\text{expl2})))
| \text{eval vs } i \text{ (Formula.Until } \varphi \text{ } I \text{ } \psi) =
(\text{let expl1} = \text{eval vs } i \text{ } \varphi \text{ in}
\text{let expl2} = \text{eval vs } i \text{ } \psi \text{ in}
(\text{if right } I = \infty \text{ then undefined}
\text{else (if right } I \geq \text{enat } (\Delta \sigma \text{ (} i+1)) \text{ then}
\text{apply\_pdt3 vs } (\lambda p1 p2 p'. \text{min\_list\_wrt\_cmp } (\text{do\_until } i \text{ (left } I \text{) } p1 p2 p')) \text{ expl1 expl2}
(\text{eval vs } (i+1) \text{ (Formula.Until } \varphi \text{ (subtract } (\Delta \sigma \text{ (} i+1)) \text{ } I) \text{ } \psi))}
\text{else apply\_pdt2 vs } (\lambda p1 p2. \text{min\_list\_wrt\_cmp } (\text{do\_until\_base } i \text{ (left } I \text{) } p1 p2)) \text{ expl1 expl2})))
| \text{eval vs } i \text{ (Formula.MatchP } I \text{ } r) = \text{undefined}
| \text{eval vs } i \text{ (Formula.MatchF } I \text{ } r) = \text{undefined}
\langle \text{proof} \rangle$ 
```

fun dist where

```

dist i (Formula.Once _ _) = i
| dist i (Formula.Historically _ _) = i
| dist i (Formula.Eventually I _) = LTP  $\sigma$  (case right I of  $\infty \Rightarrow 0$  |  $\text{enat } b \Rightarrow (\tau \sigma i + b)$ ) - i
| dist i (Formula.Always I _) = LTP  $\sigma$  (case right I of  $\infty \Rightarrow 0$  |  $\text{enat } b \Rightarrow (\tau \sigma i + b)$ ) - i
| dist i (Formula.Since _ _ _) = i
| dist i (Formula.Until _ I _) = LTP  $\sigma$  (case right I of  $\infty \Rightarrow 0$  |  $\text{enat } b \Rightarrow (\tau \sigma i + b)$ ) - i
| dist _ _ = undefined

```

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

$\langle \text{proof} \rangle$

termination eval

$\langle \text{proof} \rangle$

end

end

13 Examples

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

```

monitor  $\pi \varphi = \text{map } (\lambda i. \text{eval } (\text{trace\_of\_list } \pi) (\lambda p q. \text{size } p \leq \text{size } q) (\text{sorted\_list\_of\_set } (fv \varphi)) i \varphi)
[0 ..< \text{length } \pi]$ 

```

definition check :: $((n :: \text{linorder} \times d :: \{\text{default}, \text{linorder}\} \text{ list}) \text{ set} \times \text{nat}) \text{ list} \Rightarrow (n, d) \text{ formula} \Rightarrow \text{bool}$ **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 :: $((\text{string} \times \text{string list}) \text{ set} \times \text{nat}) \text{ list}$ **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*

<proof>

13.2 Finite Domain

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

definition *ord* :: *Domain* \Rightarrow *nat* **where**

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

instantiation *Domain* :: *default* **begin**

definition *default_Domain* = *Default*

instance *<proof>*

end

instantiation *Domain* :: *universe* **begin**

definition *universe_Domain* = *Some* [*Mallory*, *Merlin*, *Martin*, *Alice*, *Bob*, *Charlie*, *David*, *Default*,
R42, *R152*, *R160*, *R163*, *R187*]

instance *<proof>*

end

instantiation *Domain* :: *linorder* **begin**

definition *less_eq_Domain* *d d'* = (*ord d* \leq *ord d'*)

definition *less_Domain* *d d'* = (*ord d* $<$ *ord d'*)

instance *<proof>*

end

definition *fprefix* :: ((*string* \times *Domain list*) *set* \times *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*

⟨*proof*⟩

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.