A Verified Proof Checker for Metric First-Order Temporal Logic

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

March 17, 2025

Abstract

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

Contents

1 Traces and Trace Prefixes

1.1 Infinite Traces

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

lemma *ssorted_siterate*[*simp*]: $(\bigwedge n.\ n \leq f\ n) \Longrightarrow ssorted\ (siterate\ f\ n)$
⟨*proof*⟩

lemma *ssortedD*: $ssorted\ s \Longrightarrow s\ !!\ i \leq stl\ s\ !!\ i$
⟨*proof*⟩

lemma *ssorted_sdrop*: $ssorted\ s \Longrightarrow ssorted\ (sdrop\ i\ s)$
⟨*proof*⟩

lemma *ssorted_monoD*: $ssorted\ s \Longrightarrow i \leq j \Longrightarrow s\ !!\ i \leq s\ !!\ j$
⟨*proof*⟩

lemma *sorted_stake*: $ssorted\ s \Longrightarrow sorted\ (stake\ i\ s)$
⟨*proof*⟩

lemma *ssorted_monoI*: $\forall i\ j.\ i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j \Longrightarrow ssorted\ s$
⟨*proof*⟩

lemma *ssorted_iff_mono*: $ssorted\ s \longleftrightarrow (\forall i\ j.\ i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j)$
⟨*proof*⟩

lemma *ssorted_iff_le_Suc*: $ssorted\ s \longleftrightarrow (\forall i.\ s\ !!\ i \leq s\ !!\ Suc\ i)$
⟨*proof*⟩

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

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

$\langle proof \rangle$

lemma *sincreasing_grD*:
fixes x :: $'a$ :: *semilattice_sup*
assumes *sincreasing* s
shows $\exists j{>}i.\ x < s\ !!\ j$
$\langle proof \rangle$

lemma *sincreasing_siterate_nat*[*simp*]:
fixes n :: *nat*
assumes $(\bigwedge n.\ n < f\ n)$
shows *sincreasing* (*siterate* f n)
$\langle proof \rangle$

lemma *sincreasing_stl*: *sincreasing* $s \Longrightarrow$ *sincreasing* (*stl* s) **for** s :: $'a$ :: *semilattice_sup stream*
$\langle proof \rangle$

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

lemma *sfiniteI*: $(\bigwedge i.\ \mathit{finite}\ (s\ !!\ i)) \Longrightarrow \mathit{sfinite}\ s$
$\langle proof \rangle$

typedef $'a$ *trace* = $\{s$ :: $('a\ \mathit{set} \times \mathit{nat})$ *stream*. *ssorted* (*smap snd* s) $\wedge$ *sincreasing* (*smap snd* s) $\wedge$ *sfinite* (*smap fst* s)$\}$
$\langle proof \rangle$

setup_lifting *type_definition_trace*

lift_definition Γ :: $'a$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *set* **is**
$\lambda s\ i.\ \mathit{fst}\ (s\ !!\ i)$ $\langle proof \rangle$
lift_definition τ :: $'a$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **is**
$\lambda s\ i.\ \mathit{snd}\ (s\ !!\ i)$ $\langle proof \rangle$

lemma *stream_eq_iff*: $s = s' \longleftrightarrow (\forall\, n.\ s\ !!\ n = s'\ !!\ n)$
$\langle proof \rangle$

lemma *trace_eqI*: $(\bigwedge i.\ \Gamma\ \sigma\ i = \Gamma\ \sigma'\ i) \Longrightarrow (\bigwedge i.\ \tau\ \sigma\ i = \tau\ \sigma'\ i) \Longrightarrow \sigma = \sigma'$
$\langle proof \rangle$

lemma *τ_mono*[*simp*]: $i \le j \Longrightarrow \tau\ s\ i \le \tau\ s\ j$
$\langle proof \rangle$

lemma *ex_le_τ*: $\exists j{\ge}i.\ x \le \tau\ s\ j$
$\langle proof \rangle$

lemma *le_τ_less*: $\tau\ \sigma\ i \le \tau\ \sigma\ j \Longrightarrow j < i \Longrightarrow \tau\ \sigma\ i = \tau\ \sigma\ j$
$\langle proof \rangle$

lemma *less_τD*: $\tau\ \sigma\ i < \tau\ \sigma\ j \Longrightarrow i < j$
$\langle proof \rangle$

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

1.2 Finite Trace Prefixes

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

setup_lifting *type_definition_prefix*

lift_definition *pmap_Γ* :: $('a\ set \Rightarrow 'b\ set) \Rightarrow 'a\ prefix \Rightarrow 'b\ prefix$ **is**
$\lambda f.\ map\ (\lambda(x,\ i).\ (f\ x,\ i))$
⟨*proof*⟩

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

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

lift_definition *pnil* :: $'a\ prefix$ **is** [] ⟨*proof*⟩

lift_definition *plen* :: $'a\ prefix \Rightarrow nat$ **is** *length* ⟨*proof*⟩

lift_definition *psnoc* :: $'a\ prefix \Rightarrow 'a\ set \times nat \Rightarrow 'a\ prefix$ **is**
$\lambda p\ x.\ if\ (case\ p\ of\ [] \Rightarrow 0 \mid \_ \Rightarrow snd\ (last\ p)) \le snd\ x\ then\ p\ @\ [x]\ else\ []$
⟨*proof*⟩

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

lift_definition *less_eq_prefix* :: $'a\ prefix \Rightarrow 'a\ prefix \Rightarrow bool$ **is**
$\lambda p\ q.\ \exists r.\ q = p\ @\ r$ ⟨*proof*⟩

definition *less_prefix* :: $'a\ prefix \Rightarrow 'a\ prefix \Rightarrow bool$ **where**
$less\_prefix\ x\ y = (x \le y \wedge \neg\ y \le x)$

instance
⟨*proof*⟩

end

lemma *psnoc_inject*[*simp*]:
$last\_ts\ p \le snd\ x \Longrightarrow last\_ts\ q \le snd\ y \Longrightarrow psnoc\ p\ x = psnoc\ q\ y \longleftrightarrow (p = q \wedge x = y)$
⟨*proof*⟩

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

lemma *prefix_of_pnil*[*simp*]: $prefix\_of\ pnil\ \sigma$
⟨*proof*⟩

lemma *plen_pnil*[*simp*]: $plen\ pnil = 0$
⟨*proof*⟩

lemma *plen_mono*: $\pi \le \pi' \Longrightarrow plen\ \pi \le plen\ \pi'$
⟨*proof*⟩

lemma *prefix_of_psnocE*: $prefix\_of\ (psnoc\ p\ x)\ s \Longrightarrow last\_ts\ p \le snd\ x \Longrightarrow$
$(prefix\_of\ p\ s \Longrightarrow \Gamma\ s\ (plen\ p) = fst\ x \Longrightarrow \tau\ s\ (plen\ p) = snd\ x \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *le_pnil*[*simp*]: $pnil \le \pi$
⟨*proof*⟩

lift_definition *take_prefix* :: $nat \Rightarrow 'a\ trace \Rightarrow 'a\ prefix$ **is** *stake*
⟨*proof*⟩

lemma *plen_take_prefix*[*simp*]: $plen\ (take\_prefix\ i\ \sigma) = i$
⟨*proof*⟩

lemma *plen_psnoc*[*simp*]: $last\_ts\ \pi \leq snd\ x \Longrightarrow plen\ (psnoc\ \pi\ x) = plen\ \pi + 1$
⟨*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*: $\pi \leq \pi' \Longrightarrow prefix\_of\ \pi'\ s \Longrightarrow prefix\_of\ \pi\ s$
⟨*proof*⟩

lemma *prefix_of_imp_linear*: $prefix\_of\ \pi\ \sigma \Longrightarrow prefix\_of\ \pi'\ \sigma \Longrightarrow \pi \leq \pi' \lor \pi' \leq \pi$
⟨*proof*⟩

lemma *τ_prefix_conv*: $prefix\_of\ p\ s \Longrightarrow prefix\_of\ p\ s' \Longrightarrow i < plen\ p \Longrightarrow \tau\ s\ i = \tau\ s'\ i$
⟨*proof*⟩

lemma *Γ_prefix_conv*: $prefix\_of\ p\ s \Longrightarrow prefix\_of\ p\ s' \Longrightarrow i < plen\ p \Longrightarrow \Gamma\ s\ i = \Gamma\ s'\ i$
⟨*proof*⟩

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

lemma *ssorted_shift*:
$ssorted\ (xs\ @-\ s) = (sorted\ xs \land ssorted\ s \land (\forall x \in set\ xs.\ \forall y \in sset\ s.\ x \leq y))$
⟨*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*]: $pts\ (pmap\_\Gamma\ f\ \pi) = pts\ \pi$
⟨*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**
$max\_opt\ a\ b = (case\ (a,b)\ of\ (Some\ x,\ Some\ y) \Rightarrow Some\ (max\ x\ y) \mid \_ \Rightarrow None)$

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_δ*: $i \geq ETP\ \sigma\ (\tau\ \sigma\ l + n) \Longrightarrow \delta\ \sigma\ i\ l \geq n$
⟨*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*⟩

lemma *etp_0*[*simp*]: $ETP\ \sigma\ 0 = 0$
⟨*proof*⟩

2 Regular expressions

context begin

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

lemma *finite_atms*[*simp*]: *finite* (*atms r*)
⟨*proof*⟩

definition *Wild* = *Skip 1*

lemma *size_regex_estimation*[*termination_simp*]: $x \in atms\ r \Longrightarrow y < f\ x \Longrightarrow y < size\_regex\ f\ r$

$\langle proof \rangle$

lemma *size_regex_estimation'*[*termination_simp*]: $x \in atms\ r \Longrightarrow y \leq f\ x \Longrightarrow y \leq size\_regex\ f\ r$
$\langle proof \rangle$ **definition** $TimesL\ r\ S = Times\ r\ `\ S$
qualified definition $TimesR\ R\ s = (\lambda r.\ Times\ r\ s)\ `\ R$

qualified primrec *collect* **where**
$collect\ f\ (Skip\ n) = \{\}$
$|\ collect\ f\ (Test\ \varphi) = f\ \varphi$
$|\ collect\ f\ (Plus\ r\ s) = collect\ f\ r \cup collect\ f\ s$
$|\ collect\ f\ (Times\ r\ s) = collect\ f\ r \cup collect\ f\ s$
$|\ collect\ f\ (Star\ r) = collect\ f\ r$

lemma *collect_cong*[*fundef_cong*]:
$r = r' \Longrightarrow (\bigwedge z.\ z \in atms\ r \Longrightarrow f\ z = f'\ z) \Longrightarrow collect\ f\ r = collect\ f'\ r'$
$\langle proof \rangle$

lemma *finite_collect*[*simp*]: $(\bigwedge z.\ z \in atms\ r \Longrightarrow finite\ (f\ z)) \Longrightarrow finite\ (collect\ f\ r)$
$\langle proof \rangle$

lemma *collect_commute*:
$(\bigwedge z.\ z \in atms\ r \Longrightarrow x \in f\ z \longleftrightarrow g\ x \in f'\ z) \Longrightarrow x \in collect\ f\ r \longleftrightarrow g\ x \in collect\ f'\ r$
$\langle proof \rangle$

lemma *collect_alt*: $collect\ f\ r = (\bigcup z \in atms\ r.\ f\ z)$
$\langle proof \rangle$ **definition** *ncollect* **where**
$ncollect\ f\ r = Max\ (insert\ 0\ (Suc\ `\ collect\ f\ r))$

lemma *insert_Un*: $insert\ x\ (A \cup B) = insert\ x\ A \cup insert\ x\ B$
$\langle proof \rangle$

lemma *ncollect_simps*[*simp*]:
assumes [*simp*]: $(\bigwedge z.\ z \in atms\ r \Longrightarrow finite\ (f\ z))\ (\bigwedge z.\ z \in atms\ s \Longrightarrow finite\ (f\ z))$
shows
$ncollect\ f\ (Skip\ n) = 0$
$ncollect\ f\ (Test\ \varphi) = Max\ (insert\ 0\ (Suc\ `\ f\ \varphi))$
$ncollect\ f\ (Plus\ r\ s) = max\ (ncollect\ f\ r)\ (ncollect\ f\ s)$
$ncollect\ f\ (Times\ r\ s) = max\ (ncollect\ f\ r)\ (ncollect\ f\ s)$
$ncollect\ f\ (Star\ r) = ncollect\ f\ r$
$\langle proof \rangle$

abbreviation $min\_regex\_default\ f\ r\ j \equiv (if\ atms\ r = \{\}\ then\ j\ else\ Min\ ((\lambda z.\ f\ z\ j)\ `\ atms\ r))$

qualified primrec $match :: (nat \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ regex \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**
$match\ test\ (Skip\ n) = (\lambda i\ j.\ j = i + n)$
$|\ match\ test\ (Test\ \varphi) = (\lambda i\ j.\ i = j \land test\ i\ \varphi)$
$|\ match\ test\ (Plus\ r\ s) = match\ test\ r \sqcup match\ test\ s$
$|\ match\ test\ (Times\ r\ s) = match\ test\ r\ OO\ match\ test\ s$
$|\ match\ test\ (Star\ r) = (match\ test\ r)^{**}$

lemma *match_cong*[*fundef_cong*]:
$r = r' \Longrightarrow (\bigwedge i\ z.\ z \in atms\ r \Longrightarrow t\ i\ z = t'\ i\ z) \Longrightarrow match\ t\ r = match\ t'\ r'$
$\langle proof \rangle$

lemma *match_le*: $match\ test\ r\ i\ j \Longrightarrow i \leq j$
$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *match_map_regex*: $match\ t\ (map\_regex\ f\ r) = match\ (\lambda k\ z.\ t\ k\ (f\ z))\ r$
$\langle proof \rangle$

lemma *match_mono_strong*:
$(\bigwedge k\ z.\ k \in \{i\ ..<\ j + 1\} \Longrightarrow z \in atms\ r \Longrightarrow t\ k\ z \Longrightarrow t'\ k\ z) \Longrightarrow match\ t\ r\ i\ j \Longrightarrow match\ t'\ r\ i\ j$
$\langle proof \rangle$

lemma *match_cong_strong*:
$(\bigwedge k\ z.\ k \in \{i\ ..<\ j + 1\} \Longrightarrow z \in atms\ r \Longrightarrow t\ k\ z = t'\ k\ z) \Longrightarrow match\ t\ r\ i\ j = match\ t'\ r\ i\ j$
$\langle proof \rangle$

end

3 Metric First-Order Temporal Logic

3.1 Syntax

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

datatype $(fv\_trm\text{: } 'n, 'a)\ trm = is\_Var\text{: } Var\ 'n\ (\langle \mathbf{v} \rangle) \mid is\_Const\text{: } Const\ 'a\ (\langle \mathbf{c} \rangle)$

lemma *in_fv_trm_conv*: $x \in fv\_trm\ t \longleftrightarrow t = \mathbf{v}\ x$
$\langle proof \rangle$

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

fun $fv :: ('n, 'a)\ formula \Rightarrow 'n\ set$ **where**
$fv\ (r \dagger ts) = \bigcup\ (fv\_trm\ `\ set\ ts)$
$\mid fv\ \top = \{\}$

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

fun *consts* :: $('n, 'a)$ *formula* $\Rightarrow$ *'a set* **where**
consts $(r \dagger ts)$ = {} — terms may also contain constants, but these only filter out values from the trace and do not introduce new values of interest (i.e., do not extend the active domain)
| *consts* $\top$ = {}
| *consts* $\bot$ = {}
| *consts* $(x \approx c)$ = $\{c\}$
| *consts* $(\neg_F\ \varphi)$ = *consts* φ
| *consts* $(\varphi \lor_F \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\varphi \land_F \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\varphi \longrightarrow_F \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\varphi \longleftrightarrow_F \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\exists_F x.\ \varphi)$ = *consts* φ
| *consts* $(\forall_F x.\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{Y}\ I\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{X}\ I\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{P}\ I\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{H}\ I\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{F}\ I\ \varphi)$ = *consts* φ
| *consts* $(\mathbf{G}\ I\ \varphi)$ = *consts* φ
| *consts* $(\varphi\ \mathbf{S}\ I\ \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\varphi\ \mathbf{U}\ I\ \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\triangleleft\ I\ r)$ = *Regex.collect consts r*
| *consts* $(\triangleright\ I\ r)$ = *Regex.collect consts r*

lemma *finite_fv_trm*[*simp*]: *finite* (*fv_trm t*)
⟨*proof*⟩

lemma *finite_fv*[*simp*]: *finite* (*fv* φ)
⟨*proof*⟩

lemma *finite_consts*[*simp*]: *finite* (*consts* φ)
⟨*proof*⟩

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

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

| $future\_bounded\ \bot = True$
| $future\_bounded\ (\_ \dagger \_) = True$
| $future\_bounded\ (\_ \approx \_) = True$
| $future\_bounded\ (\neg_F\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\varphi \vee_F \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi)$
| $future\_bounded\ (\varphi \wedge_F \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi)$
| $future\_bounded\ (\varphi \longrightarrow_F \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi)$
| $future\_bounded\ (\varphi \longleftrightarrow_F \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi)$
| $future\_bounded\ (\exists_F\_.\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\forall_F\_.\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\mathbf{Y}\ I\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\mathbf{X}\ I\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\mathbf{P}\ I\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\mathbf{H}\ I\ \varphi) = future\_bounded\ \varphi$
| $future\_bounded\ (\mathbf{F}\ I\ \varphi) = (future\_bounded\ \varphi \wedge right\ I \neq \infty)$
| $future\_bounded\ (\mathbf{G}\ I\ \varphi) = (future\_bounded\ \varphi \wedge right\ I \neq \infty)$
| $future\_bounded\ (\varphi\ \mathbf{S}\ I\ \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi)$
| $future\_bounded\ (\varphi\ \mathbf{U}\ I\ \psi) = (future\_bounded\ \varphi \wedge future\_bounded\ \psi \wedge right\ I \neq \infty)$
| $future\_bounded\ (\triangleleft\ I\ r) = Regex.pred\_regex\ future\_bounded\ r$
| $future\_bounded\ (\triangleright\ I\ r) = (Regex.pred\_regex\ future\_bounded\ r \wedge right\ I \neq \infty)$

3.2 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 $eval\_trms\_set :: ('n, 'a)\ envset \Rightarrow ('n, 'a)\ trm\ list \Rightarrow (('n, 'a)\ trm \times 'a\ set)\ list\ (‹\_\{\!\{\_\}\!\}›$
$[70,89]\ 89)$
where $eval\_trms\_set\ vs\ ts = map\ (eval\_trm\_set\ vs)\ ts$

lemma $eval\_trms\_set\_Nil$: $vs\{\!\{[]\}\!\} = []$
$\langle proof \rangle$

lemma $eval\_trms\_set\_Cons$:
$vs\{\!\{(t \# ts)\}\!\} = vs\{\!|t|\!\} \# vs\{\!\{ts\}\!\}$
$\langle proof \rangle$

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

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

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

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

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

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

lemma *sat_Since_pointD*: $\langle\sigma, v, i\rangle \models \varphi\ \mathbf{S}\ (\mathit{point}\ t)\ \psi \Longrightarrow \mathit{mem}\ t\ I \Longrightarrow \langle\sigma, v, i\rangle \models \varphi\ \mathbf{S}\ I\ \psi$
⟨*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*: $\langle\sigma, v, i\rangle \models \mathbf{P}\ I\ \varphi \longleftrightarrow$
$\mathit{mem}\ 0\ I \wedge \langle\sigma, v, i\rangle \models \varphi \vee$
$(i > 0 \wedge \Delta\ \sigma\ i \leq \mathit{right}\ I \wedge \langle\sigma, v, i - 1\rangle \models \mathbf{P}\ (\mathit{subtract}\ (\Delta\ \sigma\ i)\ I)\ \varphi)$
⟨*proof*⟩

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

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

$(i > 0 \longrightarrow \Delta\ \sigma\ i \leq right\ I \longrightarrow \langle\sigma,\ v,\ i - 1\rangle \models \mathbf{H}\ (subtract\ (\Delta\ \sigma\ i)\ I)\ \varphi)$
⟨*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*: $\langle\sigma,\ v,\ i\rangle \models \mathbf{F}\ I\ \varphi \longleftrightarrow$
$mem\ 0\ I \wedge \langle\sigma,\ v,\ i\rangle \models \varphi \vee$
$(\Delta\ \sigma\ (i + 1) \leq right\ I \wedge \langle\sigma,\ v,\ i + 1\rangle \models \mathbf{F}\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \varphi)$
⟨*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*: $\langle\sigma,\ v,\ i\rangle \models \mathbf{G}\ I\ \varphi \longleftrightarrow$
$(mem\ 0\ I \longrightarrow \langle\sigma,\ v,\ i\rangle \models \varphi) \wedge$
$(\Delta\ \sigma\ (i + 1) \leq right\ I \longrightarrow \langle\sigma,\ v,\ i + 1\rangle \models \mathbf{G}\ (subtract\ (\Delta\ \sigma\ (i + 1))\ I)\ \varphi)$
⟨*proof*⟩

bundle *MFOTL_syntax*
begin

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

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

For subscripts type “backslash” followed by “sub”

notation *TT* (‹⊤›)
  and *FF* (‹⊥›)
  and *Pred* (‹_ † _› [*85*, *85*] *85*)
  and *Eq_Const* (‹_ ≈ _› [*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* (‹_ **S** _ _› [*60*,*1000*,*60*] *60*)
  and *Until* (‹_ **U** _ _› [*60*,*1000*,*60*] *60*)

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

end

unbundle *no MFOTL_syntax*

4 Valued Partitions

lemma *part_list_set_eq_aux1*:
  assumes
    $\forall\, i{<}length\ xs.\ \forall\, j{<}length\ xs.\ i \neq j \longrightarrow\ fst\ (xs\ !\ i) \cap fst\ (xs\ !\ j) = \{\}$
    $\{\} \notin fst\ `\ set\ xs$
  shows $disjoint\ (fst\ `\ set\ xs) \wedge distinct\ (map\ fst\ xs)$
⟨*proof*⟩

lemma *part_list_set_eq_aux2*:
  assumes
    $disjoint\ (fst\ `\ set\ xs)$
    $distinct\ (map\ fst\ xs)$
    $i < length\ xs$
    $j < length\ xs$
    $i \neq j$
  shows $fst\ (xs\ !\ i) \cap fst\ (xs\ !\ j) = \{\}$
⟨*proof*⟩

lemma *part_list_eq*:
  $(\bigcup X \in fst\ `\ set\ xs.\ X) = UNIV$
    $\wedge (\forall\, i < length\ xs.\ \forall\, j < length\ xs.\ i \neq j$
      $\longrightarrow fst\ (xs\ !\ i) \cap fst\ (xs\ !\ j) = \{\}) \wedge \{\} \notin fst\ `\ set\ xs$
  $\longleftrightarrow partition\_on\ UNIV\ (set\ (map\ fst\ xs)) \wedge distinct\ (map\ fst\ xs)$
  ⟨*proof*⟩

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

typedef $('d,\ 'a)\ part = \{xs :: ('d\ set \times 'a)\ list.\ partition\_on\ UNIV\ (set\ (map\ fst\ xs)) \wedge distinct\ (map\ fst\ xs)\}$
  ⟨*proof*⟩

setup_lifting *type_definition_part*

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

4.1 *size* setup

lift_definition $subs :: ('d,\ 'a)\ part \Rightarrow 'd\ set\ list$ **is** *map fst* ⟨*proof*⟩

lift_definition $Subs :: ('d,\ 'a)\ part \Rightarrow 'd\ set\ set$ **is** *set o map fst* ⟨*proof*⟩

lift_definition $vals :: ('d,\ 'a)\ part \Rightarrow 'a\ list$ **is** *map snd* ⟨*proof*⟩

lift_definition $SubsVals :: ('d,\ 'a)\ part \Rightarrow ('d\ set \times 'a)\ set$ **is** *set* ⟨*proof*⟩

lift_definition $subsvals :: ('d,\ 'a)\ part \Rightarrow ('d\ set \times 'a)\ list$ **is** *id* ⟨*proof*⟩

lift_definition $size\_part :: ('d \Rightarrow nat) \Rightarrow ('a \Rightarrow nat) \Rightarrow ('d,\ 'a)\ part \Rightarrow nat$ **is** $\lambda f\ g.\ size\_list\ (\lambda(x,\ y).\ sum\ f\ x + g\ y)$ ⟨*proof*⟩

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

definition *size_part* **where**
size_part_overloaded_def: $size\_part = Partition.size\_part\ (\lambda\_.\ 0)\ (\lambda\_.\ 0)$

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*)
⟨*proof*⟩

lemma *size_part_estimation*[*termination_simp*]: *x* ∈ *Vals xs* ⟹ *y* < *g x* ⟹ *y* < *size_part f g xs*
⟨*proof*⟩

lemma *size_part_estimation′*[*termination_simp*]: *x* ∈ *Vals xs* ⟹ *y* ≤ *g x* ⟹ *y* ≤ *size_part f g xs*
⟨*proof*⟩

lemma *size_part_pointwise*[*termination_simp*]: (⋀*x*. *x* ∈ *Vals xs* ⟹ *f x* ≤ *g x*) ⟹ *size_part h f xs* ≤ *size_part h g xs*
⟨*proof*⟩

4.2 Functions on Valued Partitions

lemma *Vals_code*[*code*]: *Vals x* = *set* (*map snd* (*Rep_part x*))
⟨*proof*⟩

lemma *Vals_transfer*[*transfer_rule*]: *rel_fun* (*pcr_part* (=) (=)) (=) (*set* ∘ *map snd*) *Vals*
⟨*proof*⟩

lemma *set_vals*[*simp*]: *set* (*vals xs*) = *Vals xs*
⟨*proof*⟩

lift_definition *part_hd* :: (′*d*, ′*a*) *part* ⇒ ′*a* **is** *snd* ∘ *hd* ⟨*proof*⟩

lift_definition *tabulate* :: ′*d list* ⇒ (′*d* ⇒ ′*n*) ⇒ ′*n* ⇒ (′*d*, ′*n*) *part* **is**
λ*ds f z*. *if distinct ds then if set ds* = *UNIV then map* (λ*d*. ({*d*}, *f d*)) *ds else* (− *set ds*, *z*) # *map* (λ*d*. ({*d*}, *f d*)) *ds else* [(*UNIV*, *z*)]
⟨*proof*⟩

lift_definition *lookup_part* :: (′*d*, ′*a*) *part* ⇒ ′*d* ⇒ ′*a* **is** λ*xs d*. *snd* (*the* (*find* (λ(*D*, _). *d* ∈ *D*) *xs*))
⟨*proof*⟩

lemma *Vals_tabulate*[*simp*]: *Vals* (*tabulate xs f z*) =
(*if distinct xs then if set xs* = *UNIV then f* ‘ *set xs else* {*z*} ∪ *f* ‘ *set xs else* {*z*})
⟨*proof*⟩

lemma *lookup_part_tabulate*[*simp*]: *lookup_part* (*tabulate xs f z*) *x* =
(*if distinct xs* ∧ *x* ∈ *set xs then f x else z*)
⟨*proof*⟩

lemma *part_hd_Vals*[*simp*]: *part_hd part* ∈ *Vals part*
⟨*proof*⟩

lemma *lookup_part_Vals*[*simp*]: *lookup_part part d* ∈ *Vals part*
⟨*proof*⟩

lemma *lookup_part_SubsVals*: $\exists D.\ d \in D \wedge (D,\ lookup\_part\ part\ d) \in SubsVals\ part$
⟨*proof*⟩

lemma *lookup_part_from_subvals*: $(D,\ e) \in set\ (subsvals\ part) \Longrightarrow d \in D \Longrightarrow lookup\_part\ part\ d = e$
⟨*proof*⟩

lemma *size_lookup_part_estimation*[*termination_simp*]: $size\ (lookup\_part\ part\ d) < Suc\ (size\_part\ (\lambda\_.\ 0)\ size\ part)$
⟨*proof*⟩

lemma *subsvals_part_estimation*[*termination_simp*]: $(D,\ e) \in set\ (subsvals\ part) \Longrightarrow size\ e < Suc\ (size\_part\ (\lambda\_.\ 0)\ size\ part)$
⟨*proof*⟩

lemma *size_part_hd_estimation*[*termination_simp*]: $size\ (part\_hd\ part) < Suc\ (size\_part\ (\lambda\_.\ 0)\ size\ part)$
⟨*proof*⟩

lemma *size_last_estimation*[*termination_simp*]: $xs \neq [] \Longrightarrow size\ (last\ xs) < size\_list\ size\ xs$
⟨*proof*⟩

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

lemma *snd_lookup*[*simp*]: $snd\ (lookup\ part\ d) = lookup\_part\ part\ d$
⟨*proof*⟩

lemma *distinct_disjoint_uniq*: $distinct\ xs \Longrightarrow disjoint\ (set\ xs) \Longrightarrow$
$i < j \Longrightarrow j < length\ xs \Longrightarrow d \in xs\ !\ i \Longrightarrow d \in xs\ !\ j \Longrightarrow False$
⟨*proof*⟩

lemma *partition_on_UNIV_find_Some*:
$partition\_on\ UNIV\ (set\ (map\ fst\ part)) \Longrightarrow distinct\ (map\ fst\ part) \Longrightarrow$
$\exists y.\ find\ (\lambda(D,\ \_).\ d \in D)\ part = Some\ y$
⟨*proof*⟩

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

lemma *lookup_subsvals*: $lookup\ part\ d \in set\ (subsvals\ part)$
⟨*proof*⟩

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

lemma *part_hd_trivial*[*simp*]: $part\_hd\ (trivial\_part\ pt) = pt$
⟨*proof*⟩

lemma *SubsVals_trivial*[*simp*]: $SubsVals\ (trivial\_part\ pt) = \{(UNIV,\ pt)\}$
⟨*proof*⟩

5 Partitioned Decision Trees

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

inductive *vars_order* :: $'n\ list \Rightarrow ('d,\ 'l,\ 'n)\ pdt \Rightarrow bool$ **where**
$vars\_order\ vs\ (Leaf\ \_)$
$|\ \forall expl \in Vals\ part1.\ vars\_order\ vs\ expl \Longrightarrow vars\_order\ (x\ \#\ vs)\ (Node\ x\ part1)$

| $vars\_order\ vs\ (Node\ x\ part1) \Longrightarrow x \neq z \Longrightarrow vars\_order\ (z \ \#\ vs)\ (Node\ x\ part1)$

lemma $vars\_order\_Node$:
  assumes $distinct\ xs$
  shows $vars\_order\ xs\ (Node\ x\ part) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge (\forall\, e \in\ Vals\ part.\ vars\_order\ zs\ e))$
$\langle proof \rangle$

fun $distinct\_paths$ **where**
  $distinct\_paths\ (Leaf\ \_) = True$
| $distinct\_paths\ (Node\ x\ part) = (\forall\, e \in\ Vals\ part.\ x \notin vars\ e \wedge distinct\_paths\ e)$

fun $eval\_pdt$ **where**
  $eval\_pdt\ v\ (Leaf\ l) = l$
| $eval\_pdt\ v\ (Node\ x\ part) = eval\_pdt\ v\ (lookup\_part\ part\ (v\ x))$

lemma $eval\_pdt\_cong$: $\forall\, x \in vars\ e.\ v\ x = v'\ x \Longrightarrow\ eval\_pdt\ v\ e = eval\_pdt\ v'\ e$
  $\langle proof \rangle$

lemma $vars\_order\_vars$: $vars\_order\ vs\ e \Longrightarrow vars\ e \subseteq set\ vs$
  $\langle proof \rangle$

lemma $vars\_order\_distinct\_paths$: $vars\_order\ vs\ e \Longrightarrow distinct\ vs \Longrightarrow distinct\_paths\ e$
  $\langle proof \rangle$

lemma $eval\_pdt\_fun\_upd$: $vars\_order\ vs\ e \Longrightarrow x \notin set\ vs \Longrightarrow eval\_pdt\ (v(x := d))\ e = eval\_pdt\ v\ e$
  $\langle proof \rangle$

context begin

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

lemma $SAT\_mono[mono]$:
  assumes $X \leq Y$
  shows $SAT\ X \leq SAT\ Y$
  $\langle proof \rangle$

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

qualified inductive
  $VIO :: (nat \Rightarrow 'a \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ Regex.regex \Rightarrow bool$
  for vio **where**
  $VSkip$: $j \neq i + n \Longrightarrow VIO\ vio\ i\ j\ (Regex.Skip\ n)$
| $VTest$: $i = j \Longrightarrow vio\ i\ x \Longrightarrow VIO\ vio\ i\ j\ (Regex.Test\ x)$
| $VTest\_neq$: $i \neq j \Longrightarrow VIO\ vio\ i\ j\ (Regex.Test\ x)$
| $VPlus$: $VIO\ vio\ i\ j\ r \Longrightarrow VIO\ vio\ i\ j\ s \Longrightarrow VIO\ vio\ i\ j\ (Regex.Plus\ r\ s)$
| $VTimes$: $\forall\, k \in \{i\ ..\ j\}.\ VIO\ vio\ i\ k\ r \vee VIO\ vio\ k\ j\ s \Longrightarrow VIO\ vio\ i\ j\ (Regex.Times\ r\ s)$

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

lemma *VIO_mono*[*mono*]:
assumes $X \le Y$
shows $VIO\ X \le VIO\ Y$
⟨*proof*⟩

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

lemma
chain_Nil[*simp*]: $\neg\ chain\ R\ []$ **and**
chain_not_Nil: $chain\ R\ xs \Longrightarrow xs \neq []$
⟨*proof*⟩

lemma *chain_rtranclp*: $chain\ R\ xs \Longrightarrow R^{**}\ (hd\ xs)\ (last\ xs)$
⟨*proof*⟩

lemma *chain_append*:
assumes $chain\ R\ xs\ chain\ R\ ys\ R\ (last\ xs)\ (hd\ ys)$
shows $chain\ R\ (xs\ @\ ys)$
⟨*proof*⟩

lemma *tranclp_imp_exists_finite_chain_list*:
$R^{++}\ x\ y \Longrightarrow \exists xs.\ chain\ R\ (x\ \#\ xs\ @\ [y])$
⟨*proof*⟩

lemma *chain_pairwise*:
$chain\ R\ xs \Longrightarrow Suc\ i < length\ xs \Longrightarrow R\ (xs\ !\ i)\ (xs\ !\ Suc\ i)$
⟨*proof*⟩

lemma *chain_sorted_remdups*:
$chain\ R\ xs \Longrightarrow (\bigwedge x\ y.\ R\ x\ y \Longrightarrow x \le y) \Longrightarrow sorted\ xs \wedge chain\ R\ (remdups\ xs)$
⟨*proof*⟩

lemma *sorted_remdups*: $sorted\ xs \Longrightarrow sorted\_wrt\ (<)\ (remdups\ xs)$
⟨*proof*⟩

lemma *remdups_sorted_start_end*:
$sorted\ (i\ \#\ xs\ @\ [j]) \Longrightarrow i \neq j \Longrightarrow$
$remdups\ (i\ \#\ xs\ @\ [j]) = i\ \#\ remdups\ (removeAll\ j\ (removeAll\ i\ xs))\ @\ [j]$
⟨*proof*⟩

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

abbreviation *match_rel* **where**
$match\_rel\ test\ r\ xs\ k \equiv (xs\ !\ k < xs\ !\ (Suc\ k) \wedge Regex.match\ test\ r\ (xs\ !\ k)\ (xs\ !\ (Suc\ k)))$

lemma *list_to_chain*:

$xs \neq [] \Longrightarrow \forall k \in \{0 ..< \textit{length xs} - 1\}.\ R\ (xs\ !\ k)\ (xs\ !\ \textit{Suc}\ k) \Longrightarrow \textit{chain}\ R\ xs$
⟨*proof*⟩

lemma *match_rel_list_to_tranclp*:
$\exists xs\ zs.\ xs = i\ \#\ zs\ @\ [j] \wedge (\forall k \in \{0 ..< \textit{length xs} - 1\}.\ \textit{match\_rel test r xs k}) \Longrightarrow i \neq j \Longrightarrow (\textit{Regex.match test r})^{++}\ i\ j$
⟨*proof*⟩

lemma *completeness_SAT*:
$\forall x \in \textit{Regex.atms}\ r.\ \forall i.\ \textit{test}\ i\ x \longrightarrow \textit{sat}\ i\ x \Longrightarrow \textit{Regex.match test}\ r\ i\ j \Longrightarrow \textit{SAT sat}\ i\ j\ r$
⟨*proof*⟩

lemma *completeness_VIO*:
$\forall x \in \textit{Regex.atms}\ r.\ \forall i.\ \neg\ \textit{test}\ i\ x \longrightarrow \textit{vio}\ i\ x \Longrightarrow i \leq j \Longrightarrow \neg\ \textit{Regex.match test}\ r\ i\ j \Longrightarrow \textit{VIO vio}\ i\ j\ r$
⟨*proof*⟩

lemma *soundness_SAT*:
$\forall x \in \textit{Regex.atms}\ r.\ \forall i.\ \textit{sat}\ i\ x \longrightarrow \textit{test}\ i\ x \Longrightarrow \textit{SAT sat}\ i\ j\ r \Longrightarrow \textit{Regex.match test}\ r\ i\ j$
⟨*proof*⟩

lemma *soundness_VIO*:
$\forall x \in \textit{Regex.atms}\ r.\ \forall i.\ \textit{vio}\ i\ x \longrightarrow \neg\ \textit{test}\ i\ x \Longrightarrow i \leq j \Longrightarrow \textit{VIO vio}\ i\ j\ r \Longrightarrow \neg\ \textit{Regex.match test}\ r\ i\ j$
⟨*proof*⟩

end

6 Proof System

unbundle *MFOTL_syntax*

context begin

inductive *SAT* **and** *VIO* :: $('n, 'd)\ \textit{trace} \Rightarrow ('n, 'd)\ \textit{env} \Rightarrow \textit{nat} \Rightarrow ('n, 'd)\ \textit{formula} \Rightarrow \textit{bool}$ **for** σ **where**
STT: $\textit{SAT}\ \sigma\ v\ i\ \textit{TT}$
| *VFF*: $\textit{VIO}\ \sigma\ v\ i\ \textit{FF}$
| *SPred*: $(r, \textit{eval\_trms}\ v\ ts) \in \Gamma\ \sigma\ i \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Pred}\ r\ ts)$
| *VPred*: $(r, \textit{eval\_trms}\ v\ ts) \notin \Gamma\ \sigma\ i \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Pred}\ r\ ts)$
| *SEq_Const*: $v\ x = c \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Eq\_Const}\ x\ c)$
| *VEq_Const*: $v\ x \neq c \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Eq\_Const}\ x\ c)$
| *SNeg*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Neg}\ \varphi)$
| *VNeg*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Neg}\ \varphi)$
| *SOrL*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Or}\ \varphi\ \psi)$
| *SOrR*: $\textit{SAT}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Or}\ \varphi\ \psi)$
| *VOr*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Or}\ \varphi\ \psi)$
| *SAnd*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{And}\ \varphi\ \psi)$
| *VAndL*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{And}\ \varphi\ \psi)$
| *VAndR*: $\textit{VIO}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{And}\ \varphi\ \psi)$
| *SImpL*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Imp}\ \varphi\ \psi)$
| *SImpR*: $\textit{SAT}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Imp}\ \varphi\ \psi)$
| *VImp*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Imp}\ \varphi\ \psi)$
| *SIffSS*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Iff}\ \varphi\ \psi)$
| *SIffVV*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Iff}\ \varphi\ \psi)$
| *VIffSV*: $\textit{SAT}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Iff}\ \varphi\ \psi)$
| *VIffVS*: $\textit{VIO}\ \sigma\ v\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ \psi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Iff}\ \varphi\ \psi)$
| *SExists*: $\exists z.\ \textit{SAT}\ \sigma\ (v\ (x := z))\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Exists}\ x\ \varphi)$
| *VExists*: $\forall z.\ \textit{VIO}\ \sigma\ (v\ (x := z))\ i\ \varphi \Longrightarrow \textit{VIO}\ \sigma\ v\ i\ (\textit{Exists}\ x\ \varphi)$
| *SForall*: $\forall z.\ \textit{SAT}\ \sigma\ (v\ (x := z))\ i\ \varphi \Longrightarrow \textit{SAT}\ \sigma\ v\ i\ (\textit{Forall}\ x\ \varphi)$

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

$SAT\ \sigma\ v\ i\ (MatchF\ I\ r)$
$|\ VMatchF$: $(\bigwedge j.\ j \in (case\ right\ I\ of\ \infty \Rightarrow \{ETP\_f\ \sigma\ i\ I\ ..\}$
$|\ enat\ b \Rightarrow \{ETP\_f\ \sigma\ i\ I\ ..\ LTP\_f\ \sigma\ i\ b\}) \Longrightarrow Regex\_Proof\_System.VIO\ (VIO\ \sigma\ v)\ i\ j\ r) \Longrightarrow$
$VIO\ \sigma\ v\ i\ (MatchF\ I\ r)$

6.1 Soundness and Completeness

lemma *not_sat_SinceD*:
assumes *unsat*: $\neg\ \langle\sigma, v, i\rangle \models \varphi$ **S** $I\ \psi$ **and**
witness: $\exists j \leq i.\ mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \langle\sigma, v, j\rangle \models \psi$
shows $\exists j \leq i.\ ETP\ \sigma\ (case\ right\ I\ of\ \infty \Rightarrow 0\ |\ enat\ n \Rightarrow \tau\ \sigma\ i - n) \leq j \wedge \neg\ \langle\sigma, v, j\rangle \models \varphi$
$\wedge\ (\forall k \in \{j\ ..\ (min\ i\ (LTP\ \sigma\ (\tau\ \sigma\ i - left\ I)))\}.\ \neg\ \langle\sigma, v, k\rangle \models \psi)$
$\langle proof \rangle$

lemma *not_sat_UntilD*:
assumes *unsat*: $\neg\ \langle\sigma, v, i\rangle \models \varphi$ **U** $I\ \psi$
and *witness*: $\exists j \geq i.\ mem\ (\delta\ \sigma\ j\ i)\ I \wedge \langle\sigma, v, j\rangle \models \psi$
shows $\exists j \geq i.\ (case\ right\ I\ of\ \infty \Rightarrow True\ |\ enat\ n \Rightarrow j < LTP\ \sigma\ (\tau\ \sigma\ i + n))$
$\wedge\ \neg\ (\langle\sigma, v, j\rangle \models \varphi) \wedge (\forall k \in \{(max\ i\ (ETP\ \sigma\ (\tau\ \sigma\ i + left\ I)))\ ..\ j\}.$
$\neg\ \langle\sigma, v, k\rangle \models \psi)$
$\langle proof \rangle$

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

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

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

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

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

end

unbundle *no MFOTL_syntax*

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

lemma *size_hd_estimation[termination_simp]*: $xs \neq [] \Longrightarrow size\ (hd\ xs) < size\_list\ size\ xs$
$\langle proof \rangle$
lemma *size_last_estimation[termination_simp]*: $xs \neq [] \Longrightarrow size\ (last\ xs) < size\_list\ size\ xs$
$\langle proof \rangle$
lemma *size_rsproof_estimation[termination_simp]*: $x \in spatms\ p \Longrightarrow y < f\ x \Longrightarrow y < size\_rsproof\ f\ p$
$\langle proof \rangle$
lemma *size_rsproof_estimation'[termination_simp]*: $x \in spatms\ p \Longrightarrow y \leq f\ x \Longrightarrow y \leq size\_rsproof\ f\ p$
$\langle proof \rangle$
lemma *size_rvproof_estimation[termination_simp]*: $x \in vpatms\ p \Longrightarrow y < f\ x \Longrightarrow y < size\_rvproof\ f\ p$

⟨*proof*⟩
lemma *size_rvproof_estimation*′[*termination_simp*]: $x \in vpatms\ p \Longrightarrow y \leq f\,x \Longrightarrow y \leq size\_rvproof\,f\,p$
⟨*proof*⟩

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

lemma *rs_at_cong*[*fundef_cong*]:
$p = p' \Longrightarrow (\bigwedge x.\ x \in spatms\ p \Longrightarrow t\ x = t'\ x) \Longrightarrow rs\_at\ t\ p = rs\_at\ t'\ p'$
⟨*proof*⟩

function(*sequential*) *rv_at* **where**
$rv\_at\ test\ (VSkip\ n\ n') = (n,\ n')$
$|\ rv\_at\ test\ (VTest\ p) = (test\ p,\ test\ p)$
$|\ rv\_at\ test\ (VTest\_neq\ n\ n') = (n,\ n')$
$|\ rv\_at\ test\ (VPlus\ p1\ p2) = rv\_at\ test\ p1$
$|\ rv\_at\ test\ (VTimes\ ps) = (if\ ps = []\ then\ (0,0)\ else\ (fst\ (rv\_at\ test\ (snd\ (hd\ ps))),\ snd\ (rv\_at\ test\ (snd\ (last\ ps)))))$
$|\ rv\_at\ test\ (VStar\ ps) = (Min\ (set\ (map\ (fst \circ (rv\_at\ test))\ ps)),\ Max\ (set\ (map\ (snd \circ (rv\_at\ test))\ ps)))$
$|\ rv\_at\ test\ (VStar\_gt\ n\ n') = (n,\ n')$
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *rv_at_cong*[*fundef_cong*]:
$p = p' \Longrightarrow (\bigwedge x.\ x \in vpatms\ p \Longrightarrow t\ x = t'\ x) \Longrightarrow rv\_at\ t\ p = rv\_at\ t'\ p'$
⟨*proof*⟩

7 Proof Objects

datatype (*dead* ′*n*, *dead* ′*d*) *sproof* = *STT nat*
| *SPred nat* ′*n* (′*n*, ′*d*) *Formula.trm list*
| *SEq_Const nat* ′*n* ′*d*
| *SNeg* (′*n*, ′*d*) *vproof*
| *SOrL* (′*n*, ′*d*) *sproof*
| *SOrR* (′*n*, ′*d*) *sproof*
| *SAnd* (′*n*, ′*d*) *sproof* (′*n*, ′*d*) *sproof*
| *SImpL* (′*n*, ′*d*) *vproof*
| *SImpR* (′*n*, ′*d*) *sproof*
| *SIffSS* (′*n*, ′*d*) *sproof* (′*n*, ′*d*) *sproof*
| *SIffVV* (′*n*, ′*d*) *vproof* (′*n*, ′*d*) *vproof*
| *SExists* ′*n* ′*d* (′*n*, ′*d*) *sproof*
| *SForall* ′*n* (′*d*, (′*n*, ′*d*) *sproof*) *part*
| *SPrev* (′*n*, ′*d*) *sproof*
| *SNext* (′*n*, ′*d*) *sproof*
| *SOnce nat* (′*n*, ′*d*) *sproof*
| *SEventually nat* (′*n*, ′*d*) *sproof*
| *SHistorically nat nat* (′*n*, ′*d*) *sproof list*
| *SHistoricallyOut nat*
| *SAlways nat nat* (′*n*, ′*d*) *sproof list*
| *SSince* (′*n*, ′*d*) *sproof* (′*n*, ′*d*) *sproof list*
| *SUntil* (′*n*, ′*d*) *sproof list* (′*n*, ′*d*) *sproof*

```
  | SMatchP ('n, 'd) sproof Regex_Proof_Object.rsproof
  | SMatchF ('n, 'd) sproof Regex_Proof_Object.rsproof
  and ('n, 'd) vproof = VFF nat
  | VPred nat 'n ('n, 'd) Formula.trm list
  | VEq_Const nat 'n 'd
  | VNeg ('n, 'd) sproof
  | VOr ('n, 'd) vproof ('n, 'd) vproof
  | VAndL ('n, 'd) vproof
  | VAndR ('n, 'd) vproof
  | VImp ('n, 'd) sproof ('n, 'd) vproof
  | VIffSV ('n, 'd) sproof ('n, 'd) vproof
  | VIffVS ('n, 'd) vproof ('n, 'd) sproof
  | VExists 'n ('d, ('n, 'd) vproof) part
  | VForall 'n 'd ('n, 'd) vproof
  | VPrev ('n, 'd) vproof
  | VPrevZ
  | VPrevOutL nat
  | VPrevOutR nat
  | VNext ('n, 'd) vproof
  | VNextOutL nat
  | VNextOutR nat
  | VOnceOut nat
  | VOnce nat nat ('n, 'd) vproof list
  | VEventually nat nat ('n, 'd) vproof list
  | VHistorically nat ('n, 'd) vproof
  | VAlways nat ('n, 'd) vproof
  | VSinceOut nat
  | VSince nat ('n, 'd) vproof ('n, 'd) vproof list
  | VSinceInf nat nat ('n, 'd) vproof list
  | VUntil nat ('n, 'd) vproof list ('n, 'd) vproof
  | VUntilInf nat nat ('n, 'd) vproof list
  | VMatchPOut nat
  | VMatchP nat ('n, 'd) vproof Regex_Proof_Object.rvproof list
  | VMatchF nat ('n, 'd) vproof Regex_Proof_Object.rvproof list

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

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

fun s_at :: ('n, 'd) sproof ⇒ nat and
  v_at :: ('n, 'd) vproof ⇒ nat where
  s_at (STT i) = i
| s_at (SPred i _ _) = i
| s_at (SEq_Const i _ _) = i
| s_at (SNeg vp) = v_at vp
| s_at (SOrL sp1) = s_at sp1
| s_at (SOrR sp2) = s_at sp2
| s_at (SAnd sp1 _) = s_at sp1
| s_at (SImpL vp1) = v_at vp1
| s_at (SImpR sp2) = s_at sp2
| s_at (SIffSS sp1 _) = s_at sp1
| s_at (SIffVV vp1 _) = v_at vp1
| s_at (SExists _ _ sp) = s_at sp
| s_at (SForall _ part) = s_at (part_hd part)
| s_at (SPrev sp) = s_at sp + 1
| s_at (SNext sp) = s_at sp - 1
| s_at (SOnce i _) = i
| s_at (SEventually i _) = i
```

| s_at (*SHistorically i _ _*) = *i*
| s_at (*SHistoricallyOut i*) = *i*
| s_at (*SAlways i _ _*) = *i*
| s_at (*SSince sp2 sp1s*) = (*case sp1s of* [] ⇒ *s_at sp2* | _ ⇒ *s_at* (*last sp1s*))
| s_at (*SUntil sp1s sp2*) = (*case sp1s of* [] ⇒ *s_at sp2* | *sp1* # _ ⇒ *s_at sp1*)
| s_at (*SMatchP rsp*) = (*snd* (*rs_at s_at rsp*))
| s_at (*SMatchF rsp*) = (*fst* (*rs_at s_at rsp*))
| v_at (*VFF i*) = *i*
| v_at (*VPred i _ _*) = *i*
| v_at (*VEq_Const i _ _*) = *i*
| v_at (*VNeg sp*) = *s_at sp*
| v_at (*VOr vp1 _*) = *v_at vp1*
| v_at (*VAndL vp1*) = *v_at vp1*
| v_at (*VAndR vp2*) = *v_at vp2*
| v_at (*VImp sp1 _*) = *s_at sp1*
| v_at (*VIffSV sp1 _*) = *s_at sp1*
| v_at (*VIffVS vp1 _*) = *v_at vp1*
| v_at (*VExists _ part*) = *v_at* (*part_hd part*)
| v_at (*VForall _ _ vp1*) = *v_at vp1*
| v_at (*VPrev vp*) = *v_at vp* + *1*
| v_at (*VPrevZ*) = *0*
| v_at (*VPrevOutL i*) = *i*
| v_at (*VPrevOutR i*) = *i*
| v_at (*VNext vp*) = *v_at vp* − *1*
| v_at (*VNextOutL i*) = *i*
| v_at (*VNextOutR i*) = *i*
| v_at (*VOnceOut i*) = *i*
| v_at (*VOnce i _ _*) = *i*
| v_at (*VEventually i _ _*) = *i*
| v_at (*VHistorically i _*) = *i*
| v_at (*VAlways i _*) = *i*
| v_at (*VSinceOut i*) = *i*
| v_at (*VSince i _ _*) = *i*
| v_at (*VSinceInf i _ _*) = *i*
| v_at (*VUntil i _ _*) = *i*
| v_at (*VUntilInf i _ _*) = *i*
| v_at (*VMatchPOut i*) = *i*
| v_at (*VMatchP i _*) = *i*
| v_at (*VMatchF i _*) = *i*

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

8 Auxiliary Lemmas

lemma *Cons_eq_upt_conv*: *x* # *xs* = [*m* ..< *n*] ⟷ *m* < *n* ∧ *x* = *m* ∧ *xs* = [*Suc m* ..< *n*]
⟨*proof*⟩

lemma *map_setE*[*elim_format*]: *map f xs* = *ys* ⟹ *y* ∈ *set ys* ⟹ ∃ *x*∈*set xs*. *f x* = *y*
⟨*proof*⟩

lemma *set_Cons_eq*: *set_Cons X XS* = (⋃ *xs*∈*XS*. (λ*x*. *x* # *xs*) ‘ *X*)
⟨*proof*⟩

lemma *set_Cons_empty_iff*: *set_Cons X XS* = {} ⟷ (*X* = {} ∨ *XS* = {})
⟨*proof*⟩

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)
⟨*proof*⟩

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

lemma *fst_pos_None_iff*: *fst_pos xs x* = *None* ⟷ $x \notin$ *set xs*
⟨*proof*⟩

lemma *nth_fst_pos*: $x \in$ *set xs* ⟹ *xs* ! (*the* (*fst_pos xs x*)) = x
⟨*proof*⟩

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

lemma *eq_positions_iff*: *length xs* = *length ys*
⟹ *positions xs x* = *positions ys y* ⟷ (∀ *n*< *length xs*. *xs* ! *n* = *x* ⟷ *ys* ! *n* = *y*)
⟨*proof*⟩

lemma *positions_eq_nil_iff*: *positions xs x* = [] ⟷ $x \notin$ *set xs*
⟨*proof*⟩

lemma *positions_nth*: $n \in$ *set* (*positions xs x*) ⟹ *xs* ! *n* = *x*
⟨*proof*⟩

lemma *set_positions_eq*: *set* (*positions xs x*) = {*n*. *xs* ! *n* = *x* ∧ *n* < *length xs*}
⟨*proof*⟩

lemma *positions_length*: $n \in$ *set* (*positions xs x*) ⟹ *n* < *length xs*
⟨*proof*⟩

lemma *positions_nth_cong*:
$m \in$ *set* (*positions xs x*) ⟹ $n \in$ *set* (*positions xs x*) ⟹ *xs* ! *n* = *xs* ! *m*
⟨*proof*⟩

lemma *fst_pos_in_positions*: $x \in$ *set xs* ⟹ *the* (*fst_pos xs x*) ∈ *set* (*positions xs x*)
⟨*proof*⟩

lemma *hd_positions_eq_fst_pos*: $x \in$ *set xs* ⟹ *hd* (*positions xs x*) = *the* (*fst_pos xs x*)
⟨*proof*⟩

lemma *sorted_positions*: *sorted* (*positions xs x*)
⟨*proof*⟩

lemma *Min_sorted_list*: *sorted xs* ⟹ *xs* ≠ [] ⟹ *Min* (*set xs*) = *hd xs*
⟨*proof*⟩

lemma *Min_positions*: $x \in$ *set xs* ⟹ *Min* (*set* (*positions xs x*)) = *the* (*fst_pos xs x*)
⟨*proof*⟩

lemma *subset_positions_map_fst*: *set* (*positions tXs tX*) ⊆ *set* (*positions* (*map fst tXs*) (*fst tX*))
⟨*proof*⟩

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

⟨proof⟩

lemma *Max_eqI*: *finite A* ⟹ *A* ≠ {} ⟹ (⋀*a*. *a* ∈ *A* ⟹ *a* ≤ *b*) ⟹ ∃ *a*∈*A*. *b* ≤ *a* ⟹ *Max A* = *b*
⟨proof⟩

lemma *Max_Suc*: *X* ≠ {} ⟹ *finite X* ⟹ *Max* (*Suc* ‘ *X*) = *Suc* (*Max X*)
⟨proof⟩

lemma *Max_insert0*: *X* ≠ {} ⟹ *finite X* ⟹ *Max* (*insert* (*0*::*nat*) *X*) = *Max X*
⟨proof⟩

lemma *positions_Cons_notin_tail*: *x* ∉ *set xs* ⟹ *positions* (*x* # *xs*) *x* = [*0*::*nat*]
⟨proof⟩

lemma *Max_set_positions_Cons_hd*:
x ∉ *set xs* ⟹ *Max* (*set* (*positions* (*x* # *xs*) *x*)) = *0*
⟨proof⟩

lemma *Max_set_positions_Cons_tl*:
y ∈ *set xs* ⟹ *Max* (*set* (*positions* (*x* # *xs*) *y*)) = *Suc* (*Max* (*set* (*positions xs y*)))
⟨proof⟩

lemma *max_aux*: *finite X* ⟹ *Suc j* ∈ *X* ⟹ *Max* (*insert* (*Suc j*) (*X* − {*j*})) = *Max* (*insert j X*)
⟨proof⟩

lemma *ball_swap*: (∀ *x* ∈ *A*. ∀ *y* ∈ *B*. *P x y*) = (∀ *y* ∈ *B*. ∀ *x* ∈ *A*. *P x y*)
⟨proof⟩

lemma *ball_triv_nonempty*: *A* ≠ {} ⟹ (∀ *x* ∈ *A*. *P*) = *P*
⟨proof⟩

lemma *ball_if_distrib*: (∀ *x* ∈ *B*. *if p then f x else g x*) ⟷ (*if p then* (∀ *x* ∈ *B*. *f x*) *else* (∀ *x* ∈ *B*. *g x*))
⟨proof⟩

context fixes *test* :: ′*a* ⇒ ′*b* ⇒ *bool* **and** *testi* :: ′*b* ⇒ *nat* **begin**
fun *rs_check* **where**
rs_check (*Regex.Skip n*) (*SSkip x y*) = ((*snd* (*rs_at testi* (*SSkip x y*)) = *x* + *n*))
| *rs_check* (*Regex.Test x*) (*STest y*) = *test x y*
| *rs_check* (*Regex.Plus r r*′) (*SPlusL z*) = *rs_check r z*
| *rs_check* (*Regex.Plus r r*′) (*SPlusR z*) = *rs_check r*′ *z*
| *rs_check* (*Regex.Times r r*′) (*STimes p1 p2*) =
(*snd* (*rs_at testi p1*) = *fst* (*rs_at testi p2*) ∧ *rs_check r p1* ∧ *rs_check r*′ *p2*)
| *rs_check* (*Regex.Star r*) (*SStar_eps n*) = *True*
| *rs_check* (*Regex.Star r*) (*SStar ps*) = (*ps* ≠ [] ∧
(∀ *k* ∈ {*1* ..< *length ps*}. *fst* (*rs_at testi* (*ps* ! *k*)) = *snd* (*rs_at testi* (*ps* ! (*k*−*1*)))) ∧
(∀ *k* ∈ {*0* ..< *length ps*}. *fst* (*rs_at testi* (*ps* ! *k*)) < *snd* (*rs_at testi* (*ps* ! *k*)) ∧ *rs_check r* (*ps* ! *k*)))
| *rs_check* _ _ = *False*
end

lemma *rs_check_cong*[*fundef_cong*]:
p = *p*′ ⟹ (⋀*x sp*. *x* ∈ *regex.atms r* ⟹ *sp* ∈ *spatms p* ⟹ *t x sp* = *t*′ *x sp*)
⟹ (⋀*x*. *x* ∈ *spatms p* ⟹ *ti x* = *ti*′ *x*) ⟹ *rs_check t ti r p* = *rs_check t*′ *ti*′ *r p*′
⟨proof⟩

context fixes *test* :: ′*a* ⇒ ′*b* ⇒ *bool* **and** *testi* :: ′*b* ⇒ *nat* **begin**
fun *rv_check* **where**
rv_check (*Regex.Skip n*) (*VSkip i j*) = (*i* ≤ *j* ∧ *j* ≠ *i* + *n*)
| *rv_check* (*Regex.Test x*) (*VTest p*) = *test x p*

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

lemma $rv\_check\_code\_Times$:
rv_check $(Regex.Times\ r\ r')$ $(VTimes\ ps) = (ps \neq [] \wedge$
$(let\ i = fst\ (rv\_at\ testi\ (snd\ (hd\ ps)));\ j = snd\ (rv\_at\ testi\ (snd\ (last\ ps)))\ in$
$i + length\ ps - 1 = j \wedge (\forall\, k \in \{0\ ..< length\ ps\}.\ let\ (b,\ p) = ps\ !\ k\ in$
$if\ b\ then\ rv\_check\ r\ p \wedge rv\_at\ testi\ p = (i,\ i + k)$
$else\ rv\_check\ r'\ p \wedge rv\_at\ testi\ p = (i + k,\ j))))$
$\langle proof \rangle$
lemma $rv\_check\_code\_Star$:
rv_check $(Regex.Star\ r)$ $(VStar\ ps) =$
$(let\ S = set\ (map\ (fst \circ rv\_at\ testi)\ ps);\ T = set\ (map\ (snd \circ rv\_at\ testi)\ ps);$
$i = Min\ S;\ j = Max\ T\ in\ i \leq j \wedge S \cap T = \{\} \wedge S \cup T = \{i\ ..\ j\}$
$\wedge\ map\ (rv\_at\ testi)\ ps = sorted\_list\_of\_set\ (rm\ (S \times T))$
$\wedge\ (\forall\, k \in \{0\ ..< length\ ps\}.\ rv\_check\ r\ (ps\ !\ k)))$
$\langle proof \rangle$

declare $rv\_check.simps[code\ del]$
lemmas $rv\_check\_code[code] = rv\_check.simps(1{-}4)\ rv\_check\_code\_Times\ rv\_check\_code\_Star\ rv\_check.simps(7{-})$
end

lemma $rv\_check\_cong[fundef\_cong]$:
$p = p' \Longrightarrow (\bigwedge x\ vp.\ x \in regex.atms\ r \wedge vp \in vpatms\ p \Longrightarrow t\ x\ vp = t'\ x\ vp)$
$\Longrightarrow (\bigwedge x.\ x \in vpatms\ p \Longrightarrow ti\ x = ti'\ x) \Longrightarrow rv\_check\ t\ ti\ r\ p = rv\_check\ t'\ ti'\ r\ p'$
$\langle proof \rangle$

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 rs_check_sound:
$\forall\, x \in Regex.atms\ r.\ \forall\, p' \in spatms\ p.\ test\ x\ p' \longrightarrow sat\ (testi\ p')\ x \Longrightarrow$
$rs\_check\ test\ testi\ r\ p \Longrightarrow Regex\_Proof\_System.SAT\ sat\ (fst\ (rs\_at\ testi\ p))\ (snd\ (rs\_at\ testi\ p))\ r$
$\langle proof \rangle$

lemma $rs\_check\_complete$:
$(\forall\, x \in Regex.atms\ r.\ \forall\, i.\ sat\ i\ x \longrightarrow (\exists\, p'.\ testi\ p' = i \wedge test\ x\ p')) \Longrightarrow$
$Regex\_Proof\_System.SAT\ sat\ i\ j\ r \Longrightarrow \exists\, p.\ rs\_check\ test\ testi\ r\ p \wedge rs\_at\ testi\ p = (i,\ j)$
$\langle proof \rangle$

lemma rv_check_sound:
$\forall\, x \in Regex.atms\ r.\ \forall\, p' \in vpatms\ p.\ test\ x\ p' \longrightarrow vio\ (testi\ p')\ x \Longrightarrow$

$rv\_check\ test\ testi\ r\ p \Longrightarrow Regex\_Proof\_System.VIO\ vio\ (fst\ (rv\_at\ testi\ p))\ (snd\ (rv\_at\ testi\ p))\ r$
⟨proof⟩

lemma *rv_check_complete*:
$(\forall x \in Regex.atms\ r.\ \forall i.\ vio\ i\ x \longrightarrow (\exists p'.\ testi\ p' = i \wedge test\ x\ p')) \Longrightarrow$
$Regex\_Proof\_System.VIO\ vio\ i\ j\ r \Longrightarrow i \leq j \Longrightarrow \exists p.\ rv\_check\ test\ testi\ r\ p \wedge rv\_at\ testi\ p = (i, j)$
⟨proof⟩

lemma *rs_check_exec_rs_check*:
fixes $test :: 'a \Rightarrow 'b \Rightarrow bool$
and $testi :: 'b \Rightarrow nat$
and $test' :: ('n \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
and $FV :: 'a \Rightarrow 'n\ set$
and $C :: 'n\ set \Rightarrow ('n \Rightarrow 'd)\ set$
assumes $C\_nonemptyI$: $\bigwedge A.\ C\ A \neq \{\}$
and C_union_eq: $\bigwedge X\ Y.\ C\ (X \cup Y) = C\ X \cap C\ Y$
and C_Union_eq: $\bigwedge X\ (Y :: 'a \Rightarrow \_).\ C\ (\bigcup\ (Y\ `\ X)) = (\bigcap x{\in}X.\ C\ (Y\ x))$
and $C\_extensible$: $\bigwedge X\ Y\ v.\ v \in C\ X \Longrightarrow X \subseteq Y \Longrightarrow \exists v'.\ v' \in C\ Y \wedge (\forall x{\in}X.\ v\ x = v'\ x)$
and $cong$: $\bigwedge v\ v'\ x\ sp.\ \forall a{\in}FV\ x.\ v\ a = v'\ a \Longrightarrow test'\ v\ x\ sp = test'\ v'\ x\ sp$
shows $(\bigwedge x\ sp.\ x \in regex.atms\ r \Longrightarrow test\ x\ sp = (\forall v{\in}C\ (FV\ x).\ test'\ v\ x\ sp)) \Longrightarrow$
$rs\_check\ test\ testi\ r\ rsp = (\forall v{\in}\bigcap x{\in}regex.atms\ r.\ C\ (FV\ x).\ rs\_check\ (test'\ v)\ testi\ r\ rsp)$
⟨proof⟩

lemma *rv_check_exec_rv_check*:
fixes $test :: 'a \Rightarrow 'b \Rightarrow bool$
and $testi :: 'b \Rightarrow nat$
and $test' :: ('n \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
and $FV :: 'a \Rightarrow 'n\ set$
and $C :: 'n\ set \Rightarrow ('n \Rightarrow 'd)\ set$
assumes $C\_nonemptyI$: $\bigwedge A.\ C\ A \neq \{\}$
and C_union_eq: $\bigwedge X\ Y.\ C\ (X \cup Y) = C\ X \cap C\ Y$
and C_Union_eq: $\bigwedge X\ (Y :: 'a \Rightarrow \_).\ C\ (\bigcup\ (Y\ `\ X)) = (\bigcap x{\in}X.\ C\ (Y\ x))$
and $C\_extensible$: $\bigwedge X\ Y\ v.\ v \in C\ X \Longrightarrow X \subseteq Y \Longrightarrow \exists v'.\ v' \in C\ Y \wedge (\forall x{\in}X.\ v\ x = v'\ x)$
and $cong$: $\bigwedge v\ v'\ x\ sp.\ \forall a{\in}FV\ x.\ v\ a = v'\ a \Longrightarrow test'\ v\ x\ sp = test'\ v'\ x\ sp$
shows $(\bigwedge x\ sp.\ x \in regex.atms\ r \Longrightarrow test\ x\ sp = (\forall v{\in}C\ (FV\ x).\ test'\ v\ x\ sp)) \Longrightarrow$
$rv\_check\ test\ testi\ r\ rsp = (\forall v{\in}\bigcap x{\in}regex.atms\ r.\ C\ (FV\ x).\ rv\_check\ (test'\ v)\ testi\ r\ rsp)$
⟨proof⟩

lemma *chain_sorted1*:
fixes $f :: \_ \Rightarrow nat \times nat$
assumes $\forall k{\in}\{Suc\ 0..{<}length\ ps\}.\ fst\ (f\ (ps\ !\ k)) = snd\ (f\ (ps\ !\ (k - Suc\ 0)))$
and $\forall k{\in}\{0..{<}length\ ps\}.\ fst\ (f\ (ps\ !\ k)) < snd\ (f\ (ps\ !\ k))$
and $j \leq k\ k < length\ ps$
shows $fst\ (f\ (ps\ !\ j)) \leq fst\ (f\ (ps\ !\ k))$
⟨proof⟩

lemma *chain_sorted2*:
fixes $f :: \_ \Rightarrow nat \times nat$
assumes $\forall k{\in}\{Suc\ 0..{<}length\ ps\}.\ fst\ (f\ (ps\ !\ k)) = snd\ (f\ (ps\ !\ (k - Suc\ 0)))$
and $\forall k{\in}\{0..{<}length\ ps\}.\ fst\ (f\ (ps\ !\ k)) < snd\ (f\ (ps\ !\ k))$
and $j \leq k\ k < length\ ps$
shows $snd\ (f\ (ps\ !\ j)) \leq snd\ (f\ (ps\ !\ k))$
⟨proof⟩

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

begin

lemma *rs_check_le*:
$rs\_check\ test\ testi\ r\ rsp \Longrightarrow fst\ (rs\_at\ testi\ rsp) \leq snd\ (rs\_at\ testi\ rsp)$
⟨*proof*⟩

lemma *rs_check_le1*:
$rs\_check\ test\ testi\ r\ rsp \Longrightarrow sp \in spatms\ rsp \Longrightarrow fst\ (rs\_at\ testi\ rsp) \leq testi\ sp$
⟨*proof*⟩

lemma *rs_check_le2*:
$rs\_check\ test\ testi\ r\ rsp \Longrightarrow sp \in spatms\ rsp \Longrightarrow testi\ sp \leq snd\ (rs\_at\ testi\ rsp)$
⟨*proof*⟩

end

lemma *rv_check_le*:
$rv\_check\ test\ testi\ r\ rvp \Longrightarrow vp \in vpatms\ rvp \Longrightarrow fst\ (rv\_at\ testi\ rvp) \leq snd\ (rv\_at\ testi\ rvp)$
⟨*proof*⟩

lemma *rv_check_le2*:
$rv\_check\ test\ testi\ r\ rvp \Longrightarrow vp \in vpatms\ rvp \Longrightarrow testi\ vp \leq snd\ (rv\_at\ testi\ rvp)$
⟨*proof*⟩

9 Proof Checker

unbundle *MFOTL_syntax*

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

begin

fun $s\_check :: ('n,\ 'd)\ env \Rightarrow ('n,\ 'd)\ formula \Rightarrow ('n,\ 'd)\ sproof \Rightarrow bool$
and $v\_check :: ('n,\ 'd)\ env \Rightarrow ('n,\ 'd)\ formula \Rightarrow ('n,\ 'd)\ vproof \Rightarrow bool$ **where**
$s\_check\ v\ f\ p = (case\ (f,\ p)\ of$
$(\top,\ STT\ i) \Rightarrow True$
$|\ (r \dagger ts,\ SPred\ i\ s\ ts') \Rightarrow$
$(r = s \wedge ts = ts' \wedge (r,\ v[\![ts]\!]) \in \Gamma\ \sigma\ i)$
$|\ (x \approx c,\ SEq\_Const\ i\ x'\ c') \Rightarrow$
$(c = c' \wedge x = x' \wedge v\ x = c)$
$|\ (\neg_F\ \varphi,\ SNeg\ vp) \Rightarrow v\_check\ v\ \varphi\ vp$
$|\ (\varphi \vee_F \psi,\ SOrL\ sp1) \Rightarrow s\_check\ v\ \varphi\ sp1$
$|\ (\varphi \vee_F \psi,\ SOrR\ sp2) \Rightarrow s\_check\ v\ \psi\ sp2$
$|\ (\varphi \wedge_F \psi,\ SAnd\ sp1\ sp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge s\_check\ v\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2$
$|\ (\varphi \longrightarrow_F \psi,\ SImpL\ vp1) \Rightarrow v\_check\ v\ \varphi\ vp1$
$|\ (\varphi \longrightarrow_F \psi,\ SImpR\ sp2) \Rightarrow s\_check\ v\ \psi\ sp2$
$|\ (\varphi \longleftrightarrow_F \psi,\ SIffSS\ sp1\ sp2) \Rightarrow s\_check\ v\ \varphi\ sp1 \wedge s\_check\ v\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2$
$|\ (\varphi \longleftrightarrow_F \psi,\ SIffVV\ vp1\ vp2) \Rightarrow v\_check\ v\ \varphi\ vp1 \wedge v\_check\ v\ \psi\ vp2 \wedge v\_at\ vp1 = v\_at\ vp2$
$|\ (\exists_F x.\ \ \varphi,\ SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s\_check\ (v\ (x := val))\ \varphi\ sp)$
$|\ (\forall_F x.\ \ \varphi,\ SForall\ y\ sp\_part) \Rightarrow (let\ i = s\_at\ (part\_hd\ sp\_part)$
$in\ x = y \wedge (\forall (sub,\ sp) \in SubsVals\ sp\_part.\ s\_at\ sp = i \wedge (\forall z \in sub.\ s\_check\ (v\ (x := z))\ \varphi\ sp)))$
$|\ (\mathbf{Y}\ I\ \varphi,\ SPrev\ sp) \Rightarrow$
$(let\ j = s\_at\ sp;\ i = s\_at\ (SPrev\ sp)\ in$
$i = j{+}1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s\_check\ v\ \varphi\ sp)$
$|\ (\mathbf{X}\ I\ \varphi,\ SNext\ sp) \Rightarrow$
$(let\ j = s\_at\ sp;\ i = s\_at\ (SNext\ sp)\ in$
$j = i{+}1 \wedge mem\ (\Delta\ \sigma\ j)\ I \wedge s\_check\ v\ \varphi\ sp)$
$|\ (\mathbf{P}\ I\ \varphi,\ SOnce\ i\ sp) \Rightarrow$

$(\mathit{let}\ j = \mathit{s\_at}\ \mathit{sp}\ \mathit{in}$
$j \leq i \wedge \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \mathit{s\_check}\ v\ \varphi\ \mathit{sp})$
$\mid (\mathbf{F}\ I\ \varphi,\ \mathit{SEventually}\ i\ \mathit{sp}) \Rightarrow$
$(\mathit{let}\ j = \mathit{s\_at}\ \mathit{sp}\ \mathit{in}$
$j \geq i \wedge \mathit{mem}\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge \mathit{s\_check}\ v\ \varphi\ \mathit{sp})$
$\mid (\mathbf{H}\ I\ \varphi,\ \mathit{SHistoricallyOut}\ i) \Rightarrow$
$\tau\ \sigma\ i < \tau\ \sigma\ 0 + \mathit{left}\ I$
$\mid (\mathbf{H}\ I\ \varphi,\ \mathit{SHistorically}\ i\ \mathit{li}\ \mathit{sps}) \Rightarrow$
$(\mathit{li} = (\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow 0 \mid \mathit{enat}\ b \Rightarrow \mathit{ETP}\ \sigma\ (\tau\ \sigma\ i - b))$
$\wedge\ \tau\ \sigma\ 0 + \mathit{left}\ I \leq \tau\ \sigma\ i$
$\wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sps} = [\mathit{li}\ ..{<}\ (\mathit{LTP\_p}\ \sigma\ i\ I) + 1]$
$\wedge\ (\forall\, \mathit{sp} \in \mathit{set}\ \mathit{sps}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp}))$
$\mid (\mathbf{G}\ I\ \varphi,\ \mathit{SAlways}\ i\ \mathit{hi}\ \mathit{sps}) \Rightarrow$
$(\mathit{hi} = (\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \mathit{enat}\ b \Rightarrow \mathit{LTP\_f}\ \sigma\ i\ b)$
$\wedge\ \mathit{right}\ I \neq \infty$
$\wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sps} = [(\mathit{ETP\_f}\ \sigma\ i\ I)\ ..{<}\ \mathit{hi} + 1]$
$\wedge\ (\forall\, \mathit{sp} \in \mathit{set}\ \mathit{sps}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp}))$
$\mid (\varphi\ \mathbf{S}\ I\ \psi,\ \mathit{SSince}\ \mathit{sp2}\ \mathit{sp1s}) \Rightarrow$
$(\mathit{let}\ i = \mathit{s\_at}\ (\mathit{SSince}\ \mathit{sp2}\ \mathit{sp1s});\ j = \mathit{s\_at}\ \mathit{sp2}\ \mathit{in}$
$j \leq i \wedge \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I$
$\wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sp1s} = [j{+}1\ ..{<}\ i{+}1]$
$\wedge\ \mathit{s\_check}\ v\ \psi\ \mathit{sp2}$
$\wedge\ (\forall\, \mathit{sp1} \in \mathit{set}\ \mathit{sp1s}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp1}))$
$\mid (\varphi\ \mathbf{U}\ I\ \psi,\ \mathit{SUntil}\ \mathit{sp1s}\ \mathit{sp2}) \Rightarrow$
$(\mathit{let}\ i = \mathit{s\_at}\ (\mathit{SUntil}\ \mathit{sp1s}\ \mathit{sp2});\ j = \mathit{s\_at}\ \mathit{sp2}\ \mathit{in}$
$j \geq i \wedge \mathit{mem}\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I$
$\wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sp1s} = [i\ ..{<}\ j] \wedge \mathit{s\_check}\ v\ \psi\ \mathit{sp2}$
$\wedge\ (\forall\, \mathit{sp1} \in \mathit{set}\ \mathit{sp1s}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp1}))$
$\mid (\triangleleft\ I\ r,\ \mathit{SMatchP}\ \mathit{rsp}) \Rightarrow$
$(\mathit{let}\ (j,\ i) = \mathit{rs\_at}\ \mathit{s\_at}\ \mathit{rsp}\ \mathit{in}\ j \leq i \wedge \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \mathit{rs\_check}\ (\mathit{s\_check}\ v)\ \mathit{s\_at}\ r\ \mathit{rsp})$
$\mid (\triangleright\ I\ r,\ \mathit{SMatchF}\ \mathit{rsp}) \Rightarrow$
$(\mathit{let}\ (i,\ j) = \mathit{rs\_at}\ \mathit{s\_at}\ \mathit{rsp}\ \mathit{in}\ i \leq j \wedge \mathit{mem}\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge \mathit{rs\_check}\ (\mathit{s\_check}\ v)\ \mathit{s\_at}\ r\ \mathit{rsp})$
$\mid (\_\ ,\ \_) \Rightarrow \mathit{False})$
$\mid \mathit{v\_check}\ v\ f\ p = (\mathit{case}\ (f,\ p)\ \mathit{of}$
$(\bot,\ \mathit{VFF}\ i) \Rightarrow \mathit{True}$
$\mid (r \dagger \mathit{ts},\ \mathit{VPred}\ i\ \mathit{pred}\ \mathit{ts}') \Rightarrow$
$(r = \mathit{pred} \wedge \mathit{ts} = \mathit{ts}' \wedge (r,\ v[\![\mathit{ts}]\!]) \notin \Gamma\ \sigma\ i)$
$\mid (x \approx c,\ \mathit{VEq\_Const}\ i\ x'\ c') \Rightarrow$
$(c = c' \wedge x = x' \wedge v\ x \neq c)$
$\mid (\neg_F\ \varphi,\ \mathit{VNeg}\ \mathit{sp}) \Rightarrow \mathit{s\_check}\ v\ \varphi\ \mathit{sp}$
$\mid (\varphi \vee_F \psi,\ \mathit{VOr}\ \mathit{vp1}\ \mathit{vp2}) \Rightarrow \mathit{v\_check}\ v\ \varphi\ \mathit{vp1} \wedge \mathit{v\_check}\ v\ \psi\ \mathit{vp2} \wedge \mathit{v\_at}\ \mathit{vp1} = \mathit{v\_at}\ \mathit{vp2}$
$\mid (\varphi \wedge_F \psi,\ \mathit{VAndL}\ \mathit{vp1}) \Rightarrow \mathit{v\_check}\ v\ \varphi\ \mathit{vp1}$
$\mid (\varphi \wedge_F \psi,\ \mathit{VAndR}\ \mathit{vp2}) \Rightarrow \mathit{v\_check}\ v\ \psi\ \mathit{vp2}$
$\mid (\varphi \longrightarrow_F \psi,\ \mathit{VImp}\ \mathit{sp1}\ \mathit{vp2}) \Rightarrow \mathit{s\_check}\ v\ \varphi\ \mathit{sp1} \wedge \mathit{v\_check}\ v\ \psi\ \mathit{vp2} \wedge \mathit{s\_at}\ \mathit{sp1} = \mathit{v\_at}\ \mathit{vp2}$
$\mid (\varphi \longleftrightarrow_F \psi,\ \mathit{VIffSV}\ \mathit{sp1}\ \mathit{vp2}) \Rightarrow \mathit{s\_check}\ v\ \varphi\ \mathit{sp1} \wedge \mathit{v\_check}\ v\ \psi\ \mathit{vp2} \wedge \mathit{s\_at}\ \mathit{sp1} = \mathit{v\_at}\ \mathit{vp2}$
$\mid (\varphi \longleftrightarrow_F \psi,\ \mathit{VIffVS}\ \mathit{vp1}\ \mathit{sp2}) \Rightarrow \mathit{v\_check}\ v\ \varphi\ \mathit{vp1} \wedge \mathit{s\_check}\ v\ \psi\ \mathit{sp2} \wedge \mathit{v\_at}\ \mathit{vp1} = \mathit{s\_at}\ \mathit{sp2}$
$\mid (\exists_F x.\ \ \varphi,\ \mathit{VExists}\ y\ \mathit{vp\_part}) \Rightarrow (\mathit{let}\ i = \mathit{v\_at}\ (\mathit{part\_hd}\ \mathit{vp\_part})$
$\mathit{in}\ x = y \wedge (\forall\, (\mathit{sub},\ \mathit{vp}) \in \mathit{SubsVals}\ \mathit{vp\_part}.\ \mathit{v\_at}\ \mathit{vp} = i \wedge (\forall\, z \in \mathit{sub}.\ \mathit{v\_check}\ (v\ (x := z))\ \varphi\ \mathit{vp})))$
$\mid (\forall_F x.\ \ \varphi,\ \mathit{VForall}\ y\ \mathit{val}\ \mathit{vp}) \Rightarrow (x = y \wedge \mathit{v\_check}\ (v\ (x := \mathit{val}))\ \varphi\ \mathit{vp})$
$\mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrev}\ \mathit{vp}) \Rightarrow$
$(\mathit{let}\ j = \mathit{v\_at}\ \mathit{vp};\ i = \mathit{v\_at}\ (\mathit{VPrev}\ \mathit{vp})\ \mathit{in}$
$i = j{+}1 \wedge \mathit{v\_check}\ v\ \varphi\ \mathit{vp})$
$\mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevZ}) \Rightarrow \mathit{True}$
$\mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevOutL}\ i) \Rightarrow$
$i > 0 \wedge \Delta\ \sigma\ i < \mathit{left}\ I$
$\mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevOutR}\ i) \Rightarrow$
$i > 0 \wedge \mathit{enat}\ (\Delta\ \sigma\ i) > \mathit{right}\ I$
$\mid (\mathbf{X}\ I\ \varphi,\ \mathit{VNext}\ \mathit{vp}) \Rightarrow$

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

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

simps_of_case *v_check_simps*[*simp*]: *v_check.simps*[*unfolded prod.case*] (*splits*: *formula.split vproof.split*)

9.1 Checker Soundness

lemma *check_soundness*:
$s\_check\ v\ \varphi\ sp \Longrightarrow SAT\ \sigma\ v\ (s\_at\ sp)\ \varphi$
$v\_check\ v\ \varphi\ vp \Longrightarrow VIO\ \sigma\ v\ (v\_at\ vp)\ \varphi$
⟨*proof*⟩

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

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

lemma *compatible_alt*:
$compatible\ X\ vs\ v \longleftrightarrow v \in compatible\_vals\ X\ vs$
⟨*proof*⟩

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

lemma *compatible_vals_empty_eq*: $compatible\_vals\ \{\}\ vs = UNIV$
⟨*proof*⟩

lemma *compatible_union_iff*:
$compatible\ (X \cup Y)\ vs\ v \longleftrightarrow compatible\ X\ vs\ v \wedge compatible\ Y\ vs\ v$
⟨*proof*⟩

lemma *compatible_vals_union_eq*:
$compatible\_vals\ (X \cup Y)\ vs = compatible\_vals\ X\ vs \cap compatible\_vals\ Y\ vs$
⟨*proof*⟩

lemma *compatible_vals_Union_eq*:
$compatible\_vals\ (\bigcup x{\in}X.\ Y\ x)\ vs = (\bigcap x \in X.\ compatible\_vals\ (Y\ x)\ vs)$
⟨*proof*⟩

lemma *compatible_antimono*:
$compatible\ X\ vs\ v \Longrightarrow Y \subseteq X \Longrightarrow compatible\ Y\ vs\ v$
⟨*proof*⟩

lemma *compatible_vals_antimono*:
$Y \subseteq X \Longrightarrow compatible\_vals\ X\ vs \subseteq compatible\_vals\ Y\ vs$
⟨*proof*⟩

lemma *compatible_extensible*:
$(\forall x.\ vs\ x \neq \{\}) \Longrightarrow compatible\ X\ vs\ v \Longrightarrow X \subseteq Y \Longrightarrow \exists v'.\ compatible\ Y\ vs\ v' \wedge (\forall x{\in}X.\ v\ x = v'\ x)$
⟨*proof*⟩

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

primrec $mk\_values :: (('n,\ 'd)\ trm \times 'a\ set)\ list \Rightarrow 'a\ list\ set$
where $mk\_values\ [] = \{[]\}$
| $mk\_values\ (T\ \#\ Ts) = (case\ T\ of$
$(\mathbf{v}\ x,\ X) \Rightarrow$
$let\ terms = map\ fst\ Ts\ in$
$if\ \mathbf{v}\ x \in set\ terms\ then$
$let\ fst\_pos = hd\ (positions\ terms\ (\mathbf{v}\ x))\ in\ (\lambda xs.\ (xs\ !\ fst\_pos)\ \#\ xs)\ `\ (mk\_values\ Ts)$
$else\ set\_Cons\ X\ (mk\_values\ Ts)$
| $(\mathbf{c}\ a,\ X) \Rightarrow set\_Cons\ X\ (mk\_values\ Ts))$

lemma *mk_values_nempty*:
$\{\} \notin$ *set* (*map snd tXs*) $\Longrightarrow$ *mk_values tXs* $\neq \{\}$
$\langle proof \rangle$

lemma *mk_values_not_Nil*:
$\{\} \notin$ *set* (*map snd tXs*) $\Longrightarrow$ *tXs* $\neq []$ $\Longrightarrow$ *vs* $\in$ *mk_values tXs* $\Longrightarrow$ *vs* $\neq []$
$\langle proof \rangle$

lemma *mk_values_nth_cong*: **v** *x* $\in$ *set* (*map fst tXs*) $\Longrightarrow$
n $\in$ *set* (*positions* (*map fst tXs*) (**v** *x*)) $\Longrightarrow$
m $\in$ *set* (*positions* (*map fst tXs*) (**v** *x*)) $\Longrightarrow$
vs $\in$ *mk_values tXs* $\Longrightarrow$
vs ! *n* = *vs* ! *m*
$\langle proof \rangle$

definition *mk_values_subset p tXs X*
$\longleftrightarrow$ (*let* (*fintXs*, *inftXs*) = *partition* (λ*tX*. *finite* (*snd tX*)) *tXs in*
if inftXs = [] *then* $\{p\} \times$ *mk_values tXs* $\subseteq X$
else let inf_dups = *filter* (λ*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''* (λ_. $\{p\}$ $\times$ *mk_values tXs* $\subseteq X$))
else if list_all (λ*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''* (λ_. $\{p\} \times$ *mk_values tXs* $\subseteq X$)))

lemma *mk_values_nemptyI*: $\forall$ *tX* $\in$ *set tXs*. *snd tX* $\neq \{\}$ $\Longrightarrow$ *mk_values tXs* $\neq \{\}$
$\langle proof \rangle$

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

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

lemma *mk_values_subset_iff*: $\forall$ *tX* $\in$ *set tXs*. *snd tX* $\neq \{\}$ $\Longrightarrow$
mk_values_subset p tXs X $\longleftrightarrow$ $\{p\} \times$ *mk_values tXs* $\subseteq X$
$\langle proof \rangle$

lemma *mk_values_sound*: *cs* $\in$ *mk_values* (*vs*$\{\!|$*ts*$|\!\}$) $\Longrightarrow$
$\exists$ *v*$\in$*compatible_vals* (*fv* (*r* $\dagger$ *ts*)) *vs*. *cs* = *v*$[\![$*ts*$]\!]$
$\langle proof \rangle$

lemma *fst_eval_trm_set*[*simp*]:
fst (*vs*$\{\!|$*t*$|\!\}$) = *t*
$\langle proof \rangle$

lemma *mk_values_complete*: *cs* = *v*$[\![$*ts*$]\!]$ $\Longrightarrow$
v $\in$ *compatible_vals* (*fv* (*r* $\dagger$ *ts*)) *vs* $\Longrightarrow$
cs $\in$ *mk_values* (*vs*$\{\!|$*ts*$|\!\}$)
$\langle proof \rangle$

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

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

lemma *mk_values_alt*:
mk_values $(vs\{\!|ts|\!\})$ =
$\{cs.\ \exists\ v \in compatible\_vals\ (\bigcup (fv\_trm\ `\ set\ ts))\ vs.\ cs = v[\![ts]\!]\}$
⟨*proof*⟩

lemma *check_values_neq_NoneI*:
assumes $v \in compatible\_vals\ (\bigcup\ (fv\_trm\ `\ set\ ts) - dom\ f)\ vs\ \bigwedge x\ y.\ f\ x = Some\ y \Longrightarrow y \in vs\ x$
shows *check_values vs ts* $((\lambda x.\ case\ f\ x\ of\ None \Rightarrow v\ x \mid Some\ y \Rightarrow y)[\![ts]\!])$ (*Some f*) $\neq$ *None*
⟨*proof*⟩

lemma *check_values_eq_NoneI*:
$\forall\ v \in compatible\_vals\ (\bigcup\ (fv\_trm\ `\ set\ ts) - dom\ f)\ vs.\ us \neq (\lambda x.\ case\ f\ x\ of\ None \Rightarrow v\ x \mid Some\ y \Rightarrow y)[\![ts]\!] \Longrightarrow$
check_values vs ts us (*Some f*) = *None*
⟨*proof*⟩

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

9.2 Executable Variant of the Checker

fun *s_check_exec* :: ('*n*, '*d*) *envset* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *sproof* ⇒ *bool*
and *v_check_exec* :: ('*n*, '*d*) *envset* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *vproof* ⇒ *bool* **where**
s_check_exec vs f p = (*case* (*f*, *p*) *of*
$(\top,\ STT\ i) \Rightarrow True$
| $(r \dagger ts,\ SPred\ i\ s\ ts') \Rightarrow$
$(r = s \wedge ts = ts' \wedge mk\_values\_subset\ r\ (vs\{\!|ts|\!\})\ (\Gamma\ \sigma\ i))$
| $(x \approx c,\ SEq\_Const\ i\ x'\ c') \Rightarrow$
$(c = c' \wedge x = x' \wedge vs\ x = \{c\})$
| $(\neg_F\ \varphi,\ SNeg\ vp) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp$
| $(\varphi \vee_F \psi,\ SOrL\ sp1) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1$
| $(\varphi \vee_F \psi,\ SOrR\ sp2) \Rightarrow s\_check\_exec\ vs\ \psi\ sp2$
| $(\varphi \wedge_F \psi,\ SAnd\ sp1\ sp2) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1 \wedge s\_check\_exec\ vs\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2$
| $(\varphi \longrightarrow_F \psi,\ SImpL\ vp1) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1$
| $(\varphi \longrightarrow_F \psi,\ SImpR\ sp2) \Rightarrow s\_check\_exec\ vs\ \psi\ sp2$
| $(\varphi \longleftrightarrow_F \psi,\ SIffSS\ sp1\ sp2) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1 \wedge s\_check\_exec\ vs\ \psi\ sp2 \wedge s\_at\ sp1 = s\_at\ sp2$
| $(\varphi \longleftrightarrow_F \psi,\ SIffVV\ vp1\ vp2) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1 \wedge v\_check\_exec\ vs\ \psi\ vp2 \wedge v\_at\ vp1 = v\_at\ vp2$
| $(\exists_F x.\ \varphi,\ SExists\ y\ val\ sp) \Rightarrow (x = y \wedge s\_check\_exec\ (vs\ (x := \{val\}))\ \varphi\ sp)$
| $(\forall_F x.\ \varphi,\ SForall\ y\ sp\_part) \Rightarrow (let\ i = s\_at\ (part\_hd\ sp\_part)$
$in\ x = y \wedge (\forall (sub, sp) \in SubsVals\ sp\_part.\ s\_at\ sp = i \wedge s\_check\_exec\ (vs\ (x := sub))\ \varphi\ sp))$
| $(\mathbf{Y}\ I\ \varphi,\ SPrev\ sp) \Rightarrow$
$(let\ j = s\_at\ sp;\ i = s\_at\ (SPrev\ sp)\ in$
$i = j{+}1 \wedge mem\ (\Delta\ \sigma\ i)\ I \wedge s\_check\_exec\ vs\ \varphi\ sp)$
| $(\mathbf{X}\ I\ \varphi,\ SNext\ sp) \Rightarrow$

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

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

| $(\triangleright\ I\ r,\ VMatchF\ i\ rvps) \Rightarrow$
  $(let\ j = LTP\ \sigma\ (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ n \Rightarrow \tau\ \sigma\ i + n)$
  $in\ map\ (snd \circ rv\_at\ v\_at)\ rvps = [ETP\_f\ \sigma\ i\ I\ ..< Suc\ j] \wedge right\ I \neq \infty \wedge$
  $(\forall rvp \in set\ rvps.\ rv\_check\ (v\_check\_exec\ vs)\ v\_at\ r\ rvp \wedge fst\ (rv\_at\ v\_at\ rvp) = i))$
| $(\_ , \_) \Rightarrow False)$

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

lemma *check_fv_cong*:
  assumes $\forall x \in fv\ \varphi.\ v\ x = v'\ x$
  shows $s\_check\ v\ \varphi\ sp \longleftrightarrow s\_check\ v'\ \varphi\ sp\ v\_check\ v\ \varphi\ vp \longleftrightarrow v\_check\ v'\ \varphi\ vp$
  ⟨*proof*⟩

lemma *s_check_fun_upd_notin*[*simp*]:
  $x \notin fv\ \varphi \Longrightarrow s\_check\ (v(x := t))\ \varphi\ sp = s\_check\ v\ \varphi\ sp$
  ⟨*proof*⟩
lemma *v_check_fun_upd_notin*[*simp*]:
  $x \notin fv\ \varphi \Longrightarrow v\_check\ (v(x := t))\ \varphi\ sp = v\_check\ v\ \varphi\ sp$
  ⟨*proof*⟩

lemma *SubsVals_nonempty*: $(X,\ t) \in SubsVals\ part \Longrightarrow X \neq \{\}$
  ⟨*proof*⟩

lemma *compatible_vals_nonemptyI*: $\forall x.\ vs\ x \neq \{\} \Longrightarrow compatible\_vals\ A\ vs \neq \{\}$
  ⟨*proof*⟩

lemma *compatible_vals_fun_upd*: $compatible\_vals\ A\ (vs(x := X)) =$
  $(if\ x \in A\ then\ \{v \in compatible\_vals\ (A - \{x\})\ vs.\ v\ x \in X\}\ else\ compatible\_vals\ A\ vs)$
  ⟨*proof*⟩

lemma *fun_upd_in_compatible_vals*: $v \in compatible\_vals\ (A - \{x\})\ vs \Longrightarrow v(x := t) \in compatible\_vals\ (A - \{x\})\ vs$
  ⟨*proof*⟩

lemma *fun_upd_in_compatible_vals_in*: $v \in compatible\_vals\ (A - \{x\})\ vs \Longrightarrow t \in vs\ x \Longrightarrow v(x := t) \in compatible\_vals\ A\ vs$
  ⟨*proof*⟩

lemma *fun_upd_in_compatible_vals_notin*: $x \notin A \Longrightarrow v \in compatible\_vals\ A\ vs \Longrightarrow v(x := t) \in compatible\_vals\ A\ vs$
  ⟨*proof*⟩

lemma *check_exec_check*:
  assumes $\forall x.\ vs\ x \neq \{\}$
  shows $s\_check\_exec\ vs\ \varphi\ sp \longleftrightarrow (\forall v \in compatible\_vals\ (fv\ \varphi)\ vs.\ s\_check\ v\ \varphi\ sp)$
    and $v\_check\_exec\ vs\ \varphi\ vp \longleftrightarrow (\forall v \in compatible\_vals\ (fv\ \varphi)\ vs.\ v\_check\ v\ \varphi\ vp)$
  ⟨*proof*⟩

lemma *s_check_code*[*code*]: $s\_check\ v\ \varphi\ sp = s\_check\_exec\ (\lambda x.\ \{v\ x\})\ \varphi\ sp$
  ⟨*proof*⟩

lemma *v_check_code*[*code*]: $v\_check\ v\ \varphi\ vp = v\_check\_exec\ (\lambda x.\ \{v\ x\})\ \varphi\ vp$
  ⟨*proof*⟩

9.3 Latest Relevant Time-Point

fun $rLRTP :: ('a \Rightarrow nat \Rightarrow nat\ option) \Rightarrow 'a\ Regex.regex \Rightarrow nat \Rightarrow nat\ option$ **where**
$rLRTP\ LRTP\ (Regex.Skip\ n)\ i = Some\ i$
$|\ rLRTP\ LRTP\ (Regex.Test\ x)\ i = LRTP\ x\ i$
$|\ rLRTP\ LRTP\ (Regex.Plus\ r\ s)\ i = max\_opt\ (rLRTP\ LRTP\ r\ i)\ (rLRTP\ LRTP\ s\ i)$
$|\ rLRTP\ LRTP\ (Regex.Times\ r\ s)\ i = max\_opt\ (rLRTP\ LRTP\ r\ i)\ (rLRTP\ LRTP\ s\ i)$
$|\ rLRTP\ LRTP\ (Regex.Star\ r)\ i = rLRTP\ LRTP\ r\ i$

lemma $rLRTP\_cong[fundef\_cong]$:
$(\bigwedge x.\ x \in regex.atms\ r \Longrightarrow LRTP\ x\ i = LRTP'\ x\ i) \Longrightarrow rLRTP\ LRTP\ r\ i = rLRTP\ LRTP'\ r\ i$
$\langle proof \rangle$

lemma fb_rLRTP:
assumes $\forall \varphi \in regex.atms\ r.\ future\_bounded\ \varphi \wedge \neg\ Option.is\_none\ (LRTP\ \varphi\ i)$
shows $\neg\ Option.is\_none\ (rLRTP\ LRTP\ r\ i)$
$\langle proof \rangle$

fun $LRTP :: ('n, 'd)\ formula \Rightarrow nat \Rightarrow nat\ option$ **where**
$LRTP\ \top\ i = Some\ i$
$|\ LRTP\ \bot\ i = Some\ i$
$|\ LRTP\ (\_\ \dagger\ \_)\ i = Some\ i$
$|\ LRTP\ (\_\ \approx\ \_)\ i = Some\ i$
$|\ LRTP\ (\neg_F\ \varphi)\ i = LRTP\ \varphi\ i$
$|\ LRTP\ (\varphi \vee_F \psi)\ i = max\_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
$|\ LRTP\ (\varphi \wedge_F \psi)\ i = max\_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
$|\ LRTP\ (\varphi \longrightarrow_F \psi)\ i = max\_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
$|\ LRTP\ (\varphi \longleftrightarrow_F \psi)\ i = max\_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ i)$
$|\ LRTP\ (\exists_F\_.\ \varphi)\ i = LRTP\ \varphi\ i$
$|\ LRTP\ (\forall_F\_.\ \varphi)\ i = LRTP\ \varphi\ i$
$|\ LRTP\ (\mathbf{Y}\ I\ \varphi)\ i = LRTP\ \varphi\ (i-1)$
$|\ LRTP\ (\mathbf{X}\ I\ \varphi)\ i = LRTP\ \varphi\ (i+1)$
$|\ LRTP\ (\mathbf{P}\ I\ \varphi)\ i = LRTP\ \varphi\ (LTP\_p\_safe\ \sigma\ i\ I)$
$|\ LRTP\ (\mathbf{H}\ I\ \varphi)\ i = LRTP\ \varphi\ (LTP\_p\_safe\ \sigma\ i\ I)$
$|\ LRTP\ (\mathbf{F}\ I\ \varphi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP\_f\ \sigma\ i\ b))$
$|\ LRTP\ (\mathbf{G}\ I\ \varphi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow LRTP\ \varphi\ (LTP\_f\ \sigma\ i\ b))$
$|\ LRTP\ (\varphi\ \mathbf{S}\ I\ \psi)\ i = max\_opt\ (LRTP\ \varphi\ i)\ (LRTP\ \psi\ (LTP\_p\_safe\ \sigma\ i\ I))$
$|\ LRTP\ (\varphi\ \mathbf{U}\ I\ \psi)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow max\_opt\ (LRTP\ \varphi\ ((LTP\_f\ \sigma\ i\ b)-1))$
$(LRTP\ \psi\ (LTP\_f\ \sigma\ i\ b)))$
$|\ LRTP\ (\triangleleft\ I\ r)\ i =$
$(let\ X = (\lambda\varphi.\ LRTP\ \varphi\ i)\ `\ regex.atms\ r\ in$
$if\ X = \{\}\ then\ Some\ i\ else\ if\ None \in X\ then\ None\ else\ Some\ (Max\ (the\ `\ X)))$
$|\ LRTP\ (\triangleright\ I\ r)\ i = (case\ right\ I\ of\ \infty \Rightarrow None\ |\ enat\ b \Rightarrow$
$let\ X = (\lambda\varphi.\ LRTP\ \varphi\ (LTP\_f\ \sigma\ i\ b))\ `\ regex.atms\ r\ in$
$if\ X = \{\}\ then\ Some\ (LTP\_f\ \sigma\ i\ b)\ else\ if\ None \in X\ then\ None\ else\ Some\ (Max\ (the\ `\ X)))$

lemma fb_LRTP:
assumes $future\_bounded\ \varphi$
shows $\neg\ Option.is\_none\ (LRTP\ \varphi\ i)$
$\langle proof \rangle$

lemma $not\_none\_fb\_LRTP$:
assumes $future\_bounded\ \varphi$
shows $LRTP\ \varphi\ i \neq None$
$\langle proof \rangle$

lemma $is\_some\_fb\_LRTP$:
assumes $future\_bounded\ \varphi$
shows $\exists j.\ LRTP\ \varphi\ i = Some\ j$

⟨*proof*⟩

lemma *enat_trans*[*simp*]: $enat\ i \leq enat\ j \wedge enat\ j \leq enat\ k \Longrightarrow enat\ i \leq enat\ k$
⟨*proof*⟩

9.4 Active Domain

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

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

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

lemma *finite_values*: $finite\ (\bigcup\ (set\ `\ snd\ `\ \Gamma\ \sigma\ k))$
⟨*proof*⟩

lemma *finite_tps*: $future\_bounded\ \varphi \Longrightarrow finite\ (\bigcup\ k < the\ (LRTP\ \varphi\ i).\ \{k\})$
⟨*proof*⟩

lemma *finite_AD* [*simp*]: $future\_bounded\ \varphi \Longrightarrow finite\ (AD\ \varphi\ i)$
⟨*proof*⟩

lemma *finite_AD_UNIV*:
assumes $future\_bounded\ \varphi$ **and** $AD\ \varphi\ i = (UNIV :: 'd\ set)$
shows $finite\ (UNIV :: 'd\ set)$
⟨*proof*⟩

9.5 Congruence Modulo Active Domain

lemma *AD_simps*[*simp*]:
$AD\ (\neg_F\ \varphi)\ i = AD\ \varphi\ i$
$future\_bounded\ (\varphi \vee_F \psi) \Longrightarrow AD\ (\varphi \vee_F \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
$future\_bounded\ (\varphi \wedge_F \psi) \Longrightarrow AD\ (\varphi \wedge_F \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
$future\_bounded\ (\varphi \longrightarrow_F \psi) \Longrightarrow AD\ (\varphi \longrightarrow_F \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
$future\_bounded\ (\varphi \longleftrightarrow_F \psi) \Longrightarrow AD\ (\varphi \longleftrightarrow_F \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ i$
$AD\ (\exists_F x.\ \ \varphi)\ i = AD\ \varphi\ i$
$AD\ (\forall_F x.\ \ \varphi)\ i = AD\ \varphi\ i$
$AD\ (\mathbf{Y}\ I\ \varphi)\ i = AD\ \varphi\ (i - 1)$
$AD\ (\mathbf{X}\ I\ \varphi)\ i = AD\ \varphi\ (i + 1)$
$future\_bounded\ (\mathbf{F}\ I\ \varphi) \Longrightarrow AD\ (\mathbf{F}\ I\ \varphi)\ i = AD\ \varphi\ (LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I)))$
$future\_bounded\ (\mathbf{G}\ I\ \varphi) \Longrightarrow AD\ (\mathbf{G}\ I\ \varphi)\ i = AD\ \varphi\ (LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I)))$
$AD\ (\mathbf{P}\ I\ \varphi)\ i = AD\ \varphi\ (LTP\_p\_safe\ \sigma\ i\ I)$
$AD\ (\mathbf{H}\ I\ \varphi)\ i = AD\ \varphi\ (LTP\_p\_safe\ \sigma\ i\ I)$
$future\_bounded\ (\varphi\ \mathbf{S}\ I\ \psi) \Longrightarrow AD\ (\varphi\ \mathbf{S}\ I\ \psi)\ i = AD\ \varphi\ i \cup AD\ \psi\ (LTP\_p\_safe\ \sigma\ i\ I)$
$future\_bounded\ (\varphi\ \mathbf{U}\ I\ \psi) \Longrightarrow AD\ (\varphi\ \mathbf{U}\ I\ \psi)\ i = AD\ \varphi\ (LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I)) - 1) \cup AD\ \psi\ (LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I)))$
⟨*proof*⟩

lemma *AD_simps_regex*[*simp*]:
$future\_bounded\ (\lhd\ I\ r) \Longrightarrow regex.atms\ r \neq \{\} \Longrightarrow AD\ (\lhd\ I\ r)\ i = (\bigcup \varphi \in regex.atms\ r.\ AD\ \varphi\ i)$
$future\_bounded\ (\rhd\ I\ r) \Longrightarrow regex.atms\ r \neq \{\} \Longrightarrow AD\ (\rhd\ I\ r)\ i = (\bigcup \varphi \in regex.atms\ r.\ AD\ \varphi\ (LTP\_f\ \sigma\ i\ (the\_enat\ (right\ I))))$

$\langle proof \rangle$

lemma *LTP_p_mono*: $i \leq j \Longrightarrow LTP\_p\_safe\ \sigma\ i\ I \leq LTP\_p\_safe\ \sigma\ j\ I$
$\langle proof \rangle$

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

lemma *LTP_f_mono*:
assumes $i \leq j$
shows $LTP\_f\ \sigma\ i\ b \leq LTP\_f\ \sigma\ j\ b$
$\langle proof \rangle$

lemma *LRTP_mono*: $future\_bounded\ \varphi \Longrightarrow i \leq j \Longrightarrow the\ (LRTP\ \varphi\ i) \leq the\ (LRTP\ \varphi\ j)$
$\langle proof \rangle$

lemma *AD_mono*: $future\_bounded\ \varphi \Longrightarrow i \leq j \Longrightarrow AD\ \varphi\ i \subseteq AD\ \varphi\ j$
$\langle proof \rangle$

lemma *LTP_p_safe_le*[*simp*]: $LTP\_p\_safe\ \sigma\ i\ I \leq i$
$\langle proof \rangle$

lemma *check_AD_cong*:
assumes $future\_bounded\ \varphi$
and $(\forall x \in fv\ \varphi.\ v\ x = v'\ x \lor (v\ x \notin AD\ \varphi\ i \land v'\ x \notin AD\ \varphi\ i))$
shows $(s\_at\ sp = i \Longrightarrow s\_check\ v\ \varphi\ sp \longleftrightarrow s\_check\ v'\ \varphi\ sp)$
$(v\_at\ vp = i \Longrightarrow v\_check\ v\ \varphi\ vp \longleftrightarrow v\_check\ v'\ \varphi\ vp)$
$\langle proof \rangle$

9.6 Checker Completeness

lemma *part_hd_tabulate*: $distinct\ xs \Longrightarrow part\_hd\ (tabulate\ xs\ f\ z) = (case\ xs\ of\ [] \Rightarrow z \mid (x\ \#\ \_) \Rightarrow (if\ set\ xs = UNIV\ then\ f\ x\ else\ z))$
$\langle proof \rangle$

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 $\forall (sub, vp) \in SubsVals\ mypart.\ s\_at\ vp = i$
$\langle proof \rangle$

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 $\forall (sub, vp) \in SubsVals\ mypart.\ v\_at\ vp = i$
$\langle proof \rangle$

lemma *s_check_tabulate*:
assumes $future\_bounded\ \varphi$
and $\forall z.\ s\_at\ (mypick\ z) = i$
and $\forall z.\ s\_check\ (v(x{:=}z))\ \varphi\ (mypick\ z)$
and $mypart = tabulate\ (sorted\_list\_of\_set\ (AD\ \varphi\ i))\ mypick\ (mypick\ (SOME\ z.\ z \notin AD\ \varphi\ i))$
shows $\forall (sub, vp) \in SubsVals\ mypart.\ \forall z \in sub.\ s\_check\ (v(x := z))\ \varphi\ vp$
$\langle proof \rangle$

lemma *v_check_tabulate*:

assumes *future_bounded* φ
and $\forall z.$ *v_at* (*mypick z*) = *i*
and $\forall z.$ *v_check* ($v(x{:=}z)$) φ (*mypick z*)
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* φ *i*)) *mypick* (*mypick* (*SOME z. z* $\notin$ *AD* φ *i*))
shows $\forall$ (*sub*, *vp*) $\in$ *SubsVals mypart*. $\forall z \in$ *sub*. *v_check* ($v(x := z)$) φ *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* φ *i*)) *f* (*f* (*SOME z. z* $\notin$ *AD* φ *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* φ *i*)) *f* (*f* (*SOME z. z* $\notin$ *AD* φ *i*))
shows *v_at* (*part_hd mypart*) = *i*
⟨*proof*⟩

lemma *check_completeness_aux*:
(*SAT* σ *v i* $\varphi \longrightarrow$ *future_bounded* $\varphi \longrightarrow$ ($\exists$ *sp. s_at sp* = *i* $\wedge$ *s_check v* φ *sp*)) $\wedge$
(*VIO* σ *v i* $\varphi \longrightarrow$ *future_bounded* $\varphi \longrightarrow$ ($\exists$ *vp. v_at vp* = *i* $\wedge$ *v_check v* φ *vp*))
⟨*proof*⟩

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

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

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

end

9.7 Lifting the Checker to PDTs

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

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

fun *collect_paths_aux* **where**
collect_paths_aux DS σ *vs* φ (*Leaf p*) = (*if p_check_exec* σ *vs* φ *p then* {} *else rev* ‘ *DS*)
| *collect_paths_aux DS* σ *vs* φ (*Node x part*) = ($\bigcup$ (*D*, *e*) $\in$ *set* (*subsvals part*). *collect_paths_aux* (*Cons D* ‘ *DS*) σ ($vs(x := D)$) φ *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*: $pdt\_at\ i\ e \Longrightarrow p\_at\ (eval\_pdt\ v\ e) = i$
⟨*proof*⟩

lemma *check_all_completeness_aux*:
fixes $\varphi :: ('n, 'd :: \{default, linorder\})\ formula$
shows $set\ vs \subseteq fv\ \varphi \Longrightarrow future\_bounded\ \varphi \Longrightarrow distinct\ vs \Longrightarrow$
$\exists e.\ pdt\_at\ i\ e \wedge vars\_order\ vs\ e \wedge (\forall v.\ (\forall x.\ x \notin set\ vs \longrightarrow v\ x = w\ x) \longrightarrow p\_check\ \sigma\ v\ \varphi\ (eval\_pdt\ v\ e))$
⟨*proof*⟩

lemma *check_all_completeness*:
fixes $\varphi :: ('n, 'd :: \{default, linorder\})\ formula$
assumes $future\_bounded\ \varphi$
shows $\exists e.\ pdt\_at\ i\ e \wedge check\_all\ \sigma\ \varphi\ e$
⟨*proof*⟩

lemma *check_all_soundness_aux*: $check\_all\ \sigma\ \varphi\ e \Longrightarrow p = eval\_pdt\ v\ e \Longrightarrow isl\ p \longleftrightarrow sat\ \sigma\ v\ (p\_at\ p)\ \varphi$
⟨*proof*⟩

lemma *check_all_soundness*: $check\_all\ \sigma\ \varphi\ e \Longrightarrow pdt\_at\ i\ e \Longrightarrow isl\ (eval\_pdt\ v\ e) \longleftrightarrow sat\ \sigma\ v\ i\ \varphi$
⟨*proof*⟩

unbundle *no MFOTL_syntax*

10 Type of Events

10.1 Code Adaptation for 8-bit strings

typedef *string8* = *UNIV* :: *char list set* ⟨*proof*⟩

setup_lifting *type_definition_string8*

lift_definition *empty_string* :: *string8* **is** [] ⟨*proof*⟩
lift_definition *string8_literal* :: *String.literal* ⇒ *string8* **is** *String.explode* ⟨*proof*⟩

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

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

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

instance ⟨*proof*⟩

end

lifting_forget *string8.lifting*

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

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

⟨*ML*⟩

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

⟨*ML*⟩

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

10.2 Event Parameters

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

definition *mod_to_zero* :: *integer* ⇒ *integer* ⇒ *integer* **where**
mod_to_zero x y = (*let z* = *snd* (*Code_Numeral.divmod_abs x y*) *in*

$if\ x < 0\ then\ -\ z\ else\ z)$

lemma $b \neq 0 \Longrightarrow div\_to\_zero\ a\ b * b + mod\_to\_zero\ a\ b = a$
⟨*proof*⟩

datatype $event\_data = EInt\ integer \mid EString\ string8$

instantiation $event\_data :: \{ord, plus, minus, uminus, times, divide, modulo\}$
begin

fun *less_eq_event_data* **where**
$EInt\ x \leq EInt\ y \longleftrightarrow x \leq y$
$\mid EString\ x \leq EString\ y \longleftrightarrow x \leq y$
$\mid EInt\ \_ \leq EString\ \_ \longleftrightarrow True$
$\mid (\_ :: event\_data) \leq \_ \longleftrightarrow False$

definition $less\_event\_data :: event\_data \Rightarrow event\_data \Rightarrow bool$ **where**
$less\_event\_data\ x\ y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$

fun *plus_event_data* **where**
$EInt\ x + EInt\ y = EInt\ (x + y)$
$\mid (\_::event\_data) + \_ = undefined$

fun *minus_event_data* **where**
$EInt\ x - EInt\ y = EInt\ (x - y)$
$\mid (\_::event\_data) - \_ = undefined$

fun *uminus_event_data* **where**
$-\ EInt\ x = EInt\ (-\ x)$
$\mid -\ (\_::event\_data) = undefined$

fun *times_event_data* **where**
$EInt\ x * EInt\ y = EInt\ (x * y)$
$\mid (\_::event\_data) * \_ = undefined$

fun *divide_event_data* **where**
$EInt\ x\ div\ EInt\ y = EInt\ (div\_to\_zero\ x\ y)$
$\mid (\_::event\_data)\ div\ \_ = undefined$

fun *modulo_event_data* **where**
$EInt\ x\ mod\ EInt\ y = EInt\ (mod\_to\_zero\ x\ y)$
$\mid (\_::event\_data)\ mod\ \_ = undefined$

instance ⟨*proof*⟩

end

lemma *infinite_UNIV_event_data*:
$\neg finite\ (UNIV :: event\_data\ set)$
⟨*proof*⟩

primrec $integer\_of\_event\_data :: event\_data \Rightarrow integer$ **where**
$integer\_of\_event\_data\ (EInt\ \_) = undefined$
$\mid integer\_of\_event\_data\ (EString\ \_) = undefined$

instantiation $event\_data :: default$ **begin**

definition $default\_event\_data :: event\_data$ **where** $default = EInt\ 0$

instance ⟨*proof*⟩

end

instantiation *event_data* :: *linorder* **begin**
instance
⟨*proof*⟩

end

11 Code Generation

11.1 Type Class Instances

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

lemma *finite_coset*: *finite* (*List.coset* (*xs* :: *'a list*)) = (*case universe of None* ⇒ *False* | _ ⇒ *True*)
  ⟨*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 × 'b) *list option* **where** *universe_prod* =
  (*case* (*universe*, *universe*) *of* (*Some xs*, *Some ys*) ⇒ *Some* (*List.product xs ys*) | _ ⇒ *None*)
instance ⟨*proof*⟩
end
instantiation *sum* :: (*universe*, *universe*) *universe* **begin**
definition *universe_sum* :: ('a + 'b) *list option* **where** *universe_sum* =
  (*case* (*universe*, *universe*) *of* (*Some xs*, *Some ys*) ⇒ *Some* (*map Inl xs* @ *map Inr ys*) | _ ⇒ *None*)
instance ⟨*proof*⟩
end
instantiation *option* :: (*universe*) *universe* **begin**
definition *universe_option* = (*case universe of Some xs* ⇒ *Some* (*None* # *map Some xs*) | _ ⇒ *None*)
instance ⟨*proof*⟩
end
instantiation *fun* :: (*universe*, *universe*) *universe* **begin**
definition *universe_fun* :: ('a ⇒ 'b) *list option* **where** *universe_fun* =
  (*case* (*universe*, *universe*) *of*
    (*Some xs*, *Some ys*) ⇒ *Some* (*map* (λ*zs. the* ∘ *map_of* (*zip xs zs*)) (*List.n_lists* (*length xs*) *ys*))
  | (_, *Some* [*x*]) ⇒ *Some* [λ_. *x*]
  | _ ⇒ *None*)
instance
⟨*proof*⟩
end
instantiation *event_data* :: *universe* **begin**
definition *universe_event_data* :: *event_data list option* **where** *universe_event_data* = *None*
instance ⟨*proof*⟩
end

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

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

instance *event_data* :: *equal* ⟨*proof*⟩

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

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

11.2 Progress

fun *progress* :: ('n, 'd) *trace* ⇒ ('n, 'd) *Formula.formula* ⇒ *nat* ⇒ *nat* **where**
  progress σ *Formula.TT j* = *j*
| *progress* σ *Formula.FF j* = *j*
| *progress* σ (*Formula.Eq_Const* _ _) *j* = *j*
| *progress* σ (*Formula.Pred* _ _) *j* = *j*
| *progress* σ (*Formula.Neg* φ) *j* = *progress* σ φ *j*

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

lemma *Inf_Min*:
  fixes $P :: nat \Rightarrow bool$
  assumes $P\ j$
  shows $Inf\ (Collect\ P) = Min\ (Set.filter\ P\ \{..j\})$
  ⟨*proof*⟩

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

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

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

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

11.3 Trace

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

lemma *extend_is_stream*:
  assumes $sorted\ (map\ snd\ list)$
  and $\bigwedge x.\ x \in set\ list \Longrightarrow snd\ x \le m$
  and $\bigwedge x.\ x \in set\ list \Longrightarrow finite\ (fst\ x)$
  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 $'a\ trace\_mapping = \{(n,\ m,\ t) :: (nat \times nat \times (nat,\ 'a\ set \times nat)\ mapping) \mid$

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

setup_lifting *type_definition_trace_mapping*

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

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

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

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

code_datatype *Trace_Mapping*

definition $trace\_of\_list\ xs = Trace\_Mapping\ (trace\_mapping\_of\_list\ xs)$

lemma Γ*_rbt_code*[*code*]: $\Gamma\ (Trace\_Mapping\ t)\ i = fst\ (trace\_mapping\_nth\ t\ i)$
$\langle proof \rangle$

lemma τ*_rbt_code*[*code*]: $\tau\ (Trace\_Mapping\ t)\ i = snd\ (trace\_mapping\_nth\ t\ i)$
$\langle proof \rangle$

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

11.4 Auxiliary results

definition $sum\_proofs\ f\ xs = sum\_list\ (map\ f\ xs)$

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

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

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

lemma *sum_proofs_app*:

fixes f :: $'a \Rightarrow nat$
shows $sum\_proofs\ f\ (qs\ @\ [p]) = f\ p + sum\_proofs\ f\ qs$
$\langle proof \rangle$

context
fixes w :: $'n \Rightarrow nat$
begin

function $(sequential)\ s\_pred :: ('n,\ 'd)\ sproof \Rightarrow nat$
and $v\_pred :: ('n,\ 'd)\ vproof \Rightarrow nat$ **where**
$s\_pred\ (STT\ \_) = 1$
| $s\_pred\ (SEq\_Const\ \_\ \_\ \_) = 1$
| $s\_pred\ (SPred\ \_\ r\ \_) = w\ r$
| $s\_pred\ (SNeg\ vp) = (v\_pred\ vp) + 1$
| $s\_pred\ (SOrL\ sp1) = (s\_pred\ sp1) + 1$
| $s\_pred\ (SOrR\ sp2) = (s\_pred\ sp2) + 1$
| $s\_pred\ (SAnd\ sp1\ sp2) = (s\_pred\ sp1) + (s\_pred\ sp2) + 1$
| $s\_pred\ (SImpL\ vp1) = (v\_pred\ vp1) + 1$
| $s\_pred\ (SImpR\ sp2) = (s\_pred\ sp2) + 1$
| $s\_pred\ (SIffSS\ sp1\ sp2) = (s\_pred\ sp1) + (s\_pred\ sp2) + 1$
| $s\_pred\ (SIffVV\ vp1\ vp2) = (v\_pred\ vp1) + (v\_pred\ vp2) + 1$
| $s\_pred\ (SExists\ \_\ \_\ sp) = (s\_pred\ sp) + 1$
| $s\_pred\ (SForall\ \_\ part) = (sum\_proofs\ s\_pred\ (vals\ part)) + 1$
| $s\_pred\ (SPrev\ sp) = (s\_pred\ sp) + 1$
| $s\_pred\ (SNext\ sp) = (s\_pred\ sp) + 1$
| $s\_pred\ (SOnce\ \_\ sp) = (s\_pred\ sp) + 1$
| $s\_pred\ (SEventually\ \_\ sp) = (s\_pred\ sp) + 1$
| $s\_pred\ (SHistorically\ \_\ \_\ sps) = (sum\_proofs\ s\_pred\ sps) + 1$
| $s\_pred\ (SHistoricallyOut\ \_) = 1$
| $s\_pred\ (SAlways\ \_\ \_\ sps) = (sum\_proofs\ s\_pred\ sps) + 1$
| $s\_pred\ (SSince\ sp2\ sp1s) = (sum\_proofs\ s\_pred\ (sp2\ \#\ sp1s)) + 1$
| $s\_pred\ (SUntil\ sp1s\ sp2) = (sum\_proofs\ s\_pred\ (sp1s\ @\ [sp2])) + 1$
| $v\_pred\ (VFF\ \_) = 1$
| $v\_pred\ (VEq\_Const\ \_\ \_\ \_) = 1$
| $v\_pred\ (VPred\ \_\ r\ \_) = w\ r$
| $v\_pred\ (VNeg\ sp) = (s\_pred\ sp) + 1$
| $v\_pred\ (VOr\ vp1\ vp2) = ((v\_pred\ vp1) + (v\_pred\ vp2)) + 1$
| $v\_pred\ (VAndL\ vp1) = (v\_pred\ vp1) + 1$
| $v\_pred\ (VAndR\ vp2) = (v\_pred\ vp2) + 1$
| $v\_pred\ (VImp\ sp1\ vp2) = ((s\_pred\ sp1) + (v\_pred\ vp2)) + 1$
| $v\_pred\ (VIffSV\ sp1\ vp2) = ((s\_pred\ sp1) + (v\_pred\ vp2)) + 1$
| $v\_pred\ (VIffVS\ vp1\ sp2) = ((v\_pred\ vp1) + (s\_pred\ sp2)) + 1$
| $v\_pred\ (VExists\ \_\ part) = (sum\_proofs\ v\_pred\ (vals\ part)) + 1$
| $v\_pred\ (VForall\ \_\ \_\ vp) = (v\_pred\ vp) + 1$
| $v\_pred\ (VPrev\ vp) = (v\_pred\ vp) + 1$
| $v\_pred\ (VPrevZ) = 1$
| $v\_pred\ (VPrevOutL\ \_) = 1$
| $v\_pred\ (VPrevOutR\ \_) = 1$
| $v\_pred\ (VNext\ vp) = (v\_pred\ vp) + 1$
| $v\_pred\ (VNextOutL\ \_) = 1$
| $v\_pred\ (VNextOutR\ \_) = 1$
| $v\_pred\ (VOnceOut\ \_) = 1$
| $v\_pred\ (VOnce\ \_\ \_\ vps) = (sum\_proofs\ v\_pred\ vps) + 1$
| $v\_pred\ (VEventually\ \_\ \_\ vps) = (sum\_proofs\ v\_pred\ vps) + 1$
| $v\_pred\ (VHistorically\ \_\ vp) = (v\_pred\ vp) + 1$
| $v\_pred\ (VAlways\ \_\ vp) = (v\_pred\ vp) + 1$
| $v\_pred\ (VSinceOut\ \_) = 1$
| $v\_pred\ (VSince\ \_\ vp1\ vp2s) = (sum\_proofs\ v\_pred\ (vp1\ \#\ vp2s)) + 1$

| *v_pred* (*VSinceInf* _ _ *vp2s*) = (*sum_proofs v_pred vp2s*) + *1*
| *v_pred* (*VUntil* _ *vp2s vp1*) = (*sum_proofs v_pred* (*vp2s* @ [*vp1*])) + *1*
| *v_pred* (*VUntilInf* _ _ *vp2s*) = (*sum_proofs v_pred vp2s*) + *1*
  ⟨*proof*⟩
termination
  ⟨*proof*⟩

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

end

11.5 *v_check_exec* setup

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

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

lemma *nat_le_iff_less*:
  fixes *n* :: *nat*
  shows $(j \leq n) \longleftrightarrow (j = 0 \lor j - 1 < n)$
  ⟨*proof*⟩

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

lemma *LTP_minus_ge_iff*: $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i \Longrightarrow j \leq LTP\ \sigma\ (\tau\ \sigma\ i - n) \longleftrightarrow$
  $(case\ n\ of\ 0 \Rightarrow \delta\ \sigma\ j\ i = 0 \mid \_ \Rightarrow j \leq i \land \delta\ \sigma\ i\ j \geq n)$
  ⟨*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_if*:
  assumes $j \leq i\ \tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i\ \delta\ \sigma\ i\ j < n$
  shows $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j$
⟨*proof*⟩

lemma *LTP_minus_lt_iff*:
  assumes $\tau\ \sigma\ 0 + n \leq \tau\ \sigma\ i$
  shows $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j \longleftrightarrow (if\ \neg\ j \leq i \land n = 0\ then\ \delta\ \sigma\ j\ i > 0\ else\ \delta\ \sigma\ i\ j < n)$
⟨*proof*⟩

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

lemma *LTP_plus_eq_iff*:
  shows $j = LTP\ \sigma\ (\tau\ \sigma\ i + n) \longleftrightarrow (\delta\ \sigma\ j\ i \leq n \land \delta\ \sigma\ (Suc\ j)\ i > n)$
  ⟨*proof*⟩

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

⟨proof⟩

definition $\mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ \mathit{xs}\ i \longleftrightarrow (\mathit{case}\ \mathit{xs}\ \mathit{of}\ [] \Rightarrow$
$(\mathit{case}\ \mathit{left}\ I\ \mathit{of}\ 0 \Rightarrow i < \mathit{li} \mid \mathit{Suc}\ n \Rightarrow$
$(\mathit{if}\ \neg\ \mathit{li} \le i \wedge \mathit{left}\ I = 0\ \mathit{then}\ 0 < \delta\ \sigma\ \mathit{li}\ i\ \mathit{else}\ \delta\ \sigma\ i\ \mathit{li} < \mathit{left}\ I))$
$\mid \_ \Rightarrow \mathit{xs} = [\mathit{li}..<\mathit{li} + \mathit{length}\ \mathit{xs}] \wedge$
$(\mathit{case}\ \mathit{left}\ I\ \mathit{of}\ 0 \Rightarrow \mathit{li} + \mathit{length}\ \mathit{xs} - 1 = i \mid \mathit{Suc}\ n \Rightarrow$
$(\mathit{li} + \mathit{length}\ \mathit{xs} - 1 \le i \wedge \mathit{left}\ I \le \delta\ \sigma\ i\ (\mathit{li} + \mathit{length}\ \mathit{xs} - 1) \wedge \delta\ \sigma\ i\ (\mathit{li} + \mathit{length}\ \mathit{xs}) < \mathit{left}\ I)))$

lemma $\mathit{check\_upt\_l\_cong}$:
assumes $\bigwedge j.\ j \le \mathit{max}\ i\ \mathit{li} \Longrightarrow \tau\ \sigma\ j = \tau\ \sigma'\ j$
shows $\mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ \mathit{xs}\ i = \mathit{check\_upt\_LTP\_p}\ \sigma'\ I\ \mathit{li}\ \mathit{xs}\ i$
⟨proof⟩

lemma $\mathit{check\_upt\_LTP\_p\_eq}$:
assumes $\tau\ \sigma\ 0 + \mathit{left}\ I \le \tau\ \sigma\ i$
shows $\mathit{xs} = [\mathit{li}..<\mathit{Suc}\ (\mathit{LTP\_p}\ \sigma\ i\ I)] \longleftrightarrow \mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ \mathit{xs}\ i$
⟨proof⟩

lemma $\mathit{v\_check\_exec\_Once\_code}[\mathit{code}]$: $\mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ (\mathit{Formula.Once}\ I\ \varphi)\ \mathit{vp} = (\mathit{case}\ \mathit{vp}\ \mathit{of}$
$\mathit{VOnce}\ i\ \mathit{li}\ \mathit{vps} \Rightarrow$
$(\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow \mathit{li} = 0 \mid \mathit{enat}\ b \Rightarrow ((\mathit{li} = 0 \vee b < \delta\ \sigma\ i\ (\mathit{li} - 1)) \wedge \delta\ \sigma\ i\ \mathit{li} \le b))$
$\wedge\ \tau\ \sigma\ 0 + \mathit{left}\ I \le \tau\ \sigma\ i$
$\wedge\ \mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ (\mathit{map}\ \mathit{v\_at}\ \mathit{vps})\ i \wedge \mathit{Ball}\ (\mathit{set}\ \mathit{vps})\ (\mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ \varphi)$
$\mid \mathit{VOnceOut}\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + \mathit{left}\ I$
$\mid \_ \Rightarrow \mathit{False})$
⟨proof⟩

lemma $\mathit{s\_check\_exec\_Historically\_code}[\mathit{code}]$: $\mathit{s\_check\_exec}\ \sigma\ \mathit{vs}\ (\mathit{Formula.Historically}\ I\ \varphi)\ \mathit{vp} = (\mathit{case}$ $\mathit{vp}\ \mathit{of}$
$\mathit{SHistorically}\ i\ \mathit{li}\ \mathit{vps} \Rightarrow$
$(\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow \mathit{li} = 0 \mid \mathit{enat}\ b \Rightarrow ((\mathit{li} = 0 \vee b < \delta\ \sigma\ i\ (\mathit{li} - 1)) \wedge \delta\ \sigma\ i\ \mathit{li} \le b))$
$\wedge\ \tau\ \sigma\ 0 + \mathit{left}\ I \le \tau\ \sigma\ i$
$\wedge\ \mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ (\mathit{map}\ \mathit{s\_at}\ \mathit{vps})\ i \wedge \mathit{Ball}\ (\mathit{set}\ \mathit{vps})\ (\mathit{s\_check\_exec}\ \sigma\ \mathit{vs}\ \varphi)$
$\mid \mathit{SHistoricallyOut}\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + \mathit{left}\ I$
$\mid \_ \Rightarrow \mathit{False})$
⟨proof⟩

lemma $\mathit{v\_check\_exec\_Since\_code}[\mathit{code}]$: $\mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ (\mathit{Formula.Since}\ \varphi\ I\ \psi)\ \mathit{vp} = (\mathit{case}\ \mathit{vp}\ \mathit{of}$
$\mathit{VSince}\ i\ \mathit{vp1}\ \mathit{vp2s} \Rightarrow$
$\mathit{let}\ j = \mathit{v\_at}\ \mathit{vp1}\ \mathit{in}$
$(\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow \mathit{True} \mid \mathit{enat}\ b \Rightarrow \delta\ \sigma\ i\ j \le b) \wedge j \le i$
$\wedge\ \tau\ \sigma\ 0 + \mathit{left}\ I \le \tau\ \sigma\ i$
$\wedge\ \mathit{check\_upt\_LTP\_p}\ \sigma\ I\ j\ (\mathit{map}\ \mathit{v\_at}\ \mathit{vp2s})\ i$
$\wedge\ \mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ \varphi\ \mathit{vp1} \wedge \mathit{Ball}\ (\mathit{set}\ \mathit{vp2s})\ (\mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ \psi)$
$\mid \mathit{VSinceInf}\ i\ \mathit{li}\ \mathit{vp2s} \Rightarrow$
$(\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow \mathit{li} = 0 \mid \mathit{enat}\ b \Rightarrow ((\mathit{li} = 0 \vee b < \delta\ \sigma\ i\ (\mathit{li} - 1)) \wedge \delta\ \sigma\ i\ \mathit{li} \le b)) \wedge$
$\tau\ \sigma\ 0 + \mathit{left}\ I \le \tau\ \sigma\ i \wedge$
$\mathit{check\_upt\_LTP\_p}\ \sigma\ I\ \mathit{li}\ (\mathit{map}\ \mathit{v\_at}\ \mathit{vp2s})\ i \wedge \mathit{Ball}\ (\mathit{set}\ \mathit{vp2s})\ (\mathit{v\_check\_exec}\ \sigma\ \mathit{vs}\ \psi)$
$\mid \mathit{VSinceOut}\ i \Rightarrow \tau\ \sigma\ i < \tau\ \sigma\ 0 + \mathit{left}\ I$
$\mid \_ \Rightarrow \mathit{False})$
⟨proof⟩

lemma $\mathit{ETP\_f\_le\_iff}$: $\mathit{max}\ i\ (\mathit{ETP}\ \sigma\ (\tau\ \sigma\ i + a)) \le j \longleftrightarrow i \le j \wedge \delta\ \sigma\ j\ i \ge a$
⟨proof⟩

lemma $\mathit{ETP\_f\_ge\_iff}$: $j \le \mathit{max}\ i\ (\mathit{ETP}\ \sigma\ (\tau\ \sigma\ i + n)) \longleftrightarrow (\mathit{case}\ n\ \mathit{of}\ 0 \Rightarrow j \le i$
$\mid \mathit{Suc}\ n' \Rightarrow (\mathit{case}\ j\ \mathit{of}\ 0 \Rightarrow \mathit{True} \mid \mathit{Suc}\ j' \Rightarrow \delta\ \sigma\ j'\ i < n))$

⟨proof⟩

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

lemma *check_upt_lu_cong*:
assumes *⋀j. min i hi ≤ j ∧ j ≤ max i hi ⟹ τ σ j = τ σ′ j*
shows *check_upt_ETP_f σ I i xs hi = check_upt_ETP_f σ′ I i xs hi*
⟨proof⟩

lemma *check_upt_ETP_f_eq*: *xs = [ETP_f σ i I..<Suc hi] ⟷ check_upt_ETP_f σ I i xs hi*
⟨proof⟩

lemma *v_check_exec_Eventually_code*[*code*]: *v_check_exec σ vs (Formula.Eventually I φ) vp = (case vp of*
VEventually i hi vps ⇒
(case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i)) ∧
check_upt_ETP_f σ I i (map v_at vps) hi ∧ Ball (set vps) (v_check_exec σ vs φ)
| _ ⇒ False)
⟨proof⟩

lemma *s_check_exec_Always_code*[*code*]: *s_check_exec σ vs (Formula.Always I φ) sp = (case sp of*
SAlways i hi sps ⇒
(case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i))
∧ check_upt_ETP_f σ I i (map s_at sps) hi ∧ Ball (set sps) (s_check_exec σ vs φ)
| _ ⇒ False)
⟨proof⟩

lemma *v_check_exec_Until_code*[*code*]: *v_check_exec σ vs (Formula.Until φ I ψ) vp = (case vp of*
VUntil i vp2s vp1 ⇒
let j = v_at vp1 in
(case right I of ∞ ⇒ True | enat b ⇒ j < LTP_f σ i b)
∧ i ≤ j ∧ check_upt_ETP_f σ I i (map v_at vp2s) j
∧ v_check_exec σ vs φ vp1 ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
| VUntilInf i hi vp2s ⇒
(case right I of ∞ ⇒ False | enat b ⇒ (δ σ hi i ≤ b ∧ b < δ σ (Suc hi) i)) ∧
check_upt_ETP_f σ I i (map v_at vp2s) hi ∧ Ball (set vp2s) (v_check_exec σ vs ψ)
| _ ⇒ False)
⟨proof⟩

11.6 ETP/LTP setup

lemma *ETP_aux*: *¬ t ≤ τ σ i ⟹ i ≤ (LEAST i. t ≤ τ σ i)*
⟨proof⟩

function *ETP_rec* **where**
ETP_rec σ t i = (if τ σ i ≥ t then i else ETP_rec σ t (i + 1))
⟨proof⟩
termination
⟨proof⟩

lemma *ETP_rec_sound*: *ETP_rec σ t j = (LEAST i. i ≥ j ∧ t ≤ τ σ i)*
⟨proof⟩

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

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

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

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

lemma *LTP_code*[*code*]: $LTP\ \sigma\ t = (if\ t < \tau\ \sigma\ 0$
then Code.abort (*STR ''LTP: undefined''*) (λ_. *LTP* σ *t*)
else LTP_rec σ *t 0*)
⟨*proof*⟩

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

lemma *coset_subset_set_code*[*code*]:
(*List.coset* (*xs* :: _ :: *universe list*) ⊆ *set ys*) = (*case universe of None* ⇒ *False*
| *Some zs* ⇒ ∀ *z* ∈ *set zs*. *z* ∈ *set xs* ∨ *z* ∈ *set ys*)
⟨*proof*⟩

lemma *is_empty_coset*[*code*]: *Set.is_empty* (*List.coset* (*xs* :: _ :: *universe list*)) =
(*case universe of None* ⇒ *False*
| *Some zs* ⇒ ∀ *z* ∈ *set zs*. *z* ∈ *set xs*)
⟨*proof*⟩

11.7 Exported functions

type_synonym *name* = *string8*

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

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

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

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

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

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

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

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

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

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

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

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

12 Unverified Explanation-Producing Monitoring Algorithm

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

fun *merge_part3_raw* :: (*'a* ⇒ *'b* ⇒ *'c* ⇒ *'e*) ⇒ (*'d set* × *'a*) *list* ⇒ (*'d set* × *'b*) *list* ⇒ (*'d set* × *'c*) *list* ⇒ (*'d set* × *'e*) *list* **where**
merge_part3_raw f [] _ _ = []
| *merge_part3_raw f* _ [] _ = []
| *merge_part3_raw f* _ _ [] = []
| *merge_part3_raw f part1 part2 part3* = *merge_part2_raw* (λ*pt3 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*
  $\equiv$ *List.map_filter* $(\lambda(P2, v2).$ *if* $P1 \cap P2 \neq \{\}$ *then* $Some(P1 \cap P2, f\ v1\ v2)$ *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*)) $\Longrightarrow$ *distinct* (*map fst* (*part2*)) $\Longrightarrow$
  distinct (*map fst* (*inP1 P1 f v1 part2*))
⟨*proof*⟩

lemma *wf_part_list_filter_minus*:
defines *notinP2 P1 f v1 part2*
  $\equiv$ *List.map_filter* $(\lambda(P2, v2).$ *if* $P2 - P1 \neq \{\}$ *then* $Some(P2 - P1, v2)$ *else None*$)$ *part2*
assumes *partition_on* X (*set* (*map fst* (($P1$, $v1$) # *part1*)))
  and *partition_on* X (*set* (*map fst part2*))
shows *partition_on* $(X - P1)$ (*set* (*map fst* (*notinP2 P1 f v1 part2*)))
  and *distinct* (*map fst* (($P1$, $v1$) # *part1*)) $\Longrightarrow$ *distinct* (*map fst* (*part2*)) $\Longrightarrow$
    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*) $\Longrightarrow$ *partition_on* Y (*set ys*) $\Longrightarrow$ $X \cap Y = \{\}$ $\Longrightarrow$
partition_on $(X \cup Y)$ (*set* (*xs* @ *ys*))
⟨*proof*⟩

lemma *wf_part_list_merge_part2_raw*:
partition_on X (*set* (*map fst part1*)) $\wedge$ *distinct* (*map fst part1*) $\Longrightarrow$
 partition_on X (*set* (*map fst part2*)) $\wedge$ *distinct* (*map fst part2*) $\Longrightarrow$
 partition_on X (*set* (*map fst* (*merge_part2_raw f part1 part2*)))
  $\wedge$ *distinct* (*map fst* (*merge_part2_raw f part1 part2*))
⟨*proof*⟩

lemma *wf_part_list_merge_part3_raw*:
partition_on X (*set* (*map fst part1*)) $\wedge$ *distinct* (*map fst part1*) $\Longrightarrow$
 partition_on X (*set* (*map fst part2*)) $\wedge$ *distinct* (*map fst part2*) $\Longrightarrow$
 partition_on X (*set* (*map fst part3*)) $\wedge$ *distinct* (*map fst part3*) $\Longrightarrow$
 partition_on X (*set* (*map fst* (*merge_part3_raw f part1 part2 part3*)))
  $\wedge$ *distinct* (*map fst* (*merge_part3_raw f part1 part2 part3*))
⟨*proof*⟩

lift_definition *merge_part2* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('d, 'a)$ *part* $\Rightarrow ('d, 'a)$ *part* $\Rightarrow ('d, 'a)$ *part* **is** *merge_part2_raw*
⟨*proof*⟩

lift_definition *merge_part3* :: $('a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('d, 'a)$ *part* $\Rightarrow ('d, 'a)$ *part* $\Rightarrow ('d, 'a)$ *part* $\Rightarrow$ $('d, 'a)$ *part* **is** *merge_part3_raw*
⟨*proof*⟩

definition *proof_app* :: $('n, 'd)$ *proof* $\Rightarrow ('n, 'd)$ *proof* $\Rightarrow ('n, 'd)$ *proof* (**infixl** ‹⊕› *65*) **where**
$p \oplus q$ = (*case* (p, q) *of*
  (*Inl* (*SHistorically i li sps*), *Inl q*) $\Rightarrow$ *Inl* (*SHistorically* $(i+1)$ *li* (*sps* @ $[q]$))
| (*Inl* (*SAlways i hi sps*), *Inl q*) $\Rightarrow$ *Inl* (*SAlways* $(i-1)$ *hi* (q # *sps*))
| (*Inl* (*SSince sp2 sp1s*), *Inl q*) $\Rightarrow$ *Inl* (*SSince sp2* (*sp1s* @ $[q]$))

```
| (Inl (SUntil sp1s sp2), Inl q) ⇒ Inl (SUntil (q # sp1s) sp2)
| (Inr (VSince i vp1 vp2s), Inr q) ⇒ Inr (VSince (i+1) vp1 (vp2s @ [q]))
| (Inr (VOnce i li vps), Inr q) ⇒ Inr (VOnce (i+1) li (vps @ [q]))
| (Inr (VEventually i hi vps), Inr q) ⇒ Inr (VEventually (i−1) hi (q # vps))
| (Inr (VSinceInf i li vp2s), Inr q) ⇒ Inr (VSinceInf (i+1) li (vp2s @ [q]))
| (Inr (VUntil i vp2s vp1), Inr q) ⇒ Inr (VUntil (i−1) (q # vp2s) vp1)
| (Inr (VUntilInf i hi vp2s), Inr q) ⇒ Inr (VUntilInf (i−1) hi (q # vp2s)))

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

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

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

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

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

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

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

| (Inr vp1, Inr vp2) ⇒ [Inl (SIffVV vp1 vp2)])

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

```
  do_exists x p_part = (case p_part of
  Inl p ⇒ (case p of
    Inl sp ⇒ [Inl (SExists x default sp)]
  | Inr vp ⇒ [Inr (VExists x (trivial_part vp))])
| Inr part ⇒ (if (∃x∈Vals part. isl x) then
                map (λ(D,p). map_sum (SExists x (Min D)) id p) (filter (λ(_, p). isl p) (subsvals part))
              else
                [Inr (VExists x (map_part projr part))]))
```

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

```
  do_forall x p_part = (case p_part of
  Inl p ⇒ (case p of
    Inl sp ⇒ [Inl (SForall x (trivial_part sp))]
  | Inr vp ⇒ [Inr (VForall x default vp)])
| Inr part ⇒ (if (∀x∈Vals part. isl x) then
                [Inl (SForall x (map_part projl part))]
              else
               map (λ(D,p). map_sum id (VForall x (Min D)) p) (filter (λ(_, p). ¬isl p) (subsvals part))))
```

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

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

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

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

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

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

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

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

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

```
  do_eventually_base i a p' = (case (p', a = 0) of
  (Inl sp', True) ⇒ [Inl (SEventually i sp')]
```

| (Inr vp′, True) ⇒ [Inr (VEventually i i [vp′])]
| (_ , False) ⇒ [Inr (VEventually i i [])])

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

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

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

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

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

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

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

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

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

definition do_until :: nat ⇒ nat ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof ⇒ ('n, 'd) proof list **where**
do_until i a p1 p2 p' = (case (p1, p2, a = 0, p') of
(Inl sp1, Inr _ , True, Inl (SUntil _ _)) ⇒ [p' ⊕ (Inl sp1)]
| (Inl sp1, _ , False, Inl (SUntil _ _)) ⇒ [p' ⊕ (Inl sp1)]
| (Inl sp1, Inl sp2, True, Inl (SUntil _ _)) ⇒ [p' ⊕ (Inl sp1), Inl (SUntil [] sp2)]
| (Inl _ , Inr vp2, True, Inr (VUntilInf _ _ _)) ⇒ [p' ⊕ (Inr vp2)]
| (Inl _ , _ , False, Inr (VUntilInf _ hi vp2s')) ⇒ [Inr (VUntilInf i hi vp2s')]
| (Inl _ , Inr vp2, True, Inr (VUntil _ _ _)) ⇒ [p' ⊕ (Inr vp2)]
| (Inl _ , _ , False, Inr (VUntil _ vp2s' vp1')) ⇒ [Inr (VUntil i vp2s' vp1')]
| (Inr vp1, Inr vp2, True, Inl (SUntil _ _)) ⇒ [Inr (VUntil i [vp2] vp1)]
| (Inr vp1, _ , False, Inl (SUntil _ _)) ⇒ [Inr (VUntil i [] vp1)]
| (Inr vp1, Inl sp2, True, Inl (SUntil _ _)) ⇒ [Inl (SUntil [] sp2)]
| (Inr vp1, Inr vp2, True, Inr (VUntilInf _ _ _)) ⇒ [Inr (VUntil i [vp2] vp1), p' ⊕ (Inr vp2)]
| (Inr vp1, _ , False, Inr (VUntilInf _ hi vp2s')) ⇒ [Inr (VUntil i [] vp1), Inr (VUntilInf i hi vp2s')]
| (_ , Inl sp2, True, Inr (VUntilInf _ hi vp2s')) ⇒ [Inl (SUntil [] sp2)]
| (Inr vp1, Inr vp2, True, Inr (VUntil _ _ _)) ⇒ [Inr (VUntil i [vp2] vp1), p' ⊕ (Inr vp2)]
| (Inr vp1, _ , False, Inr (VUntil _ vp2s' vp1')) ⇒ [Inr (VUntil i [] vp1), Inr (VUntil i vp2s' vp1')]
| (_ , Inl sp2, True, Inr (VUntil _ _ _)) ⇒ [Inl (SUntil [] sp2)])

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

fun pdt_of :: nat ⇒ 'n ⇒ ('n, 'd :: linorder) Formula.trm list ⇒ 'n list ⇒ ('n ⇀ 'd) list ⇒ ('n, 'd) expl **where**

```
  pdt_of i r ts [] V = (if List.null V then Leaf (Inr (VPred i r ts)) else Leaf (Inl (SPred i r ts)))
| pdt_of i r ts (x # vs) V =
    (let ds = remdups (List.map_filter (λv. v x) V);
         part = tabulate ds (λd. pdt_of i r ts vs (filter (λv. v x = Some d) V)) (pdt_of i r ts vs [])
    in Node x part)
```

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

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

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

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

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

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

function (sequential) eval :: 'n list ⇒ nat ⇒ ('n, 'd) Formula.formula ⇒ ('n, 'd) expl where
  eval vs i Formula.TT = Leaf (Inl (STT i))
| eval vs i Formula.FF = Leaf (Inr (VFF i))
| eval vs i (Eq_Const x c) = Node x (tabulate [c] (λc. Leaf (Inl (SEq_Const i x c))) (Leaf (Inr
(VEq_Const i x c))))
| eval vs i (Formula.Pred r ts) =
  (pdt_of i r ts (filter (λx. x ∈ Formula.fv (Formula.Pred r ts)) vs) (List.map_filter (match ts) (sorted_list_of_set
(snd ‘ {rd ∈ Γ σ i. fst rd = r}))))
| eval vs i (Formula.Neg φ) = apply_pdt1 vs (λp. min_list_wrt cmp (do_neg p)) (eval vs i φ)
| eval vs i (Formula.Or φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_or p1 p2)) (eval vs i φ)
(eval vs i ψ)
| eval vs i (Formula.And φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_and p1 p2)) (eval vs i
φ) (eval vs i ψ)
| eval vs i (Formula.Imp φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_imp p1 p2)) (eval vs i
```

```
φ) (eval vs i ψ)
| eval vs i (Formula.Iff φ ψ) = apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_iff p1 p2)) (eval vs i φ)
(eval vs i ψ)
| eval vs i (Formula.Exists x φ) = hide_pdt (vs @ [x]) (λp. min_list_wrt cmp (do_exists x p)) (eval (vs
@ [x]) i φ)
| eval vs i (Formula.Forall x φ) = hide_pdt (vs @ [x]) (λp. min_list_wrt cmp (do_forall x p)) (eval (vs
@ [x]) i φ)
| eval vs i (Formula.Prev I φ) = (if i = 0 then Leaf (Inr VPrevZ)
                                   else apply_pdt1 vs (λp. min_list_wrt cmp (do_prev i I (Δ σ i) p)) (eval vs
(i−1) φ))
| eval vs i (Formula.Next I φ) = apply_pdt1 vs (λl. min_list_wrt cmp (do_next i I (Δ σ (i+1)) l)) (eval
vs (i+1) φ)
| eval vs i (Formula.Once I φ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inr (VOnceOut i))
   else (let expl = eval vs i φ in
        (if i = 0 then
           apply_pdt1 vs (λp. min_list_wrt cmp (do_once_base 0 0 p)) expl
         else (if right I ≥ enat (Δ σ i) then
                apply_pdt2 vs (λp p'. min_list_wrt cmp (do_once i (left I) p p')) expl
                           (eval vs (i−1) (Formula.Once (subtract (Δ σ i) I) φ))
              else apply_pdt1 vs (λp. min_list_wrt cmp (do_once_base i (left I) p)) expl))))
| eval vs i (Formula.Historically I φ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inl (SHistoricallyOut i))
   else (let expl = eval vs i φ in
        (if i = 0 then
           apply_pdt1 vs (λp. min_list_wrt cmp (do_historically_base 0 0 p)) expl
         else (if right I ≥ enat (Δ σ i) then
                apply_pdt2 vs (λp p'. min_list_wrt cmp (do_historically i (left I) p p')) expl
                           (eval vs (i−1) (Formula.Historically (subtract (Δ σ i) I) φ))
              else apply_pdt1 vs (λp. min_list_wrt cmp (do_historically_base i (left I) p)) expl))))
| eval vs i (Formula.Eventually I φ) =
  (let expl = eval vs i φ in
  (if right I = ∞ then undefined
   else (if right I ≥ enat (Δ σ (i+1)) then
           apply_pdt2 vs (λp p'. min_list_wrt cmp (do_eventually i (left I) p p')) expl
                           (eval vs (i+1) (Formula.Eventually (subtract (Δ σ (i+1)) I) φ))
         else apply_pdt1 vs (λp. min_list_wrt cmp (do_eventually_base i (left I) p)) expl)))
| eval vs i (Formula.Always I φ) =
  (let expl = eval vs i φ in
  (if right I = ∞ then undefined
   else (if right I ≥ enat (Δ σ (i+1)) then
           apply_pdt2 vs (λp p'. min_list_wrt cmp (do_always i (left I) p p')) expl
                           (eval vs (i+1) (Formula.Always (subtract (Δ σ (i+1)) I) φ))
         else apply_pdt1 vs (λp. min_list_wrt cmp (do_always_base i (left I) p)) expl)))
| eval vs i (Formula.Since φ I ψ) =
  (if τ σ i < τ σ 0 + left I then Leaf (Inr (VSinceOut i))
   else (let expl1 = eval vs i φ in
         let expl2 = eval vs i ψ in
         (if i = 0 then
            apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_since_base 0 0 p1 p2)) expl1 expl2
          else (if right I ≥ enat (Δ σ i) then
                 apply_pdt3 vs (λp1 p2 p'. min_list_wrt cmp (do_since i (left I) p1 p2 p')) expl1 expl2
                            (eval vs (i−1) (Formula.Since φ (subtract (Δ σ i) I) ψ))
                else apply_pdt2 vs (λp1 p2. min_list_wrt cmp (do_since_base i (left I) p1 p2)) expl1
expl2))))
| eval vs i (Formula.Until φ I ψ) =
  (let expl1 = eval vs i φ in
   let expl2 = eval vs i ψ in
```

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

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

lemma i_less_LTP: $\tau\ \sigma\ (Suc\ i) \leq b + \tau\ \sigma\ i \Longrightarrow i < LTP\ \sigma\ (b + \tau\ \sigma\ i)$
$\langle proof\rangle$

termination *eval*
$\langle proof\rangle$

end

end

13 Examples

definition $monitor :: (('n :: linorder \times 'd :: \{default, linorder\}\ list)\ set \times nat)\ list \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ expl\ list$ **where**
$monitor\ \pi\ \varphi = map\ (\lambda i.\ eval\ (trace\_of\_list\ \pi)\ (\lambda p\ q.\ size\ p \leq size\ q)\ (sorted\_list\_of\_set\ (fv\ \varphi))\ i\ \varphi)\ [0\ ..{<}\ length\ \pi]$
definition $check :: (('n :: linorder \times 'd :: \{default, linorder\}\ list)\ set \times nat)\ list \Rightarrow ('n, 'd)\ formula \Rightarrow bool$ **where**
$check\ \pi\ \varphi = list\_all\ (check\_all\ (trace\_of\_list\ \pi)\ \varphi)\ (monitor\ \pi\ \varphi)$

13.1 Infinite Domain

definition $prefix :: ((string \times string\ list)\ set \times nat)\ list$ **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⟩
```

13.2 Finite Domain

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

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

instantiation Domain :: default begin
definition default_Domain = Default
instance ⟨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.