

A Verified Proof Checker for Metric First-Order Temporal Logic

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

April 19, 2024

Abstract

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

Contents

1	Traces and Trace Prefixes	2
1.1	Infinite Traces	2
1.2	Finite Trace Prefixes	3
1.3	Earliest and Latest Time-Points	5
2	Metric First-Order Temporal Logic	6
2.1	Syntax	6
2.2	Semantics	8
3	Valued Partitions	11
3.1	<i>size</i> setup	12
3.2	Functions on Valued Partitions	13
4	Partitioned Decision Trees	14
5	Proof System	15
5.1	Soundness and Completeness	16
6	Proof Objects	17
7	Auxiliary Lemmas	19
8	Proof Checker	21
8.1	Checker Soundness	23
8.2	Executable Variant of the Checker	25
8.3	Latest Relevant Time-Point	29
8.4	Active Domain	29
8.5	Congruence Modulo Active Domain	30
8.6	Checker Completeness	31

8.7	Lifting the Checker to PDTs	32
9	Type of Events	33
9.1	Code Adaptation for 8-bit strings	33
9.2	Event Parameters	34
10	Code Generation	35
10.1	Type Class Instances	35
10.2	Progress	37
10.3	Trace	38
10.4	Auxiliary results	39
10.5	<i>v_check_exec</i> setup	40
10.6	ETP/LTP setup	43
10.7	Exported functions	44
11	Unverified Explanation-Producing Monitoring Algorithm	45
12	Examples	54
12.1	Infinite Domain	54
12.2	Finite Domain	54

1 Traces and Trace Prefixes

1.1 Infinite Traces

coinductive *sorted* :: 'a :: linorder stream \Rightarrow bool **where**
shd $s \leq \text{shd } (stl\ s) \implies \text{sorted } (stl\ s) \implies \text{sorted } s$

lemma *sorted_siterate[simp]*: $(\bigwedge n. n \leq f\ n) \implies \text{sorted } (\text{siterate } f\ n)$
<proof>

lemma *sortedD*: $\text{sorted } s \implies s\ !!\ i \leq stl\ s\ !!\ i$
<proof>

lemma *sorted_sdrop*: $\text{sorted } s \implies \text{sorted } (\text{sdrop } i\ s)$
<proof>

lemma *sorted_monoD*: $\text{sorted } s \implies i \leq j \implies s\ !!\ i \leq s\ !!\ j$
<proof>

lemma *sorted_stake*: $\text{sorted } s \implies \text{sorted } (\text{stake } i\ s)$
<proof>

lemma *sorted_monoI*: $\forall i\ j. i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j \implies \text{sorted } s$
<proof>

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

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

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

lemma *sincreasingI*: $(\bigwedge x. \exists i. x < s\ !!\ i) \implies \text{sincreasing } s$
<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* = $\{s :: ('a \text{ set} \times \text{nat}) \text{ stream. } \text{sorted } (\text{smap snd } s) \wedge \text{sincreasing } (\text{smap snd } s) \wedge \text{sfinite } (\text{smap fst } s)\}$

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

<proof>

setup_lifting *type_definition_prefix*

lift_definition *pmap* $\Gamma :: ('a \text{ set} \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ prefix} \Rightarrow 'b \text{ prefix}$ **is**
 $\lambda f. \text{map } (\lambda(x, i). (f x, i))$
 $\langle \text{proof} \rangle$

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

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

lift_definition *pnil* $:: 'a \text{ prefix}$ **is** $[]$ $\langle \text{proof} \rangle$

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

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

instantiation *prefix* $:: (\text{type})$ **order begin**

lift_definition *less_eq_prefix* $:: 'a \text{ prefix} \Rightarrow 'a \text{ prefix} \Rightarrow \text{bool}$ **is**
 $\lambda p q. \exists r. q = p @ r \langle \text{proof} \rangle$

definition *less_prefix* $:: 'a \text{ prefix} \Rightarrow 'a \text{ prefix} \Rightarrow \text{bool}$ **where**
 $\text{less_prefix } x y = (x \leq y \wedge \neg y \leq x)$

instance
 $\langle \text{proof} \rangle$

end

lemma *psnoc_inject*[*simp*]:
 $\text{last_ts } p \leq \text{snd } x \Longrightarrow \text{last_ts } q \leq \text{snd } y \Longrightarrow \text{psnoc } p x = \text{psnoc } q y \longleftrightarrow (p = q \wedge x = y)$
 $\langle \text{proof} \rangle$

lift_definition *prefix_of* $:: 'a \text{ prefix} \Rightarrow 'a \text{ trace} \Rightarrow \text{bool}$ **is** $\lambda p s. \text{stake } (\text{length } p) s = p$ $\langle \text{proof} \rangle$

lemma *prefix_of_pnil*[*simp*]: *prefix_of* *pnil* σ
 $\langle \text{proof} \rangle$

lemma *plen_pnil*[*simp*]: *plen* *pnil* $= 0$
 $\langle \text{proof} \rangle$

lemma *plen_mono*: $\pi \leq \pi' \Longrightarrow \text{plen } \pi \leq \text{plen } \pi'$
 $\langle \text{proof} \rangle$

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc* $p x$) $s \Longrightarrow \text{last_ts } p \leq \text{snd } x \Longrightarrow$
 $(\text{prefix_of } p s \Longrightarrow \Gamma s (\text{plen } p) = \text{fst } x \Longrightarrow \tau s (\text{plen } p) = \text{snd } x \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma *le_pnil*[*simp*]: *pnil* $\leq \pi$
 $\langle \text{proof} \rangle$

lift_definition *take_prefix* $:: \text{nat} \Rightarrow 'a \text{ trace} \Rightarrow 'a \text{ prefix}$ **is** *stake*
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *pts_pmap_Γ[simp]*: $pts\ (pmap_Γ\ f\ \pi) = pts\ \pi$
 ⟨proof⟩

1.3 Earliest and Latest Time-Points

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

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

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

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

abbreviation $LTP_p\ \sigma\ i\ I \equiv min\ i\ (LTP\ \sigma\ ((\tau\ \sigma\ i) - left\ I))$

abbreviation $ETP_f\ \sigma\ i\ I \equiv max\ i\ (ETP\ \sigma\ ((\tau\ \sigma\ i) + left\ I))$

abbreviation $LTP_f \sigma i b \equiv LTP \sigma ((\tau \sigma i) + b)$

definition *max_opt where*

$max_opt a b = (case (a,b) of (Some x, Some y) \Rightarrow Some (max x y) | _ \Rightarrow None)$

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

lemma $i_ETP_tau: i \geq ETP \sigma n \longleftrightarrow \tau \sigma i \geq n$
(proof)

lemma $tau_LTP_k:$

assumes $\tau \sigma 0 \leq n$ $LTP \sigma n < k$

shows $\tau \sigma k > n$

(proof)

lemma $i_LTP_tau:$

assumes $n_asm: n \geq \tau \sigma 0$

shows $(i \leq LTP \sigma n \longleftrightarrow \tau \sigma i \leq n)$

(proof)

lemma $ETP_delta: i \geq ETP \sigma (\tau \sigma l + n) \implies \delta \sigma i l \geq n$

(proof)

lemma $ETP_ge: ETP \sigma (\tau \sigma l + n + 1) > l$

(proof)

lemma $i_le_LTPi: i \leq LTP \sigma (\tau \sigma i)$

(proof)

lemma $i_le_LTPi_add: i \leq LTP \sigma (\tau \sigma i + n)$

(proof)

lemma $i_le_LTPi_minus:$

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

shows $LTP \sigma (\tau \sigma i - n) < i$

(proof)

lemma $i_ge_etpi: ETP \sigma (\tau \sigma i) \leq i$

(proof)

2 Metric First-Order Temporal Logic

2.1 Syntax

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

type_synonym $('n, 'a) database = ('n, 'a) event set$

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

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

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

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

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

lemma $in_fv_trm_conv: x \in fv_trm t \longleftrightarrow t = v x$

(proof)

datatype $('n, 'a) formula =$

TT (T)

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

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

fv ($r \dagger ts$)	$= \bigcup (fv_trm \text{ ' set } ts)$
$fv \top$	$= \{\}$
$fv \perp$	$= \{\}$
$fv (x \approx c)$	$= \{x\}$
$fv (\neg_F \varphi)$	$= fv \varphi$
$fv (\varphi \vee_F \psi)$	$= fv \varphi \cup fv \psi$
$fv (\varphi \wedge_F \psi)$	$= fv \varphi \cup fv \psi$
$fv (\varphi \longrightarrow_F \psi)$	$= fv \varphi \cup fv \psi$
$fv (\varphi \longleftrightarrow_F \psi)$	$= fv \varphi \cup fv \psi$
$fv (\exists_F x. \varphi)$	$= fv \varphi - \{x\}$
$fv (\forall_F x. \varphi)$	$= fv \varphi - \{x\}$
$fv (\mathbf{Y} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\mathbf{X} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\mathbf{P} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\mathbf{H} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\mathbf{F} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\mathbf{G} \mathcal{I} \varphi)$	$= fv \varphi$
$fv (\varphi \mathbf{S} \mathcal{I} \psi)$	$= fv \varphi \cup fv \psi$
$fv (\varphi \mathbf{U} \mathcal{I} \psi)$	$= fv \varphi \cup fv \psi$

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

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

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

| $consts (\mathbf{G} I \varphi) = consts \varphi$
| $consts (\varphi \mathbf{S} I \psi) = consts \varphi \cup consts \psi$
| $consts (\varphi \mathbf{U} I \psi) = consts \varphi \cup consts \psi$

lemma $finite_fv_trm[simp]$: $finite (fv_trm t)$
 $\langle proof \rangle$

lemma $finite_fv[simp]$: $finite (fv \varphi)$
 $\langle proof \rangle$

lemma $finite_consts[simp]$: $finite (consts \varphi)$
 $\langle proof \rangle$

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

fun $future_bounded :: ('n, 'a) formula \Rightarrow bool$ **where**
 $future_bounded \top = True$
| $future_bounded \perp = True$
| $future_bounded (_ \dagger _) = True$
| $future_bounded (_ \approx _) = True$
| $future_bounded (\neg_F \varphi) = future_bounded \varphi$
| $future_bounded (\varphi \vee_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)$
| $future_bounded (\varphi \wedge_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)$
| $future_bounded (\varphi \longrightarrow_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)$
| $future_bounded (\varphi \longleftarrow_F \psi) = (future_bounded \varphi \wedge future_bounded \psi)$
| $future_bounded (\exists_{F_} \varphi) = future_bounded \varphi$
| $future_bounded (\forall_{F_} \varphi) = future_bounded \varphi$
| $future_bounded (\mathbf{Y} I \varphi) = future_bounded \varphi$
| $future_bounded (\mathbf{X} I \varphi) = future_bounded \varphi$
| $future_bounded (\mathbf{P} I \varphi) = future_bounded \varphi$
| $future_bounded (\mathbf{H} I \varphi) = future_bounded \varphi$
| $future_bounded (\mathbf{F} I \varphi) = (future_bounded \varphi \wedge right I \neq \infty)$
| $future_bounded (\mathbf{G} I \varphi) = (future_bounded \varphi \wedge right I \neq \infty)$
| $future_bounded (\varphi \mathbf{S} I \psi) = (future_bounded \varphi \wedge future_bounded \psi)$
| $future_bounded (\varphi \mathbf{U} I \psi) = (future_bounded \varphi \wedge future_bounded \psi \wedge right I \neq \infty)$

2.2 Semantics

primrec $eval_trm :: ('n, 'a) env \Rightarrow ('n, 'a) trm \Rightarrow 'a(___) [70,89] 89$ **where**
 $eval_trm v (\mathbf{v} x) = v x$
| $eval_trm v (\mathbf{c} x) = x$

lemma $eval_trm_fv_cong$: $\forall x \in fv_trm t. v x = v' x \Longrightarrow v[[t]] = v'[[t]]$
 $\langle proof \rangle$

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

lemma $eval_trms_fv_cong$:
 $\forall t \in set ts. \forall x \in fv_trm t. v x = v' x \Longrightarrow v[[ts]] = v'[[ts]]$
 $\langle proof \rangle$

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

definition $eval_trms_set :: ('n, 'a) envset \Rightarrow ('n, 'a) trm\ list \Rightarrow (('n, 'a) trm \times 'a\ set)\ list\ (_ \llbracket _ \rrbracket)$
[70,89] 89)

where $eval_trms_set\ vs\ ts = map\ (eval_trm_set\ vs)\ ts$

lemma $eval_trms_set_Nil: vs \llbracket [] \rrbracket = []$
<proof>

lemma $eval_trms_set_Cons:$
 $vs \llbracket (t \# ts) \rrbracket = vs \llbracket t \rrbracket \# vs \llbracket ts \rrbracket$
<proof>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

bundle *MFOTL_no_notation* **begin**

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

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

For subscripts type “backslash” followed by “sub”

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

```

no_notation eval_trm (⟦_⟧ [70,89] 89)
  and eval_trms (⟦_⟧ [70,89] 89)
  and eval_trm_set (⟦_⟧ [70,89] 89)
  and eval_trms_set (⟦_⟧ [70,89] 89)
  and sat (⟦_, _, _⟧ ≡_ [56, 56, 56, 56] 55)
  and Interval.interval (⟦_, _⟧)

```

end

bundle MFOTL_notation begin

```

notation Var (v)
  and Const (c)

```

```

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

```

```

notation eval_trm (⟦_⟧ [70,89] 89)
  and eval_trms (⟦_⟧ [70,89] 89)
  and eval_trm_set (⟦_⟧ [70,89] 89)
  and eval_trms_set (⟦_⟧ [70,89] 89)
  and sat (⟦_, _, _⟧ ≡_ [56, 56, 56, 56] 55)
  and Interval.interval (⟦_, _⟧)

```

end

unbundle MFOTL_no_notation

3 Valued Partitions

lemma part_list_set_eq_aux1:

assumes

$\forall i < \text{length } xs. \forall j < \text{length } xs. i \neq j \longrightarrow \text{fst } (xs ! i) \cap \text{fst } (xs ! j) = \{\}$
 $\{\} \notin \text{fst } \text{' set } xs$

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

<proof>

lemma part_list_set_eq_aux2:

assumes

```

    disjoint (fst ' set xs)
    distinct (map fst xs)
    i < length xs
    j < length xs
    i ≠ j
shows fst (xs ! i) ∩ fst (xs ! j) = {}
⟨proof⟩

```

```

lemma part_list_eq:
  (⋃ X ∈ fst ' set xs. X) = UNIV
  ∧ (∀ i < length xs. ∀ j < length xs. i ≠ j
    → fst (xs ! i) ∩ fst (xs ! j) = {}) ∧ {} ∉ fst ' set xs
  ←→ partition_on UNIV (set (map fst xs)) ∧ distinct (map fst xs)
⟨proof⟩

```

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

```

typedef ('d, 'a) part = {xs :: ('d set × 'a) list. partition_on UNIV (set (map fst xs)) ∧ distinct (map
fst xs)}
⟨proof⟩

```

setup_lifting type_definition_part

```

lift_bnf (no_warn_wits, no_warn_transfer) (dead 'd, Vals: 'a) part
⟨proof⟩

```

3.1 size setup

```

lift_definition subs :: ('d, 'a) part ⇒ 'd set list is map fst ⟨proof⟩

```

```

lift_definition Subs :: ('d, 'a) part ⇒ 'd set set is set o map fst ⟨proof⟩

```

```

lift_definition vals :: ('d, 'a) part ⇒ 'a list is map snd ⟨proof⟩

```

```

lift_definition SubsVals :: ('d, 'a) part ⇒ ('d set × 'a) set is set ⟨proof⟩

```

```

lift_definition subsvals :: ('d, 'a) part ⇒ ('d set × 'a) list is id ⟨proof⟩

```

```

lift_definition size_part :: ('d ⇒ nat) ⇒ ('a ⇒ nat) ⇒ ('d, 'a) part ⇒ nat is λf g. size_list (λ(x, y).
sum f x + g y) ⟨proof⟩

```

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

```

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

```

instance ⟨proof⟩

end

```

lemma size_part_overloaded_simps[simp]: size x = size (vals x)
⟨proof⟩

```

```

lemma part_size_o_map: inj h ⇒ size_part f g ∘ map_part h = size_part f (g ∘ h)
⟨proof⟩

```

⟨ML⟩

```

lemma is_measure_size_part[measure_function]: is_measure f ⇒ is_measure g ⇒ is_measure (size_part

```

$f\ g)$
 $\langle proof \rangle$

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

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

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

3.2 Functions on Valued Partitions

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

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

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

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

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

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

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

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

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

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

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

lemma *lookup_part_from_subvals*: $(D, e) \in \text{set } (\text{subvals } \text{part}) \implies d \in D \implies \text{lookup_part } \text{part } d = e$
 $\langle proof \rangle$

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

lemma *subvals_part_estimation*[*termination_simp*]: $(D, e) \in \text{set } (\text{subvals part}) \implies \text{size } e < \text{Suc } (\text{size_part } (\lambda _. 0) \text{ size part})$
 ⟨*proof*⟩

lemma *size_part_hd_estimation*[*termination_simp*]: $\text{size } (\text{part_hd part}) < \text{Suc } (\text{size_part } (\lambda _. 0) \text{ size part})$
 ⟨*proof*⟩

lemma *size_last_estimation*[*termination_simp*]: $xs \neq [] \implies \text{size } (\text{last } xs) < \text{size_list size } xs$
 ⟨*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*]: $\text{snd } (\text{lookup part } d) = \text{lookup_part part } d$
 ⟨*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*:
 $\text{partition_on UNIV } (\text{set } (\text{map fst part})) \implies \text{distinct } (\text{map fst part}) \implies \exists y. \text{find } (\lambda(D, _). d \in D) \text{ part} = \text{Some } y$
 ⟨*proof*⟩

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

lemma *lookup_subvals*: $\text{lookup part } d \in \text{set } (\text{subvals part})$
 ⟨*proof*⟩

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

4 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**
 $\text{vars_order } vs \ (\text{Leaf } _)$
 $\mid \forall \text{expl} \in \text{Vals part1}. \text{vars_order } vs \ \text{expl} \implies \text{vars_order } (x \# vs) \ (\text{Node } x \ \text{part1})$
 $\mid \text{vars_order } vs \ (\text{Node } x \ \text{part1}) \implies x \neq z \implies \text{vars_order } (z \# vs) \ (\text{Node } x \ \text{part1})$

lemma *vars_order_Node*:
assumes *distinct_xs*
shows $\text{vars_order } xs \ (\text{Node } x \ \text{part}) = (\exists ys \ zs. xs = ys @ x \# zs \wedge (\forall e \in \text{Vals part}. \text{vars_order } zs \ e))$
 ⟨*proof*⟩

fun *distinct_paths* **where**
 $\text{distinct_paths } (\text{Leaf } _) = \text{True}$
 $\mid \text{distinct_paths } (\text{Node } x \ \text{part}) = (\forall e \in \text{Vals part}. x \notin \text{vars } e \wedge \text{distinct_paths } e)$

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

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

lemma *vars_order_vars*: $\text{vars_order } vs \ e \implies \text{vars } e \subseteq \text{set } vs$
<proof>

lemma *vars_order_distinct_paths*: $\text{vars_order } vs \ e \implies \text{distinct } vs \implies \text{distinct_paths } e$
<proof>

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

5 Proof System

unbundle *MFOTL_notation*

context **begin**

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

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

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

5.1 Soundness and Completeness

lemma *not_sat_SinceD*:

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

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

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

<proof>

lemma *not_sat_UntilD*:

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

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

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

<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 *MFOTL_no_notation*

6 Proof Objects

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

```

| VPrevOutL nat
| VPrevOutR nat
| VNext ('n, 'd) vproof
| VNextOutL nat
| VNextOutR nat
| VOnceOut nat
| VOnce nat nat ('n, 'd) vproof list
| VEventually nat nat ('n, 'd) vproof list
| VHistorically nat ('n, 'd) vproof
| VAlways nat ('n, 'd) vproof
| VSinceOut nat
| VSince nat ('n, 'd) vproof ('n, 'd) vproof list
| VSinceInf nat nat ('n, 'd) vproof list
| VUntil nat ('n, 'd) vproof list ('n, 'd) vproof
| VUntilInf nat nat ('n, 'd) vproof list

```

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

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

```

fun s_at :: ('n, 'd) sproof  $\Rightarrow$  nat and
  v_at :: ('n, 'd) vproof  $\Rightarrow$  nat where
  s_at (STT i) = i
| s_at (SPred i _) = i
| s_at (SEq_Const i _) = i
| s_at (SNeg vp) = v_at vp
| s_at (SOrL sp1) = s_at sp1
| s_at (SOrR sp2) = s_at sp2
| s_at (SAnd sp1 _) = s_at sp1
| s_at (SImpL vp1) = v_at vp1
| s_at (SImpR sp2) = s_at sp2
| s_at (SIffSS sp1 _) = s_at sp1
| s_at (SIffVV vp1 _) = v_at vp1
| s_at (SExists _ sp) = s_at sp
| s_at (SForall _ part) = s_at (part_hd part)
| s_at (SPrev sp) = s_at sp + 1
| s_at (SNext sp) = s_at sp - 1
| s_at (SOnce i _) = i
| s_at (SEventually i _) = i
| s_at (SHistorically i _) = i
| s_at (SHistoricallyOut i) = i
| s_at (SAlways i _) = i
| s_at (SSince sp2 sp1s) = (case sp1s of []  $\Rightarrow$  s_at sp2 | _  $\Rightarrow$  s_at (last sp1s))
| s_at (SUntil sp1s sp2) = (case sp1s of []  $\Rightarrow$  s_at sp2 | sp1 # _  $\Rightarrow$  s_at sp1)
| v_at (VFF i) = i
| v_at (VPred i _) = i
| v_at (VEq_Const i _) = i
| v_at (VNeg sp) = s_at sp
| v_at (VOr vp1 _) = v_at vp1
| v_at (VAndL vp1) = v_at vp1
| v_at (VAndR vp2) = v_at vp2
| v_at (VImp sp1 _) = s_at sp1
| v_at (VIffSV sp1 _) = s_at sp1
| v_at (VIffVS vp1 _) = v_at vp1
| v_at (VExists _ part) = v_at (part_hd part)
| v_at (VForall _ vp1) = v_at vp1
| v_at (VPrev vp) = v_at vp + 1
| v_at (VPrevZ) = 0

```

$| v_at (VPrevOutL i) = i$
 $| v_at (VPrevOutR i) = i$
 $| v_at (VNext vp) = v_at vp - 1$
 $| v_at (VNextOutL i) = i$
 $| v_at (VNextOutR i) = i$
 $| v_at (VOnceOut i) = i$
 $| v_at (VOnce i _ _) = i$
 $| v_at (VEventually i _ _) = i$
 $| v_at (VHistorically i _ _) = i$
 $| v_at (VAlways i _ _) = i$
 $| v_at (VSinceOut i) = i$
 $| v_at (VSince i _ _) = i$
 $| v_at (VSinceInf i _ _) = i$
 $| v_at (VUntil i _ _) = i$
 $| v_at (VUntilInf i _ _) = i$

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

7 Auxiliary Lemmas

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

lemma $map_setE[elim_format]: map f xs = ys \Longrightarrow y \in set ys \Longrightarrow \exists x \in set xs. f x = y$
 $\langle proof \rangle$

lemma $set_Cons_eq: set_Cons X XS = (\bigcup xs \in XS. (\lambda x. x \# xs) ' X)$
 $\langle proof \rangle$

lemma $set_Cons_empty_iff: set_Cons X XS = \{\} \longleftrightarrow (X = \{\} \vee XS = \{\})$
 $\langle proof \rangle$

lemma $infinite_set_ConsI:$
 $XS \neq \{\} \Longrightarrow infinite X \Longrightarrow infinite (set_Cons X XS)$
 $X \neq \{\} \Longrightarrow infinite XS \Longrightarrow infinite (set_Cons X XS)$
 $\langle proof \rangle$

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 $fst_pos_None_iff: fst_pos xs x = None \longleftrightarrow x \notin set xs$
 $\langle proof \rangle$

lemma $nth_fst_pos: x \in set xs \Longrightarrow xs ! (the (fst_pos xs x)) = x$
 $\langle proof \rangle$

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 $eq_positions_iff: length xs = length ys$
 $\Longrightarrow positions xs x = positions ys y \longleftrightarrow (\forall n < length xs. xs ! n = x \longleftrightarrow ys ! n = y)$
 $\langle proof \rangle$

lemma $positions_eq_nil_iff: positions xs x = [] \longleftrightarrow x \notin set xs$

<proof>

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

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

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

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

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

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

lemma *sorted_positions*: $\text{sorted } (\text{positions } xs \ x)$
<proof>

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

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

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

lemma *subset_positions_map_snd*: $\text{set } (\text{positions } tXs \ tX) \subseteq \text{set } (\text{positions } (\text{map } \text{snd } tXs) (\text{snd } tX))$
<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>

8 Proof Checker

unbundle *MFOTL_notation*

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

begin

fun *s_check* :: $('n, 'd) \text{ env} \Rightarrow ('n, 'd) \text{ formula} \Rightarrow ('n, 'd) \text{ sproof} \Rightarrow \text{bool}$
and *v_check* :: $('n, 'd) \text{ env} \Rightarrow ('n, 'd) \text{ formula} \Rightarrow ('n, 'd) \text{ vproof} \Rightarrow \text{bool}$ **where**
s_check *v f p* = (case (f, p) of
 | $(\top, \text{STT } i) \Rightarrow \text{True}$
 | $(r \dagger ts, \text{SPred } i \text{ s } ts') \Rightarrow$
 $(r = s \wedge ts = ts' \wedge (r, v[[ts]]) \in \Gamma \sigma i)$
 | $(x \approx c, \text{SEq_Const } i \text{ x' c'}) \Rightarrow$
 $(c = c' \wedge x = x' \wedge v x = c)$
 | $(\neg_F \varphi, \text{SNeg } vp) \Rightarrow v_check \ v \ \varphi \ vp$
 | $(\varphi \vee_F \psi, \text{SOOrL } sp1) \Rightarrow s_check \ v \ \varphi \ sp1$
 | $(\varphi \vee_F \psi, \text{SOOrR } sp2) \Rightarrow s_check \ v \ \psi \ sp2$
 | $(\varphi \wedge_F \psi, \text{SAnd } sp1 \ sp2) \Rightarrow s_check \ v \ \varphi \ sp1 \wedge s_check \ v \ \psi \ sp2 \wedge s_at \ sp1 = s_at \ sp2$
 | $(\varphi \longrightarrow_F \psi, \text{SImpl } vp1) \Rightarrow v_check \ v \ \varphi \ vp1$
 | $(\varphi \longrightarrow_F \psi, \text{SImpR } sp2) \Rightarrow s_check \ v \ \psi \ sp2$
 | $(\varphi \longleftrightarrow_F \psi, \text{SIffSS } sp1 \ sp2) \Rightarrow s_check \ v \ \varphi \ sp1 \wedge s_check \ v \ \psi \ sp2 \wedge s_at \ sp1 = s_at \ sp2$
 | $(\varphi \longleftrightarrow_F \psi, \text{SIffVV } vp1 \ vp2) \Rightarrow v_check \ v \ \varphi \ vp1 \wedge v_check \ v \ \psi \ vp2 \wedge v_at \ vp1 = v_at \ vp2$
 | $(\exists_F x. \varphi, \text{SExists } y \ \text{val } sp) \Rightarrow (x = y \wedge s_check \ (v \ (x := \text{val})) \ \varphi \ sp)$
 | $(\forall_F x. \varphi, \text{SForall } y \ \text{sp_part}) \Rightarrow (\text{let } i = s_at \ (\text{part_hd } \text{sp_part})$
 $\text{in } x = y \wedge (\forall (sub, sp) \in \text{SubsVals } \text{sp_part}. s_at \ sp = i \wedge (\forall z \in \text{sub}. s_check \ (v \ (x := z)) \ \varphi \ sp)))$
 | $(\mathbf{Y} \ I \ \varphi, \text{SPrev } sp) \Rightarrow$
 $(\text{let } j = s_at \ sp; i = s_at \ (\text{SPrev } sp) \ \text{in}$
 $i = j+1 \wedge \text{mem} \ (\Delta \ \sigma \ i) \ I \wedge s_check \ v \ \varphi \ sp)$
 | $(\mathbf{X} \ I \ \varphi, \text{SNext } sp) \Rightarrow$
 $(\text{let } j = s_at \ sp; i = s_at \ (\text{SNext } sp) \ \text{in}$
 $j = i+1 \wedge \text{mem} \ (\Delta \ \sigma \ j) \ I \wedge s_check \ v \ \varphi \ sp)$
 | $(\mathbf{P} \ I \ \varphi, \text{SONce } i \ sp) \Rightarrow$
 $(\text{let } j = s_at \ sp \ \text{in}$
 $j \leq i \wedge \text{mem} \ (\tau \ \sigma \ i - \tau \ \sigma \ j) \ I \wedge s_check \ v \ \varphi \ sp)$
 | $(\mathbf{F} \ I \ \varphi, \text{SEventually } i \ sp) \Rightarrow$
 $(\text{let } j = s_at \ sp \ \text{in}$
 $j \geq i \wedge \text{mem} \ (\tau \ \sigma \ j - \tau \ \sigma \ i) \ I \wedge s_check \ v \ \varphi \ sp)$
 | $(\mathbf{H} \ I \ \varphi, \text{SHistoricallyOut } i) \Rightarrow$
 $\tau \ \sigma \ i < \tau \ \sigma \ 0 + \text{left } I$
 | $(\mathbf{H} \ I \ \varphi, \text{SHistorically } i \ li \ sps) \Rightarrow$
 $(li = (\text{case right } I \ \text{of } \infty \Rightarrow 0 \mid \text{enat } b \Rightarrow \text{ETP } \sigma \ (\tau \ \sigma \ i - b))$
 $\wedge \tau \ \sigma \ 0 + \text{left } I \leq \tau \ \sigma \ i$
 $\wedge \text{map } s_at \ sps = [li \ ..< (\text{LTP_p } \sigma \ i \ I) + 1]$
 $\wedge (\forall sp \in \text{set } sps. s_check \ v \ \varphi \ sp))$
 | $(\mathbf{G} \ I \ \varphi, \text{SAlways } i \ hi \ sps) \Rightarrow$
 $(hi = (\text{case right } I \ \text{of } \text{enat } b \Rightarrow \text{LTP_f } \sigma \ i \ b)$
 $\wedge \text{right } I \neq \infty$
 $\wedge \text{map } s_at \ sps = [(\text{ETP_f } \sigma \ i \ I) \ ..< hi + 1]$
 $\wedge (\forall sp \in \text{set } sps. s_check \ v \ \varphi \ sp))$

$| (\varphi \mathbf{S} I \psi, \mathbf{SSince} \text{ sp2 sp1s}) \Rightarrow$
 $(\text{let } i = s_at (\mathbf{SSince} \text{ sp2 sp1s}); j = s_at \text{ sp2 in}$
 $j \leq i \wedge \text{mem } (\tau \sigma i - \tau \sigma j) I$
 $\wedge \text{map } s_at \text{ sp1s} = [j+1 ..< i+1]$
 $\wedge s_check v \psi \text{ sp2}$
 $\wedge (\forall \text{ sp1} \in \text{set } \text{sp1s}. s_check v \varphi \text{ sp1}))$

$| (\varphi \mathbf{U} I \psi, \mathbf{SUntil} \text{ sp1s sp2}) \Rightarrow$
 $(\text{let } i = s_at (\mathbf{SUntil} \text{ sp1s sp2}); j = s_at \text{ sp2 in}$
 $j \geq i \wedge \text{mem } (\tau \sigma j - \tau \sigma i) I$
 $\wedge \text{map } s_at \text{ sp1s} = [i ..< j] \wedge s_check v \psi \text{ sp2}$
 $\wedge (\forall \text{ sp1} \in \text{set } \text{sp1s}. s_check v \varphi \text{ sp1}))$

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

$| v_check v f p = (\text{case } (f, p) \text{ of}$
 $(\perp, \mathbf{VFF} i) \Rightarrow \text{True}$
 $| (r \dagger ts, \mathbf{VPred} i \text{ pred } ts') \Rightarrow$
 $(r = \text{pred} \wedge ts = ts' \wedge (r, v[[ts]]) \notin \Gamma \sigma i)$
 $| (x \approx c, \mathbf{VEq_Const} i x' c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge v x \neq c)$
 $| (\neg_F \varphi, \mathbf{VNeg} \text{ sp}) \Rightarrow s_check v \varphi \text{ sp}$
 $| (\varphi \vee_F \psi, \mathbf{VOr} \text{ vp1 vp2}) \Rightarrow v_check v \varphi \text{ vp1} \wedge v_check v \psi \text{ vp2} \wedge v_at \text{ vp1} = v_at \text{ vp2}$
 $| (\varphi \wedge_F \psi, \mathbf{VAndL} \text{ vp1}) \Rightarrow v_check v \varphi \text{ vp1}$
 $| (\varphi \wedge_F \psi, \mathbf{VAndR} \text{ vp2}) \Rightarrow v_check v \psi \text{ vp2}$
 $| (\varphi \rightarrow_F \psi, \mathbf{VImp} \text{ sp1 vp2}) \Rightarrow s_check v \varphi \text{ sp1} \wedge v_check v \psi \text{ vp2} \wedge s_at \text{ sp1} = v_at \text{ vp2}$
 $| (\varphi \longleftrightarrow_F \psi, \mathbf{VIffSV} \text{ sp1 vp2}) \Rightarrow s_check v \varphi \text{ sp1} \wedge v_check v \psi \text{ vp2} \wedge s_at \text{ sp1} = v_at \text{ vp2}$
 $| (\varphi \longleftrightarrow_F \psi, \mathbf{VIffVS} \text{ vp1 sp2}) \Rightarrow v_check v \varphi \text{ vp1} \wedge s_check v \psi \text{ sp2} \wedge v_at \text{ vp1} = s_at \text{ sp2}$
 $| (\exists Fx. \varphi, \mathbf{VExists} y \text{ vp_part}) \Rightarrow (\text{let } i = v_at (\text{part_hd } \text{vp_part})$
 $\text{in } x = y \wedge (\forall (\text{sub}, \text{vp}) \in \text{SubsVals } \text{vp_part}. v_at \text{ vp} = i \wedge (\forall z \in \text{sub}. v_check (v (x := z)) \varphi \text{ vp})))$

$| (\forall Fx. \varphi, \mathbf{VForall} y \text{ val } \text{vp}) \Rightarrow (x = y \wedge v_check (v (x := \text{val})) \varphi \text{ vp})$

$| (\mathbf{Y} I \varphi, \mathbf{VPrev} \text{ vp}) \Rightarrow$
 $(\text{let } j = v_at \text{ vp}; i = v_at (\mathbf{VPrev} \text{ vp}) \text{ in}$
 $i = j+1 \wedge v_check v \varphi \text{ vp})$

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

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

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

$| (\mathbf{X} I \varphi, \mathbf{VNext} \text{ vp}) \Rightarrow$
 $(\text{let } j = v_at \text{ vp}; i = v_at (\mathbf{VNext} \text{ vp}) \text{ in}$
 $j = i+1 \wedge v_check v \varphi \text{ vp})$

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

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

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

$| (\mathbf{P} I \varphi, \mathbf{VOnce} i \text{ li } \text{vps}) \Rightarrow$
 $(\text{li} = (\text{case } \text{right } I \text{ of } \infty \Rightarrow 0 \mid \text{enat } b \Rightarrow \text{ETP_p } \sigma i b)$
 $\wedge \tau \sigma 0 + \text{left } I \leq \tau \sigma i$
 $\wedge \text{map } v_at \text{ vps} = [\text{li} ..< (\text{LTP_p } \sigma i I) + 1]$
 $\wedge (\forall \text{ vp} \in \text{set } \text{vps}. v_check v \varphi \text{ vp}))$

$| (\mathbf{F} I \varphi, \mathbf{VEventually} i \text{ hi } \text{vps}) \Rightarrow$
 $(\text{hi} = (\text{case } \text{right } I \text{ of } \text{enat } b \Rightarrow \text{LTP_f } \sigma i b) \wedge \text{right } I \neq \infty$
 $\wedge \text{map } v_at \text{ vps} = [(\text{ETP_f } \sigma i I) ..< \text{hi} + 1]$
 $\wedge (\forall \text{ vp} \in \text{set } \text{vps}. v_check v \varphi \text{ vp}))$

$| (\mathbf{H} I \varphi, \mathbf{VHistorically} i \text{ vp}) \Rightarrow$
 $(\text{let } j = v_at \text{ vp in}$
 $j \leq i \wedge \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge v_check v \varphi \text{ vp})$

$| (\mathbf{G} I \varphi, \mathbf{VAlways} i \text{ vp}) \Rightarrow$

```

    (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))
  | ( _ , _ ) ⇒ False

```

```

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

```

8.1 Checker Soundness

lemma *check_soundness*:

```

s_check v φ sp ⇒ SAT σ v (s_at sp) φ
v_check v φ vp ⇒ VIO σ v (v_at vp) φ

```

<proof>

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 ∪ Y) vs v  $\longleftrightarrow compatible X vs v \wedge compatible Y vs v$ 

```

<proof>

lemma *compatible_vals_union_eq*:

```

compatible_vals (X ∪ Y) vs = compatible_vals X vs ∩ compatible_vals Y 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 × 'a set) list ⇒ '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:

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

lemma mk_values_not_Nil:

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

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

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

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

$vs \in mk_values\ tXs \implies$

$vs\ !\ n = vs\ !\ m$

⟨proof⟩

definition mk_values_subset p tXs X

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

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

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

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

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

inf_dups

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

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

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

⟨proof⟩

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

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

⟨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_Cmpl* $r \ vs \ ts \ i = (\{r\} \times \text{mk_values } (vs \llbracket ts \rrbracket)) \subseteq - \Gamma \ \sigma \ i$

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

lemma *mk_values_alt*:
 $\text{mk_values } (vs \llbracket ts \rrbracket) =$
 $\{cs. \exists v \in \text{compatible_vals } (\bigcup (fv_trm \ ' \ \text{set } ts)) \ vs. cs = v \llbracket ts \rrbracket\}$
 ⟨proof⟩

lemma *check_values_neq_NoneI*:
assumes $v \in \text{compatible_vals } (\bigcup (fv_trm \ ' \ \text{set } ts) - \text{dom } f) \ vs \wedge x \ y. f \ x = \text{Some } y \implies y \in vs \ x$
shows $\text{check_values } vs \ ts \ ((\lambda x. \text{case } f \ x \ \text{of } \text{None} \Rightarrow v \ x \mid \text{Some } y \Rightarrow y) \llbracket ts \rrbracket) \ (\text{Some } f) \neq \text{None}$
 ⟨proof⟩

lemma *check_values_eq_NoneI*:
 $\forall v \in \text{compatible_vals } (\bigcup (fv_trm \ ' \ \text{set } ts) - \text{dom } f) \ vs. us \neq (\lambda x. \text{case } f \ x \ \text{of } \text{None} \Rightarrow v \ x \mid \text{Some } y \Rightarrow$
 $y) \llbracket ts \rrbracket \implies$
 $\text{check_values } vs \ ts \ us \ (\text{Some } f) = \text{None}$
 ⟨proof⟩

lemma *mk_values_subset_Cmpl_code[code]*:
 $\text{mk_values_subset_Cmpl } r \ vs \ ts \ i = (\forall (q, us) \in \Gamma \ \sigma \ i. q \neq r \vee \text{check_values } vs \ ts \ us \ (\text{Some } \text{Map.empty})$
 $= \text{None})$
 ⟨proof⟩

8.2 Executable Variant of the Checker

fun *s_check_exec* :: $(\ 'n, \ 'd) \ \text{envset} \Rightarrow (\ 'n, \ 'd) \ \text{formula} \Rightarrow (\ 'n, \ 'd) \ \text{sproof} \Rightarrow \text{bool}$

and $v_check_exec :: ('n, 'd) envset \Rightarrow ('n, 'd) formula \Rightarrow ('n, 'd) vproof \Rightarrow bool$ **where**

s_check_exec vs f $p = (case (f, p) of$

- $(\top, STT i) \Rightarrow True$
- $| (r \dagger ts, SPred i s ts') \Rightarrow$
 $(r = s \wedge ts = ts' \wedge mk_values_subset r (vs\{ts\})) (\Gamma \sigma i)$
- $| (x \approx c, SEq_Const i x' c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge vs x = \{c\})$
- $| (\neg_F \varphi, SNeg vp) \Rightarrow v_check_exec vs \varphi vp$
- $| (\varphi \vee_F \psi, SOrL sp1) \Rightarrow s_check_exec vs \varphi sp1$
- $| (\varphi \vee_F \psi, SOrR sp2) \Rightarrow s_check_exec vs \psi sp2$
- $| (\varphi \wedge_F \psi, SAnd sp1 sp2) \Rightarrow s_check_exec vs \varphi sp1 \wedge s_check_exec vs \psi sp2 \wedge s_at sp1 = s_at sp2$
- $| (\varphi \longrightarrow_F \psi, SImpl vp1) \Rightarrow v_check_exec vs \varphi vp1$
- $| (\varphi \longrightarrow_F \psi, SImplR sp2) \Rightarrow s_check_exec vs \psi sp2$
- $| (\varphi \longleftrightarrow_F \psi, SIffSS sp1 sp2) \Rightarrow s_check_exec vs \varphi sp1 \wedge s_check_exec vs \psi sp2 \wedge s_at sp1 = s_at$
 $sp2$
- $| (\varphi \longleftrightarrow_F \psi, SIffVV vp1 vp2) \Rightarrow v_check_exec vs \varphi vp1 \wedge v_check_exec vs \psi vp2 \wedge v_at vp1 = v_at$
 $vp2$
- $| (\exists_F x. \varphi, SExists y val sp) \Rightarrow (x = y \wedge s_check_exec (vs (x := \{val\})) \varphi sp)$
- $| (\forall_F x. \varphi, SForall y sp_part) \Rightarrow (let i = s_at (part_hd sp_part)$
 $in x = y \wedge (\forall (sub, sp) \in SubsVals sp_part. s_at sp = i \wedge s_check_exec (vs (x := sub)) \varphi sp))$
- $| (\mathbf{Y} I \varphi, SPrev sp) \Rightarrow$
 $(let j = s_at sp; i = s_at (SPrev sp) in$
 $i = j+1 \wedge mem (\Delta \sigma i) I \wedge s_check_exec vs \varphi sp)$
- $| (\mathbf{X} I \varphi, SNext sp) \Rightarrow$
 $(let j = s_at sp; i = s_at (SNext sp) in$
 $j = i+1 \wedge mem (\Delta \sigma j) I \wedge s_check_exec vs \varphi sp)$
- $| (\mathbf{P} I \varphi, SOnce i sp) \Rightarrow$
 $(let j = s_at sp in$
 $j \leq i \wedge mem (\tau \sigma i - \tau \sigma j) I \wedge s_check_exec vs \varphi sp)$
- $| (\mathbf{F} I \varphi, SEventually i sp) \Rightarrow$
 $(let j = s_at sp in$
 $j \geq i \wedge mem (\tau \sigma j - \tau \sigma i) I \wedge s_check_exec vs \varphi sp)$
- $| (\mathbf{H} I \varphi, SHistoricallyOut i) \Rightarrow$
 $\tau \sigma i < \tau \sigma 0 + left I$
- $| (\mathbf{H} I \varphi, SHistorically i li sps) \Rightarrow$
 $(li = (case right I of \infty \Rightarrow 0 \mid enat b \Rightarrow ETP \sigma (\tau \sigma i - b))$
 $\wedge \tau \sigma 0 + left I \leq \tau \sigma i$
 $\wedge map s_at sps = [li ..< (LTP_p \sigma i I) + 1]$
 $\wedge (\forall sp \in set sps. s_check_exec vs \varphi sp))$
- $| (\mathbf{G} I \varphi, SAlways i hi sps) \Rightarrow$
 $(hi = (case right I of enat b \Rightarrow LTP_f \sigma i b)$
 $\wedge right I \neq \infty$
 $\wedge map s_at sps = [(ETP_f \sigma i I) ..< hi + 1]$
 $\wedge (\forall sp \in set sps. s_check_exec vs \varphi sp))$
- $| (\varphi \mathbf{S} I \psi, SSince sp2 sp1s) \Rightarrow$
 $(let i = s_at (SSince sp2 sp1s); j = s_at sp2 in$
 $j \leq i \wedge mem (\tau \sigma i - \tau \sigma j) I$
 $\wedge map s_at sp1s = [j+1 ..< i+1]$
 $\wedge s_check_exec vs \psi sp2$
 $\wedge (\forall sp1 \in set sp1s. s_check_exec vs \varphi sp1))$
- $| (\varphi \mathbf{U} I \psi, SUntil sp1s sp2) \Rightarrow$
 $(let i = s_at (SUntil sp1s sp2); j = s_at sp2 in$
 $j \geq i \wedge mem (\tau \sigma j - \tau \sigma i) I$
 $\wedge map s_at sp1s = [i ..< j] \wedge s_check_exec vs \psi sp2$
 $\wedge (\forall sp1 \in set sp1s. s_check_exec vs \varphi sp1))$
- $| (_ , _) \Rightarrow False$

v_check_exec vs f $p = (case (f, p) of$

- $(\perp, VFF i) \Rightarrow True$

$| (r \dagger ts, VPred\ i\ pred\ ts') \Rightarrow$
 $(r = pred \wedge ts = ts' \wedge mk_values_subset_Compl\ r\ vs\ ts\ i)$
 $| (x \approx c, VEq_Const\ i\ x'\ c') \Rightarrow$
 $(c = c' \wedge x = x' \wedge c \notin vs\ x)$
 $| (\neg_F \varphi, VNeg\ sp) \Rightarrow s_check_exec\ vs\ \varphi\ sp$
 $| (\varphi \vee_F \psi, VOr\ vp1\ vp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge v_at\ vp1 = v_at\ vp2$
 $| (\varphi \wedge_F \psi, VAndL\ vp1) \Rightarrow v_check_exec\ vs\ \varphi\ vp1$
 $| (\varphi \wedge_F \psi, VAndR\ vp2) \Rightarrow v_check_exec\ vs\ \psi\ vp2$
 $| (\varphi \rightarrow_F \psi, VImp\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $| (\varphi \longleftrightarrow_F \psi, VIffSV\ sp1\ vp2) \Rightarrow s_check_exec\ vs\ \varphi\ sp1 \wedge v_check_exec\ vs\ \psi\ vp2 \wedge s_at\ sp1 = v_at\ vp2$
 $| (\varphi \longleftrightarrow_F \psi, VIffVS\ vp1\ sp2) \Rightarrow v_check_exec\ vs\ \varphi\ vp1 \wedge s_check_exec\ vs\ \psi\ sp2 \wedge v_at\ vp1 = s_at\ sp2$
 $| (\exists_F x. \varphi, VExists\ y\ vp_part) \Rightarrow (let\ i = v_at\ (part_hd\ vp_part)$
 $\quad in\ x = y \wedge (\forall (sub, vp) \in SubsVals\ vp_part. v_at\ vp = i \wedge v_check_exec\ (vs\ (x := sub))\ \varphi\ vp))$
 $| (\forall_F x. \varphi, VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v_check_exec\ (vs\ (x := \{val\}))\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrev\ vp) \Rightarrow$
 $(let\ j = v_at\ vp; i = v_at\ (VPrev\ vp)\ in$
 $\quad i = j+1 \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{Y}\ I\ \varphi, VPrevZ) \Rightarrow True$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutL\ i) \Rightarrow$
 $\quad i > 0 \wedge \Delta\ \sigma\ i < left\ I$
 $| (\mathbf{Y}\ I\ \varphi, VPrevOutR\ i) \Rightarrow$
 $\quad i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I$
 $| (\mathbf{X}\ I\ \varphi, VNext\ vp) \Rightarrow$
 $(let\ j = v_at\ vp; i = v_at\ (VNext\ vp)\ in$
 $\quad j = i+1 \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{X}\ I\ \varphi, VNextOutL\ i) \Rightarrow$
 $\quad \Delta\ \sigma\ (i+1) < left\ I$
 $| (\mathbf{X}\ I\ \varphi, VNextOutR\ i) \Rightarrow$
 $\quad enat\ (\Delta\ \sigma\ (i+1)) > right\ I$
 $| (\mathbf{P}\ I\ \varphi, VOnceOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\mathbf{P}\ I\ \varphi, VOnce\ i\ li\ vps) \Rightarrow$
 $(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ v_at\ vps = [li\ ..<\ (LTP_p\ \sigma\ i\ I) + 1]$
 $\quad \wedge (\forall vp \in set\ vps. v_check_exec\ vs\ \varphi\ vp))$
 $| (\mathbf{F}\ I\ \varphi, VEventually\ i\ hi\ vps) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
 $\quad \wedge map\ v_at\ vps = [(ETP_f\ \sigma\ i\ I) ..<\ hi + 1]$
 $\quad \wedge (\forall vp \in set\ vps. v_check_exec\ vs\ \varphi\ vp))$
 $| (\mathbf{H}\ I\ \varphi, VHistorically\ i\ vp) \Rightarrow$
 $(let\ j = v_at\ vp\ in$
 $\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\mathbf{G}\ I\ \varphi, VAlways\ i\ vp) \Rightarrow$
 $(let\ j = v_at\ vp$
 $\quad in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v_check_exec\ vs\ \varphi\ vp)$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSinceOut\ i) \Rightarrow$
 $\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSince\ i\ vp1\ vp2s) \Rightarrow$
 $(let\ j = v_at\ vp1\ in$
 $\quad (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b \leq j) \wedge j \leq i$
 $\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\quad \wedge map\ v_at\ vp2s = [j\ ..<\ (LTP_p\ \sigma\ i\ I) + 1] \wedge v_check_exec\ vs\ \varphi\ vp1$
 $\quad \wedge (\forall vp2 \in set\ vp2s. v_check_exec\ vs\ \psi\ vp2))$
 $| (\varphi\ \mathbf{S}\ I\ \psi, VSinceInf\ i\ li\ vp2s) \Rightarrow$

$(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP_p\ \sigma\ i\ b)$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ map\ v_at\ vp2s = [li\ ..<\ (LTP_p\ \sigma\ i\ I) + 1]$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntil\ i\ vp2s\ vp1) \Rightarrow$
 $(let\ j = v_at\ vp1\ in$
 $(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP_f\ \sigma\ i\ b) \wedge\ i \leq j$
 $\wedge\ map\ v_at\ vp2s = [ETP_f\ \sigma\ i\ I\ ..<\ j + 1] \wedge\ v_check_exec\ vs\ \varphi\ vp1$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntilInf\ i\ hi\ vp2s) \Rightarrow$
 $(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP_f\ \sigma\ i\ b) \wedge\ right\ I \neq \infty$
 $\wedge\ map\ v_at\ vp2s = [ETP_f\ \sigma\ i\ I\ ..<\ hi + 1]$
 $\wedge\ (\forall\ vp2 \in set\ vp2s.\ v_check_exec\ vs\ \psi\ vp2))$
 $\mid (_ , _) \Rightarrow False)$

declare $s_check_exec.simps[simp\ del]\ v_check_exec.simps[simp\ del]$
simps_of_case $s_check_exec_simps[simp,\ code]: s_check_exec.simps[unfolding\ prod.case]$ (splits: formula.split sproof.split)
simps_of_case $v_check_exec_simps[simp,\ code]: v_check_exec.simps[unfolding\ prod.case]$ (splits: formula.split vproof.split)

lemma *check_fv_cong*:

assumes $\forall x \in fv\ \varphi.\ v\ x = v'\ x$
shows $s_check\ v\ \varphi\ sp \longleftrightarrow s_check\ v'\ \varphi\ sp\ v_check\ v\ \varphi\ vp \longleftrightarrow v_check\ v'\ \varphi\ vp$
 $\langle proof \rangle$

lemma *s_check_fun_upd_notin[simp]*:

$x \notin fv\ \varphi \implies s_check\ (v(x := t))\ \varphi\ sp = s_check\ v\ \varphi\ sp$
 $\langle proof \rangle$

lemma *v_check_fun_upd_notin[simp]*:

$x \notin fv\ \varphi \implies v_check\ (v(x := t))\ \varphi\ sp = v_check\ v\ \varphi\ sp$
 $\langle proof \rangle$

lemma *SubsVals_nonempty*: $(X, t) \in SubsVals\ part \implies X \neq \{\}$

$\langle proof \rangle$

lemma *compatible_vals_nonemptyI*: $\forall x.\ vs\ x \neq \{\} \implies compatible_vals\ A\ vs \neq \{\}$

$\langle proof \rangle$

lemma *compatible_vals_fun_upd*: $compatible_vals\ A\ (vs(x := X)) =$

$(if\ x \in A\ then\ \{v \in compatible_vals\ (A - \{x\})\ vs.\ v\ x \in X\}\ else\ compatible_vals\ A\ vs)$
 $\langle proof \rangle$

lemma *fun_upd_in_compatible_vals*: $v \in compatible_vals\ (A - \{x\})\ vs \implies v(x := t) \in compatible_vals\ (A - \{x\})\ vs$

$\langle proof \rangle$

lemma *fun_upd_in_compatible_vals_in*: $v \in compatible_vals\ (A - \{x\})\ vs \implies t \in vs\ x \implies v(x := t) \in compatible_vals\ A\ vs$

$\langle proof \rangle$

lemma *fun_upd_in_compatible_vals_notin*: $x \notin A \implies v \in compatible_vals\ A\ vs \implies v(x := t) \in compatible_vals\ A\ vs$

$\langle proof \rangle$

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 $v_check_exec\ vs\ \varphi\ vp \longleftrightarrow (\forall v \in compatible_vals\ (fv\ \varphi)\ vs.\ v_check\ v\ \varphi\ vp)$
 $\langle proof \rangle$

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

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

8.3 Latest Relevant Time-Point

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

$LRTP\ \top\ i = Some\ i$
 $| LRTP\ \perp\ i = Some\ i$
 $| LRTP\ (_ \dagger _) i = Some\ i$
 $| LRTP\ (_ \approx _) i = Some\ i$
 $| LRTP\ (\neg_F\ \varphi) i = LRTP\ \varphi\ i$
 $| LRTP\ (\varphi \vee_F\ \psi) i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \wedge_F\ \psi) i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \rightarrow_F\ \psi) i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\varphi \longleftrightarrow_F\ \psi) i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
 $| LRTP\ (\exists_{F_}\ \varphi) i = LRTP\ \varphi\ i$
 $| LRTP\ (\forall_{F_}\ \varphi) i = LRTP\ \varphi\ i$
 $| LRTP\ (\mathbf{Y}\ I\ \varphi) i = LRTP\ \varphi\ (i-1)$
 $| LRTP\ (\mathbf{X}\ I\ \varphi) i = LRTP\ \varphi\ (i+1)$
 $| LRTP\ (\mathbf{P}\ I\ \varphi) i = LRTP\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $| LRTP\ (\mathbf{H}\ I\ \varphi) i = LRTP\ \varphi\ (LTP_p_safe\ \sigma\ i\ I)$
 $| LRTP\ (\mathbf{F}\ I\ \varphi) i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP_f\ \sigma\ i\ b))$
 $| LRTP\ (\mathbf{G}\ I\ \varphi) i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP_f\ \sigma\ i\ b))$
 $| LRTP\ (\varphi\ \mathbf{S}\ I\ \psi) i = max_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ (LTP_p_safe\ \sigma\ i\ I))$
 $| LRTP\ (\varphi\ \mathbf{U}\ I\ \psi) i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow max_opt\ (LRTP\ \varphi\ ((LTP_f\ \sigma\ i\ b)-1))\ (LRTP\ \psi\ (LTP_f\ \sigma\ i\ b)))$

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 $\exists j.\ LRTP\ \varphi\ i = Some\ j$
 $\langle proof \rangle$

lemma $enat_trans[simp]: enat\ i \leq enat\ j \wedge enat\ j \leq enat\ k \Longrightarrow enat\ i \leq enat\ k$
 $\langle proof \rangle$

8.4 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 $val_in_AD_iff:$
 $x \in fv\ \varphi \Longrightarrow v\ x \in AD\ \varphi\ i \longleftrightarrow v\ x \in consts\ \varphi \vee$

$(\exists r \text{ ts } k. k \leq \text{the } (LRTP \ \varphi \ i) \wedge (r, v[[\text{ts}]]) \in \Gamma \ \sigma \ k \wedge x \in \bigcup (\text{set } (\text{map } fv_trm \ \text{ts})))$
 <proof>

lemma *val_notin_AD_iff*:

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

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

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

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

lemma *finite_AD_UNIV*:

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

<proof>

8.5 Congruence Modulo Active Domain

lemma *AD_simps[simp]*:

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

assumes $i \leq j$
shows $LTP_f \ \sigma \ i \ b \leq LTP_f \ \sigma \ j \ b$
 <proof>

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

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

lemma *LTP_p_safe_le[simp]*: $LTP_p_safe\ \sigma\ i\ I \leq i$
 ⟨proof⟩

lemma *check_AD_cong*:
 assumes *future_bounded* φ
 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⟩

8.6 Checker Completeness

lemma *part_hd_tabulate*: $distinct\ xs \implies part_hd\ (tabulate\ xs\ f\ z) = (case\ xs\ of\ [] \Rightarrow z \mid (x\ \# _) \Rightarrow (if\ set\ xs = UNIV\ then\ f\ x\ else\ z))$
 ⟨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* φ
 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* $\sigma v i \varphi \longrightarrow \text{future_bounded } \varphi \longrightarrow (\exists sp. s_at\ sp = i \wedge s_check\ v\ \varphi\ sp)) \wedge$
(*VIO* $\sigma v i \varphi \longrightarrow \text{future_bounded } \varphi \longrightarrow (\exists vp. v_at\ vp = i \wedge v_check\ v\ \varphi\ vp))$)

<proof>

lemmas *check_completeness* =

conjunct1[*OF* *check_completeness_aux*, *rule_format*]
conjunct2[*OF* *check_completeness_aux*, *rule_format*]

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

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

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
 Inl $p \Rightarrow s_check_exec\ vs\ \varphi\ p \wedge s_at\ p = i$
 | *Inr* $p \Rightarrow v_check_exec\ vs\ \varphi\ p \wedge v_at\ p = i$)

end

8.7 Lifting the Checker to PDTs

fun *check_one* **where**

check_one $\sigma v \varphi (Leaf\ p) = p_check\ \sigma v \varphi\ p$
| *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 $DS\ \sigma vs \varphi (Leaf\ p) = (\text{if } p_check_exec\ \sigma vs \varphi\ p \text{ then } \{\} \text{ else rev } 'DS)$
| *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*: $\forall x \in fv\ \varphi \cup vars\ e. v\ x = v'\ x \Longrightarrow check_one\ \sigma v \varphi\ e = check_one\ \sigma v' \varphi\ e$
<proof>

lemma *check_all_aux_check_one*: $\forall x. vs\ x \neq \{\} \Longrightarrow distinct_paths\ e \Longrightarrow (\forall x \in vars\ e. vs\ x = UNIV)$
 \Longrightarrow

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

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

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

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

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

fun *pdt_at* **where**

pdt_at $i (Leaf\ l) = (p_at\ l = i)$
| *pdt_at* $i (Node\ x\ part) = (\forall pdt \in Vals\ part. pdt_at\ i\ pdt)$

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

lemma *check_all_completeness_aux*:

fixes $\varphi :: ('n, 'd :: \{default, linorder\})\ formula$
shows $set\ vs \subseteq fv\ \varphi \implies future_bounded\ \varphi \implies distinct\ vs \implies$
 $\exists e. pdt_at\ i\ e \wedge vars_order\ vs\ e \wedge (\forall v. (\forall x. x \notin set\ vs \longrightarrow v\ x = w\ x) \longrightarrow p_check\ \sigma\ v\ \varphi\ (eval_pdt\ v\ e))$
 <proof>

lemma *check_all_completeness*:

fixes $\varphi :: ('n, 'd :: \{default, linorder\})\ formula$
assumes *future_bounded* φ
shows $\exists e. pdt_at\ i\ e \wedge check_all\ \sigma\ \varphi\ e$
 <proof>

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

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

unbundle *MFOTL_no_notation* — disable notation

9 Type of Events

9.1 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* \Rightarrow *String.literal is String.Abs_literal* <proof>

declare [[*coercion string8_literal*]]

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

begin

lift_definition *equal_string8* :: *string8* \Rightarrow *string8* \Rightarrow *bool is HOL.equal* <proof>

lift_definition *less_eq_string8* :: *string8* \Rightarrow *string8* \Rightarrow *bool is ord_class.lexordp_eq* <proof>

lift_definition *less_string8* :: *string8* \Rightarrow *string8* \Rightarrow *bool is ord_class.lexordp* <proof>

instance <proof>

end

lifting_forget *string8.lifting*

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

code_printing

```

type_constructor string8 → (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

```

9.2 Event Parameters

definition *div_to_zero* :: integer ⇒ integer ⇒ integer **where**

```

div_to_zero x y = (let z = fst (Code_Numeral.divmod_abs x y) in
  if (x < 0) ≠ (y < 0) then - z else z)

```

definition *mod_to_zero* :: integer ⇒ integer ⇒ integer **where**

```

mod_to_zero x y = (let z = snd (Code_Numeral.divmod_abs x y) in
  if x < 0 then - z else z)

```

lemma $b \neq 0 \implies \text{div_to_zero } a \ b * b + \text{mod_to_zero } a \ b = a$

⟨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 ≤ EInt y ↔ x ≤ y
| EString x ≤ EString y ↔ x ≤ y
| EInt _ ≤ EString _ ↔ True
| (_ :: event_data) ≤ _ ↔ False

```

definition *less_event_data* :: *event_data* ⇒ *event_data* ⇒ bool **where**

```

less_event_data x y ↔ x ≤ y ∧ ¬ y ≤ x

```

fun *plus_event_data* **where**

```

EInt x + EInt y = EInt (x + y)

```

```

| (_::event_data) + _ = undefined

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

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

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

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

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

instance <proof>

end

lemma infinite_UNIV_event_data:
  ¬finite (UNIV :: event_data set)
<proof>

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

instantiation event_data :: default begin

definition default_event_data :: event_data where default = EInt 0

instance <proof>

end

instantiation event_data :: linorder begin
instance
<proof>

end

```

10 Code Generation

10.1 Type Class Instances

```

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

```

```

begin

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

end

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

instantiation bool :: universe begin
definition universe_bool :: bool list option where universe_bool = Some [True, False]
instance <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)  $\Rightarrow$  Some (List.product xs ys) | _  $\Rightarrow$  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)  $\Rightarrow$  Some (map Inl xs @ map Inr ys) | _  $\Rightarrow$  None)
instance <proof>
end
instantiation option :: (universe) universe begin
definition universe_option = (case universe of Some xs  $\Rightarrow$  Some (None # map Some xs) | _  $\Rightarrow$  None)
instance <proof>
end
instantiation fun :: (universe, universe) universe begin
definition universe_fun :: ('a  $\Rightarrow$  'b) list option where universe_fun =
  (case (universe, universe) of
    (Some xs, Some ys)  $\Rightarrow$  Some (map ( $\lambda$ zs. the  $\circ$  map_of (zip xs zs)) (List.n_lists (length xs) ys))
  | (_, Some [x])  $\Rightarrow$  Some [ $\lambda$ _. x]
  | _  $\Rightarrow$  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

```

10.2 Progress

```

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

```

```

lemma Inf_Min:
  fixes P :: nat ⇒ bool
  assumes P j

```

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

lemma *progress_Eventually_code*: $\text{progress } \sigma \text{ (Formula.Eventually } I \varphi) j =$
 (let $m = \min j \text{ (Suc (progress } \sigma \varphi j)) - 1$ in $\text{Min } (\text{Set.filter } (\lambda i. \text{enat } (\delta \sigma m i) \leq \text{right } I) \{..j\})$)
 ⟨proof⟩

lemma *progress_Always_code*: $\text{progress } \sigma \text{ (Formula.Always } I \varphi) j =$
 (let $m = \min j \text{ (Suc (progress } \sigma \varphi j)) - 1$ in $\text{Min } (\text{Set.filter } (\lambda i. \text{enat } (\delta \sigma m i) \leq \text{right } I) \{..j\})$)
 ⟨proof⟩

lemma *progress_Until_code*: $\text{progress } \sigma \text{ (Formula.Until } \varphi I \psi) j =$
 (let $m = \min j \text{ (Suc (min (progress } \sigma \varphi j) \text{ (progress } \sigma \psi j))) - 1$ 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*

10.3 Trace

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

lemma *extend_is_stream*:
assumes *sorted* (map snd list)
and $\bigwedge x. x \in \text{set list} \implies \text{snd } x \leq m$
and $\bigwedge x. x \in \text{set list} \implies \text{finite } (\text{fst } x)$
shows *ssorted* (smap snd (list @- smap ($\lambda n. (\{\}, n + m)$) nats)) \wedge
sincreasing (smap snd (list @- smap ($\lambda n. (\{\}, n + m)$) nats)) \wedge
sfinite (smap fst (list @- smap ($\lambda n. (\{\}, n + m)$) nats))
 ⟨proof⟩

typedef 'a *trace_mapping* = $\{(n, m, t) :: (\text{nat} \times \text{nat} \times (\text{nat}, 'a \text{ set} \times \text{nat}) \text{ mapping}) \mid$
 $n \ m \ t. \text{Mapping.keys } t = \{..<n\} \wedge$
 $\text{sorted } (\text{map } (\text{snd} \circ (\text{the} \circ \text{Mapping.lookup } t)) [0..<n]) \wedge$
 $(\text{case } n \text{ of } 0 \Rightarrow \text{True} \mid \text{Suc } n' \Rightarrow (\text{case } \text{Mapping.lookup } t \ n' \text{ of } \text{Some } (X', t') \Rightarrow t' \leq m \mid \text{None} \Rightarrow \text{False}))$
 \wedge
 $(\forall n' < n. \text{case } \text{Mapping.lookup } t \ n' \text{ of } \text{Some } (X', t') \Rightarrow \text{finite } X' \mid \text{None} \Rightarrow \text{False})\}$
 ⟨proof⟩

setup_lifting *type_definition_trace_mapping*

lemma *lookup_bulkload_Some*: $i < \text{length list} \implies$
 $\text{Mapping.lookup } (\text{Mapping.bulkload list}) \ i = \text{Some } (\text{list } ! \ i)$
 ⟨proof⟩

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

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

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

<proof>

code_datatype *Trace_Mapping*

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

lemma $\Gamma_rbt_code[code]: \Gamma (\text{Trace_Mapping } t) i = \text{fst } (\text{trace_mapping_nth } t i)$
<proof>

lemma $\tau_rbt_code[code]: \tau (\text{Trace_Mapping } t) i = \text{snd } (\text{trace_mapping_nth } t i)$
<proof>

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

10.4 Auxiliary results

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

lemma *sum_proofs_empty[simp]*: $\text{sum_proofs } f [] = 0$
<proof>

lemma *sum_proofs_fundef_cong[fundef_cong]*: $(\bigwedge x. x \in \text{set } xs \implies f x = f' x) \implies$
 $\text{sum_proofs } f xs = \text{sum_proofs } f' xs$
<proof>

lemma *sum_proofs_Cons*:
fixes $f :: 'a \Rightarrow \text{nat}$
shows $\text{sum_proofs } f (p \# qs) = f p + \text{sum_proofs } f qs$
<proof>

lemma *sum_proofs_app*:
fixes $f :: 'a \Rightarrow \text{nat}$
shows $\text{sum_proofs } f (qs @ [p]) = f p + \text{sum_proofs } f qs$
<proof>

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

function (*sequential*) $s_pred :: ('n, 'd) \text{proof} \Rightarrow \text{nat}$
and $v_pred :: ('n, 'd) \text{vproof} \Rightarrow \text{nat}$ **where**
 $s_pred (\text{STT } _) = 1$
 $| s_pred (\text{SEq_Const } _ _ _) = 1$
 $| s_pred (\text{SPred } _ r _) = w r$
 $| s_pred (\text{SNeg } vp) = (v_pred vp) + 1$
 $| s_pred (\text{SOl } sp1) = (s_pred sp1) + 1$
 $| s_pred (\text{SOrr } sp2) = (s_pred sp2) + 1$
 $| s_pred (\text{SAnd } sp1 sp2) = (s_pred sp1) + (s_pred sp2) + 1$
 $| s_pred (\text{SImpL } vp1) = (v_pred vp1) + 1$
 $| s_pred (\text{SImpR } sp2) = (s_pred sp2) + 1$
 $| s_pred (\text{SIffSS } sp1 sp2) = (s_pred sp1) + (s_pred sp2) + 1$
 $| s_pred (\text{SIffVV } vp1 vp2) = (v_pred vp1) + (v_pred vp2) + 1$
 $| s_pred (\text{SExists } _ _ sp) = (s_pred sp) + 1$
 $| s_pred (\text{SForall } _ \text{part}) = (\text{sum_proofs } s_pred (\text{vals part})) + 1$
 $| s_pred (\text{SPrev } sp) = (s_pred sp) + 1$

$| s_pred (SNext\ sp) = (s_pred\ sp) + 1$
 $| s_pred (SONce\ _ sp) = (s_pred\ sp) + 1$
 $| s_pred (SEventually\ _ sp) = (s_pred\ sp) + 1$
 $| s_pred (SHistorically\ _ _ sps) = (sum_proofs\ s_pred\ sps) + 1$
 $| s_pred (SHistoricallyOut\ _) = 1$
 $| s_pred (SAlways\ _ _ sps) = (sum_proofs\ s_pred\ sps) + 1$
 $| s_pred (SSince\ sp2\ sp1s) = (sum_proofs\ s_pred\ (sp2\ \#\ sp1s)) + 1$
 $| s_pred (SUntil\ sp1s\ sp2) = (sum_proofs\ s_pred\ (sp1s\ @\ [sp2])) + 1$
 $| v_pred (VFF\ _) = 1$
 $| v_pred (VEq_Const\ _ _ _) = 1$
 $| v_pred (VPred\ _ r\ _) = w\ r$
 $| v_pred (VNeg\ sp) = (s_pred\ sp) + 1$
 $| v_pred (VOr\ vp1\ vp2) = ((v_pred\ vp1) + (v_pred\ vp2)) + 1$
 $| v_pred (VAndL\ vp1) = (v_pred\ vp1) + 1$
 $| v_pred (VAndR\ vp2) = (v_pred\ vp2) + 1$
 $| v_pred (VImp\ sp1\ vp2) = ((s_pred\ sp1) + (v_pred\ vp2)) + 1$
 $| v_pred (VIffSV\ sp1\ vp2) = ((s_pred\ sp1) + (v_pred\ vp2)) + 1$
 $| v_pred (VIffVS\ vp1\ sp2) = ((v_pred\ vp1) + (s_pred\ sp2)) + 1$
 $| v_pred (VExists\ _ part) = (sum_proofs\ v_pred\ (vals\ part)) + 1$
 $| v_pred (VForall\ _ _ vp) = (v_pred\ vp) + 1$
 $| v_pred (VPrev\ vp) = (v_pred\ vp) + 1$
 $| v_pred (VPrevZ) = 1$
 $| v_pred (VPrevOutL\ _) = 1$
 $| v_pred (VPrevOutR\ _) = 1$
 $| v_pred (VNext\ vp) = (v_pred\ vp) + 1$
 $| v_pred (VNextOutL\ _) = 1$
 $| v_pred (VNextOutR\ _) = 1$
 $| v_pred (VOnceOut\ _) = 1$
 $| v_pred (VOnce\ _ _ vps) = (sum_proofs\ v_pred\ vps) + 1$
 $| v_pred (VEventually\ _ _ vps) = (sum_proofs\ v_pred\ vps) + 1$
 $| v_pred (VHistorically\ _ vp) = (v_pred\ vp) + 1$
 $| v_pred (VAlways\ _ vp) = (v_pred\ vp) + 1$
 $| v_pred (VSinceOut\ _) = 1$
 $| v_pred (VSince\ _ vp1\ vp2s) = (sum_proofs\ v_pred\ (vp1\ \#\ vp2s)) + 1$
 $| v_pred (VSinceInf\ _ _ vp2s) = (sum_proofs\ v_pred\ vp2s) + 1$
 $| v_pred (VUntil\ _ vp2s\ vp1) = (sum_proofs\ v_pred\ (vp2s\ @\ [vp1])) + 1$
 $| v_pred (VUntilInf\ _ _ vp2s) = (sum_proofs\ v_pred\ vp2s) + 1$
 $\langle proof \rangle$

termination

$\langle proof \rangle$

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

$p_pred = case_sum\ s_pred\ v_pred$

end

10.5 v_check_exec setup

lemma $ETP_minus_le_iff$: $ETP\ \sigma\ (\tau\ \sigma\ i - n) \leq j \iff \delta\ \sigma\ i\ j \leq n$
 $\langle proof \rangle$

lemma $ETP_minus_gt_iff$: $j < ETP\ \sigma\ (\tau\ \sigma\ i - n) \iff \delta\ \sigma\ i\ j > n$
 $\langle proof \rangle$

lemma $nat_le_iff_less$:

fixes $n :: nat$

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

$\langle proof \rangle$

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 $check_upt_LTP_p\ \sigma\ I\ li\ xs\ i = check_upt_LTP_p\ \sigma'\ I\ li\ xs\ i$
 ⟨proof⟩

lemma *check_upt_LTP_p_eq*:
 assumes $\tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 shows $xs = [li..<Suc\ (LTP_p\ \sigma\ i\ I)] \longleftrightarrow check_upt_LTP_p\ \sigma\ I\ li\ xs\ i$
 ⟨proof⟩

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 $\infty \Rightarrow li = 0 \mid enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b)$)
 $\wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge check_upt_LTP_p\ \sigma\ I\ li\ (map\ v_at\ vps)\ i \wedge Ball\ (set\ vps)\ (v_check_exec\ \sigma\ vs\ \varphi)$)

| $VOnceOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
| $_ \Rightarrow False$
<proof>

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

$SHistorically\ i\ li\ vps \Rightarrow$
 $(case\ right\ I\ of\ \infty \Rightarrow li = 0\ |\ enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b))$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ check_upt_LTP_p\ \sigma\ I\ li\ (map\ s_at\ vps)\ i \wedge Ball\ (set\ vps)\ (s_check_exec\ \sigma\ vs\ \varphi)$
| $SHistoricallyOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
| $_ \Rightarrow False$
<proof>

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

$VSince\ i\ vp1\ vp2s \Rightarrow$
 $let\ j = v_at\ vp1\ in$
 $(case\ right\ I\ of\ \infty \Rightarrow True\ |\ enat\ b \Rightarrow \delta\ \sigma\ i\ j \leq b) \wedge j \leq i$
 $\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
 $\wedge\ check_upt_LTP_p\ \sigma\ I\ j\ (map\ v_at\ vp2s)\ i$
 $\wedge\ v_check_exec\ \sigma\ vs\ vp1 \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
| $VSinceInf\ i\ li\ vp2s \Rightarrow$
 $(case\ right\ I\ of\ \infty \Rightarrow li = 0\ |\ enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b)) \wedge$
 $\tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \wedge$
 $check_upt_LTP_p\ \sigma\ I\ li\ (map\ v_at\ vp2s)\ i \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
| $VSinceOut\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I$
| $_ \Rightarrow False$
<proof>

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

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

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

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

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

lemma `s_check_exec_Always_code`[code]: $s_check_exec\ \sigma\ vs\ (Formula.Always\ I\ \varphi)\ sp = (case\ sp\ of$
 $SAlways\ i\ hi\ sps\ \Rightarrow$
 $(case\ right\ I\ of\ \infty\ \Rightarrow\ False\ |\ enat\ b\ \Rightarrow\ (\delta\ \sigma\ hi\ i\ \leq\ b\ \wedge\ b\ <\ \delta\ \sigma\ (Suc\ hi)\ i))$
 $\wedge\ check_upt_ETP_f\ \sigma\ I\ i\ (map\ s_at\ sps)\ hi\ \wedge\ Ball\ (set\ sps)\ (s_check_exec\ \sigma\ vs\ \varphi)$
 $|\ _ \Rightarrow\ False)$
 $\langle proof \rangle$

lemma `v_check_exec_Until_code`[code]: $v_check_exec\ \sigma\ vs\ (Formula.Until\ \varphi\ I\ \psi)\ vp = (case\ vp\ of$
 $VUntil\ i\ vp2s\ vp1\ \Rightarrow$
 $let\ j = v_at\ vp1\ in$
 $(case\ right\ I\ of\ \infty\ \Rightarrow\ True\ |\ enat\ b\ \Rightarrow\ j < LTP_f\ \sigma\ i\ b)$
 $\wedge\ i \leq j \wedge check_upt_ETP_f\ \sigma\ I\ i\ (map\ v_at\ vp2s)\ j$
 $\wedge\ v_check_exec\ \sigma\ vs\ \varphi\ vp1 \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
 $|\ VUntilInf\ i\ hi\ vp2s\ \Rightarrow$
 $(case\ right\ I\ of\ \infty\ \Rightarrow\ False\ |\ enat\ b\ \Rightarrow\ (\delta\ \sigma\ hi\ i\ \leq\ b\ \wedge\ b < \delta\ \sigma\ (Suc\ hi)\ i)) \wedge$
 $check_upt_ETP_f\ \sigma\ I\ i\ (map\ v_at\ vp2s)\ hi \wedge Ball\ (set\ vp2s)\ (v_check_exec\ \sigma\ vs\ \psi)$
 $|\ _ \Rightarrow\ False)$
 $\langle proof \rangle$

10.6 ETP/LTP setup

lemma `ETP_aux`: $\neg t \leq \tau\ \sigma\ i \implies i \leq (LEAST\ i.\ t \leq \tau\ \sigma\ i)$
 $\langle proof \rangle$

function `ETP_rec` **where**

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

termination

$\langle proof \rangle$

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

lemma `ETP_code`[code]: $ETP\ \sigma\ t = ETP_rec\ \sigma\ t\ 0$
 $\langle proof \rangle$

lemma `LTP_aux`:

assumes $\tau\ \sigma\ (Suc\ i) \leq t$
shows $i \leq Max\ \{i.\ \tau\ \sigma\ i \leq t\}$

$\langle proof \rangle$

function (sequential) `LTP_rec` **where**

$LTP_rec\ \sigma\ t\ i = (if\ \tau\ \sigma\ (Suc\ i) \leq t\ then\ LTP_rec\ \sigma\ t\ (i + 1)\ else\ i)$
 $\langle proof \rangle$

termination

$\langle proof \rangle$

lemma `LTP_rec_sound`: $LTP_rec\ \sigma\ t\ j = Max\ (\{i.\ i \geq j \wedge (\tau\ \sigma\ i) \leq t\} \cup \{j\})$
 $\langle proof \rangle$

lemma `LTP_code`[code]: $LTP\ \sigma\ t = (if\ t < \tau\ \sigma\ 0$
 $then\ Code.abort\ (STR\ ''LTP:\ undefined'')\ (\lambda\ _.\ LTP\ \sigma\ t)$
 $else\ LTP_rec\ \sigma\ t\ 0)$
 $\langle proof \rangle$

lemma `map_part_code`[code]: $Rep_part\ (map_part\ f\ xs) = map\ (map_prod\ id\ f)\ (Rep_part\ xs)$
 $\langle proof \rangle$

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>

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>

10.7 Exported functions

type_synonym name = string8

declare Formula.future_bounded.simps[code]

definition collect_paths :: ('n, 'd :: {default, linorder}) trace \Rightarrow ('n, 'd) formula \Rightarrow ('n, 'd) expl \Rightarrow 'd set list set option **where**
 collect_paths $\sigma \varphi e =$ (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 \times event_data list) set \times nat) list \Rightarrow (name, event_data)
 trace **where**
 trace_of_list_specialized xs = trace_of_list xs

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

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

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

definition sub_nat :: nat \Rightarrow nat \Rightarrow nat **where**
 sub_nat m n = m - n

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

export_code interval enat nat_of_integer integer_of_nat
 STT Formula.TT Inl EInt Formula.Var Leaf set part_hd sum_nat sub_nat subsvals
 check trace_of_list_specialized specialized_set ed_set abs_part

collect_paths_specialized
in *OCaml module_name Checker file_prefix checker*

11 Unverified Explanation-Producing Monitoring Algorithm

fun *merge_part2_raw* :: ('a ⇒ 'b ⇒ 'c) ⇒ ('d set × 'a) list ⇒ ('d set × 'b) list ⇒ ('d set × 'c) list
where

merge_part2_raw f [] _ = []
| *merge_part2_raw* f ((P1, v1) # part1) part2 =
 (let part12 = List.map_filter (λ(P2, v2). if P1 ∩ P2 ≠ {} then Some(P1 ∩ P2, f v1 v2) else None)
 part2 in
 let part2not1 = List.map_filter (λ(P2, v2). if P2 - P1 ≠ {} then Some(P2 - P1, v2) else None)
 part2 in
 part12 @ (*merge_part2_raw* f part1 part2not1))

fun *merge_part3_raw* :: ('a ⇒ 'b ⇒ 'c ⇒ 'e) ⇒ ('d set × 'a) list ⇒ ('d set × 'b) list ⇒ ('d set × 'c)
list ⇒ ('d set × 'e) list **where**

merge_part3_raw f [] _ _ = []
| *merge_part3_raw* f _ [] _ = []
| *merge_part3_raw* f _ _ [] = []
| *merge_part3_raw* f part1 part2 part3 = *merge_part2_raw* (λpt3 f'. f' pt3) part3 (*merge_part2_raw* f
part1 part2)

lemma *partition_on_empty_iff*:

partition_on X P ⇒ P = {} ↔ X = {}
partition_on X P ⇒ P ≠ {} ↔ X ≠ {}
⟨proof⟩

lemma *wf_part_list_filter_inter*:

defines *inP1* P1 f v1 part2
≡ List.map_filter (λ(P2, v2). if P1 ∩ P2 ≠ {} then Some(P1 ∩ P2, f v1 v2) else None) part2
assumes *partition_on* X (set (map fst ((P1, v1) # part1)))
and *partition_on* X (set (map fst part2))
shows *partition_on* P1 (set (map fst (*inP1* P1 f v1 part2)))
and *distinct* (map fst ((P1, v1) # part1)) ⇒ *distinct* (map fst (part2)) ⇒
distinct (map fst (*inP1* P1 f v1 part2))
⟨proof⟩

lemma *wf_part_list_filter_minus*:

defines *notinP2* P1 f v1 part2
≡ List.map_filter (λ(P2, v2). if P2 - P1 ≠ {} then Some(P2 - P1, v2) else None) part2
assumes *partition_on* X (set (map fst ((P1, v1) # part1)))
and *partition_on* X (set (map fst part2))
shows *partition_on* (X - P1) (set (map fst (*notinP2* P1 f v1 part2)))
and *distinct* (map fst ((P1, v1) # part1)) ⇒ *distinct* (map fst (part2)) ⇒
distinct (map fst (*notinP2* P1 f v1 part2))
⟨proof⟩

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) ⇒ *partition_on* Y (set ys) ⇒ X ∩ Y = {} ⇒
partition_on (X ∪ Y) (set (xs @ ys))

<proof>

lemma *wf_part_list_merge_part2_raw*:

partition_on X (set (map fst part1)) ∧ distinct (map fst part1) ⇒
partition_on X (set (map fst part2)) ∧ distinct (map fst part2) ⇒
partition_on X (set (map fst (merge_part2_raw f part1 part2)))
∧ distinct (map fst (merge_part2_raw f part1 part2))

<proof>

lemma *wf_part_list_merge_part3_raw*:

partition_on X (set (map fst part1)) ∧ distinct (map fst part1) ⇒
partition_on X (set (map fst part2)) ∧ distinct (map fst part2) ⇒
partition_on X (set (map fst part3)) ∧ distinct (map fst part3) ⇒
partition_on X (set (map fst (merge_part3_raw f part1 part2 part3)))
∧ distinct (map fst (merge_part3_raw f part1 part2 part3))

<proof>

lift_definition *merge_part2* :: ('a ⇒ 'a ⇒ 'a) ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒ ('d, 'a) part **is**
merge_part2_raw

<proof>

lift_definition *merge_part3* :: ('a ⇒ 'a ⇒ 'a ⇒ 'a) ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒ ('d, 'a) part ⇒
(('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**

$do_neg\ p = (case\ p\ of$
 $\quad Inl\ sp \Rightarrow [Inr\ (VNeg\ sp)]$
 $| Inr\ vp \Rightarrow [Inl\ (SNeg\ vp)])$

definition $do_or :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$
 $do_or\ p1\ p2 = (case\ (p1, p2)\ of$
 $\quad (Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SOrL\ sp1), Inl\ (SOrR\ sp2)]$
 $| (Inl\ sp1, Inr\ _) \Rightarrow [Inl\ (SOrL\ sp1)]$
 $| (Inr\ _ , Inl\ sp2) \Rightarrow [Inl\ (SOrR\ sp2)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inr\ (VOr\ vp1\ vp2)])$

definition $do_and :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$
 $do_and\ p1\ p2 = (case\ (p1, p2)\ of$
 $\quad (Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SAnd\ sp1\ sp2)]$
 $| (Inl\ _ , Inr\ vp2) \Rightarrow [Inr\ (VAndR\ vp2)]$
 $| (Inr\ vp1, Inl\ _) \Rightarrow [Inr\ (VAndL\ vp1)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inr\ (VAndL\ vp1), Inr\ (VAndR\ vp2)])$

definition $do_imp :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$
 $do_imp\ p1\ p2 = (case\ (p1, p2)\ of$
 $\quad (Inl\ _ , Inl\ sp2) \Rightarrow [Inl\ (SImpR\ sp2)]$
 $| (Inl\ sp1, Inr\ vp2) \Rightarrow [Inr\ (VImp\ sp1\ vp2)]$
 $| (Inr\ vp1, Inl\ sp2) \Rightarrow [Inl\ (SImpL\ vp1), Inl\ (SImpR\ sp2)]$
 $| (Inr\ vp1, Inr\ _) \Rightarrow [Inl\ (SImpL\ vp1)]$

definition $do_iff :: ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$
 $do_iff\ p1\ p2 = (case\ (p1, p2)\ of$
 $\quad (Inl\ sp1, Inl\ sp2) \Rightarrow [Inl\ (SIffSS\ sp1\ sp2)]$
 $| (Inl\ sp1, Inr\ vp2) \Rightarrow [Inr\ (VIffSV\ sp1\ vp2)]$
 $| (Inr\ vp1, Inl\ sp2) \Rightarrow [Inr\ (VIffVS\ vp1\ sp2)]$
 $| (Inr\ vp1, Inr\ vp2) \Rightarrow [Inl\ (SIffVV\ vp1\ vp2)])$

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

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

definition $do_prev :: nat \Rightarrow \mathcal{I} \Rightarrow nat \Rightarrow ('n, 'd)\ proof \Rightarrow ('n, 'd)\ proof\ list\ \mathbf{where}$
 $do_prev\ i\ I\ t\ p = (case\ (p, t < left\ I)\ of$
 $\quad (Inl\ _ , True) \Rightarrow [Inr\ (VPrevOutL\ i)]$
 $| (Inl\ sp, False) \Rightarrow (if\ mem\ t\ I\ then\ [Inl\ (SPrev\ sp)]\ else\ [Inr\ (VPrevOutR\ i)])$

| (*Inr vp*, *True*) \Rightarrow [*Inr (VPrev vp)*, *Inr (VPrevOutL i)*]
| (*Inr vp*, *False*) \Rightarrow (*if mem t I then [Inr (VPrev vp)] else [Inr (VPrev vp), Inr (VPrevOutR i)]*)

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

definition *do_eventually_base* :: *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')) ⇒ [Inl (SHistorically i li sps')]
 | (Inr _ , False, Inr (VHistorically _ vp')) ⇒ [Inr (VHistorically i vp')]

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

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

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

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

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

do_since_base i a p1 p2 = (case (p1, p2, a = 0) of
 (_ , Inl sp2, True) ⇒ [Inl (SSince sp2 [])]
 | (Inl _ , _ , False) ⇒ [Inr (VSinceInf i i [])]
 | (Inl _ , Inr vp2, True) ⇒ [Inr (VSinceInf i i [vp2])]
 | (Inr vp1, _ , False) ⇒ [Inr (VSince i vp1 []), Inr (VSinceInf i i [])]
 | (Inr vp1, Inr sp2, True) ⇒ [Inr (VSince i vp1 [sp2]), Inr (VSinceInf i i [sp2])]

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

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

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

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

    apply_pdt2 vs f (Node x part1) (Node y part2))
| apply_pdt2 [] _ (Node _ _) (Node _ _) = undefined

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

```

```

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

```

context

```

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

```

```

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

```

```

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

```

fun dist where

```

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

```

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

⟨proof⟩

termination eval

⟨proof⟩

end

end

12 Examples

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

monitor π φ = map (λi. eval (trace_of_list π) (λp q. size p ≤ size q) (sorted_list_of_set (fv φ)) i φ) [0 ..< length π]

definition *check* :: (('n :: linorder × 'd :: {default, linorder} list) set × nat) list ⇒ ('n, 'd) formula ⇒ bool **where**

check π φ = list_all (check_all (trace_of_list π) φ) (monitor π φ)

12.1 Infinite Domain

definition *prefix* :: (string × string list) set × nat) list **where**
prefix =

```
[({("mgr_S", ["Mallory", "Alice"]),
  ("mgr_S", ["Merlin", "Bob"]),
  ("mgr_S", ["Merlin", "Charlie"])}), 1307532861::nat),
({("approve", ["Mallory", "152"])}), 1307532861),
({("approve", ["Merlin", "163"]),
  ("publish", ["Alice", "160"])}),
  ("mgr_F", ["Merlin", "Charlie"])}), 1307955600),
({("approve", ["Merlin", "187"]),
  ("publish", ["Bob", "163"]),
  ("publish", ["Alice", "163"]),
  ("publish", ["Charlie", "163"]),
  ("publish", ["Charlie", "152"])}), 1308477599)]
```

definition *phi* :: (string, string) Formula.formula **where**

```
phi = Formula.Imp (Formula.Pred "publish" [Formula.Var "a", Formula.Var "f"])
  (Formula.Once (init 604800) (Formula.Exists "m" (Formula.Since
    (Formula.Neg (Formula.Pred "mgr_F" [Formula.Var "m", Formula.Var "a"])) all
    (Formula.And (Formula.Pred "mgr_S" [Formula.Var "m", Formula.Var "a"])
      (Formula.Pred "approve" [Formula.Var "m", Formula.Var "f"]))))))
```

value *monitor prefix phi*

lemma *check prefix phi*

⟨proof⟩

12.2 Finite Domain

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

definition *ord* :: Domain ⇒ nat **where**

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

```

instantiation Domain :: default begin
definition default_Domain = Default
instance <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 ≤ ord d')
definition less_Domain d d' = (ord d < ord d')
instance <proof>
end

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

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

value monitor fprefix fphi
lemma check fprefix fphi
  <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.