A Verified Proof Checker for Metric First-Order Temporal Logic

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

June 20, 2024

Abstract

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

Contents

1 Traces and Trace Prefixes

1.1 Infinite Traces

coinductive *ssorted* :: $'a$ :: *linorder stream* ⇒ *bool* **where**
shd s ≤ *shd* (*stl s*) ⟹ *ssorted* (*stl s*) ⟹ *ssorted s*

lemma *ssorted_siterate*[*simp*]: (⋀*n*. *n* ≤ *f n*) ⟹ *ssorted* (*siterate f n*)
⟨*proof*⟩

lemma *ssortedD*: *ssorted s* ⟹ *s* !! *i* ≤ *stl s* !! *i*
⟨*proof*⟩

lemma *ssorted_sdrop*: *ssorted s* ⟹ *ssorted* (*sdrop i s*)
⟨*proof*⟩

lemma *ssorted_monoD*: *ssorted s* ⟹ *i* ≤ *j* ⟹ *s* !! *i* ≤ *s* !! *j*
⟨*proof*⟩

lemma *sorted_stake*: *ssorted s* ⟹ *sorted* (*stake i s*)
⟨*proof*⟩

lemma *ssorted_monoI*: ∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j* ⟹ *ssorted s*
⟨*proof*⟩

lemma *ssorted_iff_mono*: *ssorted s* ⟷ (∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j*)
⟨*proof*⟩

lemma *ssorted_iff_le_Suc*: *ssorted s* ⟷ (∀ *i*. *s* !! *i* ≤ *s* !! *Suc i*)
⟨*proof*⟩

definition *sincreasing s* = (∀ *x*. ∃ *i*. *x* < *s* !! *i*)

lemma *sincreasingI*: (⋀*x*. ∃ *i*. *x* < *s* !! *i*) ⟹ *sincreasing s*
⟨*proof*⟩

lemma *sincreasing_grD*:
fixes $x :: 'a :: semilattice\_sup$
assumes *sincreasing s*
shows $\exists j{>}i.\ x < s\ !!\ j$
⟨*proof*⟩

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

lemma *sincreasing_stl*: *sincreasing s* $\Longrightarrow$ *sincreasing* (*stl s*) **for** $s :: 'a :: semilattice\_sup\ stream$
⟨*proof*⟩

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

lemma *sfiniteI*: $(\bigwedge i.\ finite\ (s\ !!\ i)) \Longrightarrow sfinite\ s$
⟨*proof*⟩

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

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: 'a\ trace \Rightarrow nat \Rightarrow 'a\ set$ **is**
$\lambda s\ i.\ fst\ (s\ !!\ i)$ ⟨*proof*⟩
lift_definition $\tau :: 'a\ trace \Rightarrow nat \Rightarrow nat$ **is**
$\lambda s\ i.\ snd\ (s\ !!\ i)$ ⟨*proof*⟩

lemma *stream_eq_iff*: $s = s' \longleftrightarrow (\forall n.\ s\ !!\ n = s'\ !!\ n)$
⟨*proof*⟩

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

lemma τ*_mono*[*simp*]: $i \le j \Longrightarrow \tau\ s\ i \le \tau\ s\ j$
⟨*proof*⟩

lemma *ex_le_*τ: $\exists j{\ge}i.\ x \le \tau\ s\ j$
⟨*proof*⟩

lemma *le_*τ*_less*: $\tau\ \sigma\ i \le \tau\ \sigma\ j \Longrightarrow j < i \Longrightarrow \tau\ \sigma\ i = \tau\ \sigma\ j$
⟨*proof*⟩

lemma *less_*τ*D*: $\tau\ \sigma\ i < \tau\ \sigma\ j \Longrightarrow i < j$
⟨*proof*⟩

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

1.2 Finite Trace Prefixes

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

setup_lifting *type_definition_prefix*

lift_definition *pmap_Γ* :: (*'a set* ⇒ *'b set*) ⇒ *'a prefix* ⇒ *'b prefix* **is**
λ*f*. *map* (λ(*x*, *i*). (*f x*, *i*))
⟨*proof*⟩

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

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

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

lift_definition *plen* :: *'a prefix* ⇒ *nat* **is** *length* ⟨*proof*⟩

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

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

lift_definition *less_eq_prefix* :: *'a prefix* ⇒ *'a prefix* ⇒ *bool* **is**
λ*p q*. ∃ *r*. *q* = *p* @ *r* ⟨*proof*⟩

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

instance
⟨*proof*⟩

end

lemma *psnoc_inject*[*simp*]:
last_ts p ≤ *snd x* ⟹ *last_ts q* ≤ *snd y* ⟹ *psnoc p x* = *psnoc q y* ⟷ (*p* = *q* ∧ *x* = *y*)
⟨*proof*⟩

lift_definition *prefix_of* :: *'a prefix* ⇒ *'a trace* ⇒ *bool* **is** λ*p s*. *stake* (*length p*) *s* = *p* ⟨*proof*⟩

lemma *prefix_of_pnil*[*simp*]: *prefix_of pnil σ*
⟨*proof*⟩

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

lemma *plen_mono*: *π* ≤ *π'* ⟹ *plen π* ≤ *plen π'*
⟨*proof*⟩

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc p x*) *s* ⟹ *last_ts p* ≤ *snd x* ⟹
(*prefix_of p s* ⟹ *Γ s* (*plen p*) = *fst x* ⟹ *τ s* (*plen p*) = *snd x* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *le_pnil*[*simp*]: *pnil* ≤ *π*
⟨*proof*⟩

lift_definition *take_prefix* :: *nat* ⇒ *'a trace* ⇒ *'a prefix* **is** *stake*
⟨*proof*⟩

lemma *plen_take_prefix*[*simp*]: *plen* (*take_prefix i σ*) = *i*
⟨*proof*⟩

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

lemma *prefix_of_take_prefix*[*simp*]: *prefix_of* (*take_prefix* i σ) σ
⟨*proof*⟩

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

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

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

lemma *prefix_of_imp_linear*: *prefix_of* π $\sigma \Longrightarrow$ *prefix_of* π' $\sigma \Longrightarrow \pi \leq \pi' \vee \pi' \leq \pi$
⟨*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* ∧ *ssorted* s ∧ (∀ x∈*set xs*. ∀ y∈*sset* s. $x \leq y$))
⟨*proof*⟩

lemma *sincreasing_shift*:
assumes *sincreasing* s
shows *sincreasing* (*xs* @− s)
⟨*proof*⟩

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

lemma *pts_pmap_*Γ[*simp*]: *pts* (*pmap_*Γ f π) = *pts* π
⟨*proof*⟩

1.3 Earliest and Latest Time-Points

definition *ETP*:: $'a$ *trace* ⇒ *nat* ⇒ *nat* **where**
ETP σ t = (*LEAST* i. τ σ $i \geq t$)

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

abbreviation δ σ i $j \equiv (\tau$ σ $i - \tau$ σ $j)$

abbreviation *ETP_p* σ i $b \equiv$ *ETP* σ ((τ σ i) − b)
abbreviation *LTP_p* σ i $I \equiv$ *min* i (*LTP* σ ((τ σ i) − *left* I))
abbreviation *ETP_f* σ i $I \equiv$ *max* i (*ETP* σ ((τ σ 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*) ⇒ *Some* (*max x y*) | _ ⇒ *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*⟩

2 Metric First-Order Temporal Logic

2.1 Syntax

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

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

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

datatype ('*n*, '*a*) *formula* =
TT (⊤)

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

primrec *fv* :: *('n, 'a) formula* $\Rightarrow$ *'n set* **where**
fv (r † ts) = $\bigcup$ *(fv_trm ‘ set ts)*
| *fv* $\top$ = {}
| *fv* $\bot$ = {}
| *fv* $(x \approx\!\!\!\approx c)$ = $\{x\}$
| *fv* $(\neg_F \varphi)$ = *fv* φ
| *fv* $(\varphi \vee_F \psi)$ = *fv* φ $\cup$ *fv* ψ
| *fv* $(\varphi \wedge_F \psi)$ = *fv* φ $\cup$ *fv* ψ
| *fv* $(\varphi \longrightarrow_F \psi)$ = *fv* φ $\cup$ *fv* ψ
| *fv* $(\varphi \longleftrightarrow_F \psi)$ = *fv* φ $\cup$ *fv* ψ
| *fv* $(\exists_F x.\ \varphi)$ = *fv* φ $-$ $\{x\}$
| *fv* $(\forall_F x.\ \varphi)$ = *fv* φ $-$ $\{x\}$
| *fv* (**Y** *I* φ) = *fv* φ
| *fv* (**X** *I* φ) = *fv* φ
| *fv* (**P** *I* φ) = *fv* φ
| *fv* (**H** *I* φ) = *fv* φ
| *fv* (**F** *I* φ) = *fv* φ
| *fv* (**G** *I* φ) = *fv* φ
| *fv* (φ **S** *I* ψ) = *fv* φ $\cup$ *fv* ψ
| *fv* (φ **U** *I* ψ) = *fv* φ $\cup$ *fv* ψ

primrec *consts* :: *('n, 'a) formula* $\Rightarrow$ *'a set* **where**
consts (r † 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* $(x \approx\!\!\!\approx c)$ = $\{c\}$
| *consts* $(\neg_F \varphi)$ = *consts* φ
| *consts* $(\varphi \vee_F \psi)$ = *consts* φ $\cup$ *consts* ψ
| *consts* $(\varphi \wedge_F \psi)$ = *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* (**Y** *I* φ) = *consts* φ
| *consts* (**X** *I* φ) = *consts* φ
| *consts* (**P** *I* φ) = *consts* φ
| *consts* (**H** *I* φ) = *consts* φ
| *consts* (**F** *I* φ) = *consts* φ

| *consts* (**G** *I* φ) = *consts* φ
| *consts* (φ **S** *I* ψ) = *consts* φ ∪ *consts* ψ
| *consts* (φ **U** *I* ψ) = *consts* φ ∪ *consts* ψ

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* ⇒ *nat* **where**
nfv φ = *card* (*fv* φ)

fun *future_bounded* :: ('*n*, '*a*) *formula* ⇒ *bool* **where**
future_bounded ⊤ = *True*
| *future_bounded* ⊥ = *True*
| *future_bounded* (_ † _) = *True*
| *future_bounded* (_ ≈ _) = *True*
| *future_bounded* ($\neg_F$ φ) = *future_bounded* φ
| *future_bounded* (φ $\vee_F$ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
| *future_bounded* (φ $\wedge_F$ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
| *future_bounded* (φ $\longrightarrow_F$ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
| *future_bounded* (φ $\longleftrightarrow_F$ ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
| *future_bounded* (∃ $_F$_. φ) = *future_bounded* φ
| *future_bounded* (∀ $_F$_. φ) = *future_bounded* φ
| *future_bounded* (**Y** *I* φ) = *future_bounded* φ
| *future_bounded* (**X** *I* φ) = *future_bounded* φ
| *future_bounded* (**P** *I* φ) = *future_bounded* φ
| *future_bounded* (**H** *I* φ) = *future_bounded* φ
| *future_bounded* (**F** *I* φ) = (*future_bounded* φ ∧ *right I* ≠ ∞)
| *future_bounded* (**G** *I* φ) = (*future_bounded* φ ∧ *right I* ≠ ∞)
| *future_bounded* (φ **S** *I* ψ) = (*future_bounded* φ ∧ *future_bounded* ψ)
| *future_bounded* (φ **U** *I* ψ) = (*future_bounded* φ ∧ *future_bounded* ψ ∧ *right I* ≠ ∞)

2.2 Semantics

primrec *eval_trm* :: ('*n*, '*a*) *env* ⇒ ('*n*, '*a*) *trm* ⇒ '*a*(_⟦_⟧ [*70*,*89*] *89*) **where**
eval_trm v (**v** *x*) = *v x*
| *eval_trm v* (**c** *x*) = *x*

lemma *eval_trm_fv_cong*: ∀ *x*∈*fv_trm t*. *v x* = *v*' *x* ⟹ *v*⟦*t*⟧ = *v*'⟦*t*⟧
⟨*proof*⟩

definition *eval_trms* :: ('*n*, '*a*) *env* ⇒ ('*n*, '*a*) *trm list* ⇒ '*a list* (_⟦_⟧ [*70*,*89*] *89*) **where**
eval_trms v ts = *map* (*eval_trm v*) *ts*

lemma *eval_trms_fv_cong*:
∀ *t*∈*set ts*. ∀ *x*∈*fv_trm t*. *v x* = *v*' *x* ⟹ *v*⟦*ts*⟧ = *v*'⟦*ts*⟧
⟨*proof*⟩

primrec *eval_trm_set* :: ('*n*, '*a*) *envset* ⇒ ('*n*, '*a*) *trm* ⇒ ('*n*, '*a*) *trm* × '*a set*(_⦃_⦄ [*70*,*89*] *89*) **where**
eval_trm_set vs (**v** *x*) = (**v** *x*, *vs x*)
| *eval_trm_set vs* (**c** *x*) = (**c** *x*, {*x*})

definition *eval_trms_set* :: ('n, 'a) *envset* ⇒ ('n, 'a) *trm list* ⇒ (('n, 'a) *trm* × 'a *set*) *list* (_⦃_⦄ [70,89] 89)
  where *eval_trms_set vs ts* = *map* (*eval_trm_set vs*) *ts*

lemma *eval_trms_set_Nil*: *vs*⦃[]⦄ = []
  ⟨*proof*⟩

lemma *eval_trms_set_Cons*:
  vs⦃(*t* # *ts*)⦄ = *vs*⦃*t*⦄ # *vs*⦃*ts*⦄
  ⟨*proof*⟩

primrec *sat* :: ('n, 'a) *trace* ⇒ ('n, 'a) *env* ⇒ *nat* ⇒ ('n, 'a) *formula* ⇒ *bool* (⟨_, _, _⟩ ⊨ _ [56, 56, 56, 56] 55) **where**
  ⟨σ, v, i⟩ ⊨ ⊤ = *True*
| ⟨σ, v, i⟩ ⊨ ⊥ = *False*
| ⟨σ, v, i⟩ ⊨ *r* † *ts* = ((*r*, *v*⟦*ts*⟧) ∈ Γ σ *i*)
| ⟨σ, v, i⟩ ⊨ *x* ≈ *c* = (*v x* = *c*)
| ⟨σ, v, i⟩ ⊨ ¬$_F$ φ = (¬ ⟨σ, v, i⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ φ ∨$_F$ ψ = (⟨σ, v, i⟩ ⊨ φ ∨ ⟨σ, v, i⟩ ⊨ ψ)
| ⟨σ, v, i⟩ ⊨ φ ∧$_F$ ψ = (⟨σ, v, i⟩ ⊨ φ ∧ ⟨σ, v, i⟩ ⊨ ψ)
| ⟨σ, v, i⟩ ⊨ φ ⟶$_F$ ψ = (⟨σ, v, i⟩ ⊨ φ ⟶ ⟨σ, v, i⟩ ⊨ ψ)
| ⟨σ, v, i⟩ ⊨ φ ⟷$_F$ ψ = (⟨σ, v, i⟩ ⊨ φ ⟷ ⟨σ, v, i⟩ ⊨ ψ)
| ⟨σ, v, i⟩ ⊨ ∃$_F$*x*. φ = (∃ *z*. ⟨σ, *v*(*x* := *z*), *i*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ ∀$_F$*x*. φ = (∀ *z*. ⟨σ, *v*(*x* := *z*), *i*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **Y** *I* φ = (*case i of* 0 ⇒ *False* | *Suc j* ⇒ *mem* (τ σ *i* − τ σ *j*) *I* ∧ ⟨σ, *v*, *j*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **X** *I* φ = (*mem* (τ σ (*Suc i*) − τ σ *i*) *I* ∧ ⟨σ, *v*, *Suc i*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **P** *I* φ = (∃*j*≤*i*. *mem* (τ σ *i* − τ σ *j*) *I* ∧ ⟨σ, *v*, *j*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **H** *I* φ = (∀*j*≤*i*. *mem* (τ σ *i* − τ σ *j*) *I* ⟶ ⟨σ, *v*, *j*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **F** *I* φ = (∃*j*≥*i*. *mem* (τ σ *j* − τ σ *i*) *I* ∧ ⟨σ, *v*, *j*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ **G** *I* φ = (∀*j*≥*i*. *mem* (τ σ *j* − τ σ *i*) *I* ⟶ ⟨σ, *v*, *j*⟩ ⊨ φ)
| ⟨σ, v, i⟩ ⊨ φ **S** *I* ψ = (∃*j*≤*i*. *mem* (τ σ *i* − τ σ *j*) *I* ∧ ⟨σ, *v*, *j*⟩ ⊨ ψ ∧ (∀*k*∈{*j*<..*i*}. ⟨σ, *v*, *k*⟩ ⊨ φ))
| ⟨σ, v, i⟩ ⊨ φ **U** *I* ψ = (∃*j*≥*i*. *mem* (τ σ *j* − τ σ *i*) *I* ∧ ⟨σ, *v*, *j*⟩ ⊨ ψ ∧ (∀*k*∈{*i*..<*j*}. ⟨σ, *v*, *k*⟩ ⊨ φ))

lemma *sat_fv_cong*: ∀*x*∈*fv* φ. *v x* = *v*′ *x* ⟹ ⟨σ, *v*, *i*⟩ ⊨ φ = ⟨σ, *v*′, *i*⟩ ⊨ φ
⟨*proof*⟩

lemma *sat_Until_rec*: ⟨σ, *v*, *i*⟩ ⊨ φ **U** *I* ψ ⟷
  (*mem* 0 *I* ∧ ⟨σ, *v*, *i*⟩ ⊨ ψ ∨
   Δ σ (*i* + 1) ≤ *right I* ∧ ⟨σ, *v*, *i*⟩ ⊨ φ ∧ ⟨σ, *v*, *i* + 1⟩ ⊨ φ **U** (*subtract* (Δ σ (*i* + 1)) *I*) ψ)
  (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *sat_Since_rec*: ⟨σ, *v*, *i*⟩ ⊨ φ **S** *I* ψ ⟷
  mem 0 *I* ∧ ⟨σ, *v*, *i*⟩ ⊨ ψ ∨
  (*i* > 0 ∧ Δ σ *i* ≤ *right I* ∧ ⟨σ, *v*, *i*⟩ ⊨ φ ∧ ⟨σ, *v*, *i* − 1⟩ ⊨ φ **S** (*subtract* (Δ σ *i*) *I*) ψ)
  (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *sat_Since_0*: ⟨σ, *v*, 0⟩ ⊨ φ **S** *I* ψ ⟷ *mem* 0 *I* ∧ ⟨σ, *v*, 0⟩ ⊨ ψ
  ⟨*proof*⟩

lemma *sat_Since_point*: ⟨σ, *v*, *i*⟩ ⊨ φ **S** *I* ψ ⟹
   (⋀*j*. *j* ≤ *i* ⟹ *mem* (τ σ *i* − τ σ *j*) *I* ⟹ ⟨σ, *v*, *i*⟩ ⊨ φ **S** (*point* (τ σ *i* − τ σ *j*)) ψ ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

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

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

lemma *sat_Historically_rec*: $\langle \sigma, v, i \rangle \models \mathbf{H}\ I\ \varphi \longleftrightarrow$
$(mem\ 0\ I \longrightarrow \langle \sigma, v, i \rangle \models \varphi) \wedge$
$(i > 0 \longrightarrow \Delta\ \sigma\ i \leq right\ I \longrightarrow \langle \sigma, v, i - 1 \rangle \models \mathbf{H}\ (subtract\ (\Delta\ \sigma\ i)\ I)\ \varphi)$
⟨*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_no_notation* **begin**

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

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

For subscripts type “backslash” followed by “sub”

no_notation *TT* ($\top$)
and *FF* ($\bot$)
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*)

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

end

bundle *MFOTL_notation* **begin**

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

notation *TT* (⊤)
  and *FF* (⊥)
  and *Pred* (_ † _ [*85*, *85*] *85*)
  and *Eq_Const* (_ ≈ _ [*85*, *85*] *85*)
  and *Neg* (¬$_F$ _ [*82*] *82*)
  and *And* (**infixr** ∧$_F$ *80*)
  and *Or* (**infixr** ∨$_F$ *80*)
  and *Imp* (**infixr** ⟶$_F$ *79*)
  and *Iff* (**infixr** ⟷$_F$ *79*)
  and *Exists* (∃$_F$_. _ [*70*,*70*] *70*)
  and *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 *MFOTL_no_notation*

3 Valued Partitions

lemma *part_list_set_eq_aux1*:
 assumes
  ∀ *i*<*length xs*. ∀ *j*<*length xs*. *i* ≠ *j* ⟶ *fst* (*xs* ! *i*) ∩ *fst* (*xs* ! *j*) = {}
  {} ∉ *fst* ‘ *set xs*
 shows *disjoint* (*fst* ‘ *set xs*) ∧ *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*⟩

3.1 *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*: $inj\ h \Longrightarrow size\_part\ f\ g \circ map\_part\ h = size\_part\ f\ (g \circ h)$
⟨*proof*⟩

⟨*ML*⟩

lemma *is_measure_size_part*[*measure_function*]: $is\_measure\ f \Longrightarrow is\_measure\ g \Longrightarrow is\_measure\ (size\_part$

f g)
⟨*proof*⟩

lemma *size_part_estimation*[*termination_simp*]: $x \in Vals\ xs \Longrightarrow y < g\ x \Longrightarrow y < size\_part\ f\ g\ xs$
⟨*proof*⟩

lemma *size_part_estimation'*[*termination_simp*]: $x \in Vals\ xs \Longrightarrow y \leq g\ x \Longrightarrow y \leq size\_part\ f\ g\ xs$
⟨*proof*⟩

lemma *size_part_pointwise*[*termination_simp*]: $(\bigwedge x.\ x \in Vals\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow size\_part\ h\ f\ xs \leq size\_part\ h\ g\ xs$
⟨*proof*⟩

3.2 Functions on Valued Partitions

lemma *Vals_code*[*code*]: $Vals\ x = set\ (map\ snd\ (Rep\_part\ x))$
⟨*proof*⟩

lemma *Vals_transfer*[*transfer_rule*]: $rel\_fun\ (pcr\_part\ (=)\ (=))\ (=)\ (set \circ map\ snd)\ Vals$
⟨*proof*⟩

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

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

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

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

4 Partitioned Decision Trees

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

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

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

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*: ∀ *x* ∈ *vars e*. *v x* = *v*′ *x* ⟹ *eval_pdt v e* = *eval_pdt v*′ *e*
⟨*proof*⟩

lemma *vars_order_vars*: *vars_order vs e* ⟹ *vars e* ⊆ *set vs*
⟨*proof*⟩

lemma *vars_order_distinct_paths*: *vars_order vs e* ⟹ *distinct vs* ⟹ *distinct_paths e*
⟨*proof*⟩

lemma *eval_pdt_fun_upd*: *vars_order vs e* ⟹ *x* ∉ *set vs* ⟹ *eval_pdt* (*v*(*x* := *d*)) *e* = *eval_pdt v e*
⟨*proof*⟩

5 Proof System

unbundle *MFOTL_notation*

context begin

inductive *SAT* **and** *VIO* :: ('*n*, '*d*) *trace* ⇒ ('*n*, '*d*) *env* ⇒ *nat* ⇒ ('*n*, '*d*) *formula* ⇒ *bool* **for** *σ* **where**
  STT: *SAT σ v i TT*
| *VFF*: *VIO σ v i FF*
| *SPred*: (*r*, *eval_trms v ts*) ∈ Γ *σ i* ⟹ *SAT σ v i* (*Pred r ts*)
| *VPred*: (*r*, *eval_trms v ts*) ∉ Γ *σ i* ⟹ *VIO σ v i* (*Pred r ts*)
| *SEq_Const*: *v x* = *c* ⟹ *SAT σ v i* (*Eq_Const x c*)
| *VEq_Const*: *v x* ≠ *c* ⟹ *VIO σ v i* (*Eq_Const x c*)
| *SNeg*: *VIO σ v i φ* ⟹ *SAT σ v i* (*Neg φ*)
| *VNeg*: *SAT σ v i φ* ⟹ *VIO σ v i* (*Neg φ*)
| *SOrL*: *SAT σ v i φ* ⟹ *SAT σ v i* (*Or φ ψ*)
| *SOrR*: *SAT σ v i ψ* ⟹ *SAT σ v i* (*Or φ ψ*)
| *VOr*: *VIO σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Or φ ψ*)
| *SAnd*: *SAT σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *SAT σ v i* (*And φ ψ*)
| *VAndL*: *VIO σ v i φ* ⟹ *VIO σ v i* (*And φ ψ*)
| *VAndR*: *VIO σ v i ψ* ⟹ *VIO σ v i* (*And φ ψ*)
| *SImpL*: *VIO σ v i φ* ⟹ *SAT σ v i* (*Imp φ ψ*)
| *SImpR*: *SAT σ v i ψ* ⟹ *SAT σ v i* (*Imp φ ψ*)
| *VImp*: *SAT σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Imp φ ψ*)
| *SIffSS*: *SAT σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *SAT σ v i* (*Iff φ ψ*)
| *SIffVV*: *VIO σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *SAT σ v i* (*Iff φ ψ*)
| *VIffSV*: *SAT σ v i φ* ⟹ *VIO σ v i ψ* ⟹ *VIO σ v i* (*Iff φ ψ*)
| *VIffVS*: *VIO σ v i φ* ⟹ *SAT σ v i ψ* ⟹ *VIO σ v i* (*Iff φ ψ*)
| *SExists*: ∃ *z*. *SAT σ* (*v* (*x* := *z*)) *i φ* ⟹ *SAT σ v i* (*Exists x φ*)
| *VExists*: ∀ *z*. *VIO σ* (*v* (*x* := *z*)) *i φ* ⟹ *VIO σ v i* (*Exists x φ*)
| *SForall*: ∀ *z*. *SAT σ* (*v* (*x* := *z*)) *i φ* ⟹ *SAT σ v i* (*Forall x φ*)
| *VForall*: ∃ *z*. *VIO σ* (*v* (*x* := *z*)) *i φ* ⟹ *VIO σ v i* (*Forall x φ*)
| *SPrev*: *i* > *0* ⟹ *mem* (Δ *σ i*) *I* ⟹ *SAT σ v* (*i*−*1*) *φ* ⟹ *SAT σ v i* (**Y** *I φ*)
| *VPrev*: *i* > *0* ⟹ *VIO σ v* (*i*−*1*) *φ* ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevZ*: *i* = *0* ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevOutL*: *i* > *0* ⟹ (Δ *σ i*) < (*left I*) ⟹ *VIO σ v i* (**Y** *I φ*)
| *VPrevOutR*: *i* > *0* ⟹ *enat* (Δ *σ i*) > (*right I*) ⟹ *VIO σ v i* (**Y** *I φ*)
| *SNext*: *mem* (Δ *σ* (*i*+*1*)) *I* ⟹ *SAT σ v* (*i*+*1*) *φ* ⟹ *SAT σ v i* (**X** *I φ*)
| *VNext*: *VIO σ v* (*i*+*1*) *φ* ⟹ *VIO σ v i* (**X** *I φ*)
| *VNextOutL*: (Δ *σ* (*i*+*1*)) < (*left I*) ⟹ *VIO σ v i* (**X** *I φ*)
| *VNextOutR*: *enat* (Δ *σ* (*i*+*1*)) > (*right I*) ⟹ *VIO σ v i* (**X** *I φ*)

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

5.1 Soundness and Completeness

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

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

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

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

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

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

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

end

unbundle *MFOTL_no_notation*

6 Proof Objects

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

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

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

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

fun s_at :: ('n, 'd) sproof ⇒ 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)
| v_at (VFF i) = i
| v_at (VPred i _ _) = i
| v_at (VEq_Const i _ _) = i
| v_at (VNeg sp) = s_at sp
| v_at (VOr vp1 _) = v_at vp1
| v_at (VAndL vp1) = v_at vp1
| v_at (VAndR vp2) = v_at vp2
| v_at (VImp sp1 _) = s_at sp1
| v_at (VIffSV sp1 _) = s_at sp1
| v_at (VIffVS vp1 _) = v_at vp1
| v_at (VExists _ part) = v_at (part_hd part)
| v_at (VForall _ _ vp1) = v_at vp1
| v_at (VPrev vp) = v_at vp + 1
| v_at (VPrevZ) = 0
```

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

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

7 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 ≠ {} ⟹ *infinite X* ⟹ *infinite* (*set_Cons X XS*)
X ≠ {} ⟹ *infinite XS* ⟹ *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* ∉ *set xs*
⟨*proof*⟩

lemma *nth_fst_pos*: *x* ∈ *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* ∉ *set xs*

⟨*proof*⟩

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

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

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

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

lemma *fst_pos_in_positions*: $x \in set\ xs \Longrightarrow the\ (fst\_pos\ xs\ x) \in set\ (positions\ xs\ x)$
⟨*proof*⟩

lemma *hd_positions_eq_fst_pos*: $x \in set\ xs \Longrightarrow hd\ (positions\ xs\ x) = the\ (fst\_pos\ xs\ x)$
⟨*proof*⟩

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

lemma *Min_sorted_list*: $sorted\ xs \Longrightarrow xs \neq [] \Longrightarrow Min\ (set\ xs) = hd\ xs$
⟨*proof*⟩

lemma *Min_positions*: $x \in set\ xs \Longrightarrow Min\ (set\ (positions\ xs\ x)) = the\ (fst\_pos\ xs\ x)$
⟨*proof*⟩

lemma *subset_positions_map_fst*: $set\ (positions\ tXs\ tX) \subseteq set\ (positions\ (map\ fst\ tXs)\ (fst\ tX))$
⟨*proof*⟩

lemma *subset_positions_map_snd*: $set\ (positions\ tXs\ tX) \subseteq set\ (positions\ (map\ snd\ tXs)\ (snd\ tX))$
⟨*proof*⟩

lemma *Max_eqI*: $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow a \leq b) \Longrightarrow \exists\, a{\in}A.\ b \leq a \Longrightarrow Max\ A = b$
⟨*proof*⟩

lemma *Max_Suc*: $X \neq \{\} \Longrightarrow finite\ X \Longrightarrow Max\ (Suc\ `\ X) = Suc\ (Max\ X)$
⟨*proof*⟩

lemma *Max_insert0*: $X \neq \{\} \Longrightarrow finite\ X \Longrightarrow Max\ (insert\ (0{::}nat)\ X) = Max\ X$
⟨*proof*⟩

lemma *positions_Cons_notin_tail*: $x \notin set\ xs \Longrightarrow positions\ (x\ \#\ xs)\ x = [0{::}nat]$
⟨*proof*⟩

lemma *Max_set_positions_Cons_hd*:
$x \notin set\ xs \Longrightarrow Max\ (set\ (positions\ (x\ \#\ xs)\ x)) = 0$
⟨*proof*⟩

lemma *Max_set_positions_Cons_tl*:
$y \in set\ xs \Longrightarrow Max\ (set\ (positions\ (x\ \#\ xs)\ y)) = Suc\ (Max\ (set\ (positions\ xs\ y)))$
⟨*proof*⟩

lemma *max_aux*: $finite\ X \Longrightarrow Suc\ j \in X \Longrightarrow Max\ (insert\ (Suc\ j)\ (X - \{j\})) = Max\ (insert\ j\ X)$
⟨*proof*⟩

lemma *ball_swap*: $(\forall\, x \in A.\ \forall\, y \in B.\ P\ x\ y) = (\forall\, y \in B.\ \forall\, x \in A.\ P\ x\ y)$
⟨*proof*⟩

lemma *ball_triv_nonempty*: $A \neq \{\} \Longrightarrow (\forall\, x \in A.\ P) = P$
⟨*proof*⟩

8 Proof Checker

unbundle *MFOTL_notation*

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

begin

fun *s_check* :: $('n, 'd)\ env \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ sproof \Rightarrow bool$
and *v_check* :: $('n, 'd)\ env \Rightarrow ('n, 'd)\ formula \Rightarrow ('n, 'd)\ vproof \Rightarrow bool$ **where**
s_check v f p = (*case* (*f*, *p*) *of*
(⊤, *STT i*) ⇒ *True*
| (*r* † *ts*, *SPred i s ts'*) ⇒
(*r* = *s* ∧ *ts* = *ts'* ∧ (*r*, *v*⟦*ts*⟧) ∈ Γ *σ i*)
| (*x* ≈ *c*, *SEq_Const i x' c'*) ⇒
(*c* = *c'* ∧ *x* = *x'* ∧ *v x* = *c*)
| (¬$_F$ *φ*, *SNeg vp*) ⇒ *v_check v φ vp*
| (*φ* ∨$_F$ *ψ*, *SOrL sp1*) ⇒ *s_check v φ sp1*
| (*φ* ∨$_F$ *ψ*, *SOrR sp2*) ⇒ *s_check v ψ sp2*
| (*φ* ∧$_F$ *ψ*, *SAnd sp1 sp2*) ⇒ *s_check v φ sp1* ∧ *s_check v ψ sp2* ∧ *s_at sp1* = *s_at sp2*
| (*φ* ⟶$_F$ *ψ*, *SImpL vp1*) ⇒ *v_check v φ vp1*
| (*φ* ⟶$_F$ *ψ*, *SImpR sp2*) ⇒ *s_check v ψ sp2*
| (*φ* ⟷$_F$ *ψ*, *SIffSS sp1 sp2*) ⇒ *s_check v φ sp1* ∧ *s_check v ψ sp2* ∧ *s_at sp1* = *s_at sp2*
| (*φ* ⟷$_F$ *ψ*, *SIffVV vp1 vp2*) ⇒ *v_check v φ vp1* ∧ *v_check v ψ vp2* ∧ *v_at vp1* = *v_at vp2*
| (∃$_F$*x*. *φ*, *SExists y val sp*) ⇒ (*x* = *y* ∧ *s_check* (*v* (*x* := *val*)) *φ sp*)
| (∀$_F$*x*. *φ*, *SForall y sp_part*) ⇒ (*let i* = *s_at* (*part_hd sp_part*)
in x = *y* ∧ (∀ (*sub*, *sp*) ∈ *SubsVals sp_part*. *s_at sp* = *i* ∧ (∀ *z* ∈ *sub*. *s_check* (*v* (*x* := *z*)) *φ sp*)))
| (**Y** *I φ*, *SPrev sp*) ⇒
(*let j* = *s_at sp*; *i* = *s_at* (*SPrev sp*) *in*
i = *j*+*1* ∧ *mem* (Δ *σ i*) *I* ∧ *s_check v φ sp*)
| (**X** *I φ*, *SNext sp*) ⇒
(*let j* = *s_at sp*; *i* = *s_at* (*SNext sp*) *in*
j = *i*+*1* ∧ *mem* (Δ *σ j*) *I* ∧ *s_check v 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 v 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 v 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 v 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 v φ sp*))

$$
\begin{array}{l}
\quad \mid (\varphi\ \mathbf{S}\ I\ \psi,\ \mathit{SSince}\ \mathit{sp2}\ \mathit{sp1s}) \Rightarrow \\
\quad\ \ (\mathit{let}\ i = \mathit{s\_at}\ (\mathit{SSince}\ \mathit{sp2}\ \mathit{sp1s});\ j = \mathit{s\_at}\ \mathit{sp2}\ \mathit{in} \\
\quad\ \ j \leq i \wedge \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \\
\quad\ \ \wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sp1s} = [j{+}1\ ..{<}\ i{+}1] \\
\quad\ \ \wedge\ \mathit{s\_check}\ v\ \psi\ \mathit{sp2} \\
\quad\ \ \wedge\ (\forall\, \mathit{sp1} \in \mathit{set}\ \mathit{sp1s}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp1})) \\
\quad \mid (\varphi\ \mathbf{U}\ I\ \psi,\ \mathit{SUntil}\ \mathit{sp1s}\ \mathit{sp2}) \Rightarrow \\
\quad\ \ (\mathit{let}\ i = \mathit{s\_at}\ (\mathit{SUntil}\ \mathit{sp1s}\ \mathit{sp2});\ j = \mathit{s\_at}\ \mathit{sp2}\ \mathit{in} \\
\quad\ \ j \geq i \wedge \mathit{mem}\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \\
\quad\ \ \wedge\ \mathit{map}\ \mathit{s\_at}\ \mathit{sp1s} = [i\ ..{<}\ j] \wedge \mathit{s\_check}\ v\ \psi\ \mathit{sp2} \\
\quad\ \ \wedge\ (\forall\, \mathit{sp1} \in \mathit{set}\ \mathit{sp1s}.\ \mathit{s\_check}\ v\ \varphi\ \mathit{sp1})) \\
\quad \mid (\_\ ,\ \_) \Rightarrow \mathit{False}) \\
\mid \mathit{v\_check}\ v\ f\ p = (\mathit{case}\ (f,\ p)\ \mathit{of} \\
\quad\ \ (\bot,\ \mathit{VFF}\ i) \Rightarrow \mathit{True} \\
\quad \mid (r \dagger \mathit{ts},\ \mathit{VPred}\ i\ \mathit{pred}\ \mathit{ts}') \Rightarrow \\
\quad\ \ (r = \mathit{pred} \wedge \mathit{ts} = \mathit{ts}' \wedge (r,\ v[\![\mathit{ts}]\!]) \notin \Gamma\ \sigma\ i) \\
\quad \mid (x \approx c,\ \mathit{VEq\_Const}\ i\ x'\ c') \Rightarrow \\
\quad\ \ (c = c' \wedge x = x' \wedge v\ x \neq c) \\
\quad \mid (\neg_F\ \varphi,\ \mathit{VNeg}\ \mathit{sp}) \Rightarrow \mathit{s\_check}\ v\ \varphi\ \mathit{sp} \\
\quad \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} \\
\quad \mid (\varphi \wedge_F \psi,\ \mathit{VAndL}\ \mathit{vp1}) \Rightarrow \mathit{v\_check}\ v\ \varphi\ \mathit{vp1} \\
\quad \mid (\varphi \wedge_F \psi,\ \mathit{VAndR}\ \mathit{vp2}) \Rightarrow \mathit{v\_check}\ v\ \psi\ \mathit{vp2} \\
\quad \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} \\
\quad \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} \\
\quad \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} \\
\quad \mid (\exists_F x.\ \varphi,\ \mathit{VExists}\ y\ \mathit{vp\_part}) \Rightarrow (\mathit{let}\ i = \mathit{v\_at}\ (\mathit{part\_hd}\ \mathit{vp\_part}) \\
\quad\quad \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}))) \\
\quad \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}) \\
\quad \mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrev}\ \mathit{vp}) \Rightarrow \\
\quad\ \ (\mathit{let}\ j = \mathit{v\_at}\ \mathit{vp};\ i = \mathit{v\_at}\ (\mathit{VPrev}\ \mathit{vp})\ \mathit{in} \\
\quad\ \ i = j{+}1 \wedge \mathit{v\_check}\ v\ \varphi\ \mathit{vp}) \\
\quad \mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevZ}) \Rightarrow \mathit{True} \\
\quad \mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevOutL}\ i) \Rightarrow \\
\quad\ \ i > 0 \wedge \Delta\ \sigma\ i < \mathit{left}\ I \\
\quad \mid (\mathbf{Y}\ I\ \varphi,\ \mathit{VPrevOutR}\ i) \Rightarrow \\
\quad\ \ i > 0 \wedge \mathit{enat}\ (\Delta\ \sigma\ i) > \mathit{right}\ I \\
\quad \mid (\mathbf{X}\ I\ \varphi,\ \mathit{VNext}\ \mathit{vp}) \Rightarrow \\
\quad\ \ (\mathit{let}\ j = \mathit{v\_at}\ \mathit{vp};\ i = \mathit{v\_at}\ (\mathit{VNext}\ \mathit{vp})\ \mathit{in} \\
\quad\ \ j = i{+}1 \wedge \mathit{v\_check}\ v\ \varphi\ \mathit{vp}) \\
\quad \mid (\mathbf{X}\ I\ \varphi,\ \mathit{VNextOutL}\ i) \Rightarrow \\
\quad\ \ \Delta\ \sigma\ (i{+}1) < \mathit{left}\ I \\
\quad \mid (\mathbf{X}\ I\ \varphi,\ \mathit{VNextOutR}\ i) \Rightarrow \\
\quad\ \ \mathit{enat}\ (\Delta\ \sigma\ (i{+}1)) > \mathit{right}\ I \\
\quad \mid (\mathbf{P}\ I\ \varphi,\ \mathit{VOnceOut}\ i) \Rightarrow \\
\quad\ \ \tau\ \sigma\ i < \tau\ \sigma\ 0 + \mathit{left}\ I \\
\quad \mid (\mathbf{P}\ I\ \varphi,\ \mathit{VOnce}\ i\ \mathit{li}\ \mathit{vps}) \Rightarrow \\
\quad\ \ (\mathit{li} = (\mathit{case}\ \mathit{right}\ I\ \mathit{of}\ \infty \Rightarrow 0 \mid \mathit{enat}\ b \Rightarrow \mathit{ETP\_p}\ \sigma\ i\ b) \\
\quad\ \ \wedge\ \tau\ \sigma\ 0 + \mathit{left}\ I \leq \tau\ \sigma\ i \\
\quad\ \ \wedge\ \mathit{map}\ \mathit{v\_at}\ \mathit{vps} = [\mathit{li}\ ..{<}\ (\mathit{LTP\_p}\ \sigma\ i\ I) + 1] \\
\quad\ \ \wedge\ (\forall\, \mathit{vp} \in \mathit{set}\ \mathit{vps}.\ \mathit{v\_check}\ v\ \varphi\ \mathit{vp})) \\
\quad \mid (\mathbf{F}\ I\ \varphi,\ \mathit{VEventually}\ i\ \mathit{hi}\ \mathit{vps}) \Rightarrow \\
\quad\ \ (\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 \\
\quad\ \ \wedge\ \mathit{map}\ \mathit{v\_at}\ \mathit{vps} = [(\mathit{ETP\_f}\ \sigma\ i\ I)\ ..{<}\ \mathit{hi} + 1] \\
\quad\ \ \wedge\ (\forall\, \mathit{vp} \in \mathit{set}\ \mathit{vps}.\ \mathit{v\_check}\ v\ \varphi\ \mathit{vp})) \\
\quad \mid (\mathbf{H}\ I\ \varphi,\ \mathit{VHistorically}\ i\ \mathit{vp}) \Rightarrow \\
\quad\ \ (\mathit{let}\ j = \mathit{v\_at}\ \mathit{vp}\ \mathit{in} \\
\quad\ \ j \leq i \wedge \mathit{mem}\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge \mathit{v\_check}\ v\ \varphi\ \mathit{vp}) \\
\quad \mid (\mathbf{G}\ I\ \varphi,\ \mathit{VAlways}\ i\ \mathit{vp}) \Rightarrow
\end{array}
$$

```
    (let j = v_at vp
    in j ≥ i ∧ mem (τ σ j − τ σ i) I ∧ v_check v φ vp)
  | (φ S I ψ, VSinceOut i) ⇒
    τ σ i < τ σ 0 + left I
  | (φ S I ψ, VSince i vp1 vp2s) ⇒
    (let j = v_at vp1 in
    (case right I of ∞ ⇒ True | enat b ⇒ ETP_p σ i b ≤ j) ∧ j ≤ i
    ∧ τ σ 0 + left I ≤ τ σ i
    ∧ map v_at vp2s = [j ..< (LTP_p σ i I) + 1] ∧ v_check v φ vp1
    ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
  | (φ S I ψ, VSinceInf i li vp2s) ⇒
    (li = (case right I of ∞ ⇒ 0 | enat b ⇒ ETP_p σ i b)
    ∧ τ σ 0 + left I ≤ τ σ i
    ∧ map v_at vp2s = [li ..< (LTP_p σ i I) + 1]
    ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
  | (φ U I ψ, VUntil i vp2s vp1) ⇒
    (let j = v_at vp1 in
    (case right I of ∞ ⇒ True | enat b ⇒ j < LTP_f σ i b) ∧ i ≤ j
    ∧ map v_at vp2s = [ETP_f σ i I ..< j + 1] ∧ v_check v φ vp1
    ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
  | (φ U I ψ, VUntilInf i hi vp2s) ⇒
    (hi = (case right I of enat b ⇒ LTP_f σ i b) ∧ right I ≠ ∞
    ∧ map v_at vp2s = [ETP_f σ i I ..< hi + 1]
    ∧ (∀ vp2 ∈ set vp2s. v_check v ψ vp2))
  | ( _ , _) ⇒ False)

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

8.1 Checker Soundness

```
lemma check_soundness:
  s_check v φ sp ⟹ SAT σ v (s_at sp) φ
  v_check v φ vp ⟹ VIO σ v (v_at vp) φ
⟨proof⟩

definition compatible X vs v ⟷ (∀ x∈X. v x ∈ vs x)

definition compatible_vals X vs = {v. ∀ x ∈ X. v x ∈ vs x}

lemma compatible_alt:
  compatible X vs v ⟷ v ∈ compatible_vals X vs
  ⟨proof⟩

lemma compatible_empty_iff: compatible {} vs v ⟷ True
  ⟨proof⟩

lemma compatible_vals_empty_eq: compatible_vals {} vs = UNIV
  ⟨proof⟩

lemma compatible_union_iff:
  compatible (X ∪ Y) vs v ⟷ compatible X vs v ∧ compatible Y vs v
  ⟨proof⟩

lemma compatible_vals_union_eq:
  compatible_vals (X ∪ Y) vs = compatible_vals X vs ∩ compatible_vals Y vs
  ⟨proof⟩
```

lemma *compatible_antimono*:
compatible X vs v $\Longrightarrow$ *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*:
(∀ *x*. *vs x* ≠ {}) $\Longrightarrow$ *compatible X vs v* $\Longrightarrow$ *X* $\subseteq$ *Y* $\Longrightarrow$ ∃ *v'*. *compatible Y vs v'* ∧ (∀ *x*∈*X*. *v x* = *v' x*)
⟨*proof*⟩

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

primrec *mk_values* :: (('*n*, '*d*) *trm* × '*a set*) *list* ⇒ '*a list set*
where *mk_values* [] = {[]}
| *mk_values* (*T* # *Ts*) = (*case T of*
(**v** *x*, *X*) ⇒
let terms = *map fst Ts in*
if **v** *x* ∈ *set terms then*
let fst_pos = *hd* (*positions terms* (**v** *x*)) *in* (λ*xs*. (*xs* ! *fst_pos*) # *xs*) ‘ (*mk_values Ts*)
else set_Cons X (*mk_values Ts*)
| (**c** *a*, *X*) ⇒ *set_Cons X* (*mk_values Ts*))

lemma *mk_values_nempty*:
{} ∉ *set* (*map snd tXs*) $\Longrightarrow$ *mk_values tXs* ≠ {}
⟨*proof*⟩

lemma *mk_values_not_Nil*:
{} ∉ *set* (*map snd tXs*) $\Longrightarrow$ *tXs* ≠ [] $\Longrightarrow$ *vs* ∈ *mk_values tXs* $\Longrightarrow$ *vs* ≠ []
⟨*proof*⟩

lemma *mk_values_nth_cong*: **v** *x* ∈ *set* (*map fst tXs*) $\Longrightarrow$
n ∈ *set* (*positions* (*map fst tXs*) (**v** *x*)) $\Longrightarrow$
m ∈ *set* (*positions* (*map fst tXs*) (**v** *x*)) $\Longrightarrow$
vs ∈ *mk_values tXs* $\Longrightarrow$
vs ! *n* = *vs* ! *m*
⟨*proof*⟩

definition *mk_values_subset p tXs X*
⟷ (*let* (*fintXs*, *inftXs*) = *partition* (λ*tX*. *finite* (*snd tX*)) *tXs in*
if inftXs = [] *then* {*p*} × *mk_values tXs* ⊆ *X*
else let inf_dups = *filter* (λ*tX*. (*fst tX*) ∈ *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*} × *mk_values tXs* ⊆ *X*))
else if list_all (λ*tX*. *Max* (*set* (*positions tXs tX*)) < *Max* (*set* (*positions* (*map fst tXs*) (*fst tX*)))) *inf_dups*
then {*p*} × *mk_values tXs* ⊆ *X*
else (*if finite X then False else Code.abort STR ''subset on infinite subset''* (λ_. {*p*} × *mk_values tXs* ⊆ *X*)))

lemma *mk_values_nemptyI*: ∀ *tX* ∈ *set tXs*. *snd tX* ≠ {} $\Longrightarrow$ *mk_values tXs* ≠ {}
⟨*proof*⟩

lemma *infinite_mk_values1*: ∀ *tX* ∈ *set tXs*. *snd tX* ≠ {} $\Longrightarrow$ *tY* ∈ *set tXs* $\Longrightarrow$
∀ *Y*. (*fst tY*, *Y*) ∈ *set tXs* ⟶ *infinite Y* $\Longrightarrow$ *infinite* (*mk_values tXs*)
⟨*proof*⟩

lemma *infinite_mk_values2*: $\forall\, tX{\in}set\ tXs.\ snd\ tX \neq \{\} \Longrightarrow$
$tY \in set\ tXs \Longrightarrow infinite\ (snd\ tY) \Longrightarrow$
$Max\ (set\ (positions\ tXs\ tY)) \geq Max\ (set\ (positions\ (map\ fst\ tXs)\ (fst\ tY))) \Longrightarrow$
$infinite\ (mk\_values\ tXs)$
⟨*proof*⟩

lemma *mk_values_subset_iff*: $\forall\, tX \in set\ tXs.\ snd\ tX \neq \{\} \Longrightarrow$
$mk\_values\_subset\ p\ tXs\ X \longleftrightarrow \{p\} \times mk\_values\ tXs \subseteq X$
⟨*proof*⟩

lemma *mk_values_sound*: $cs \in mk\_values\ (vs\{\!|ts|\!\})\Longrightarrow$
$\exists\, v{\in}compatible\_vals\ (fv\ (r \dagger ts))\ vs.\ cs = v[\![ts]\!]$
⟨*proof*⟩

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

lemma *mk_values_complete*: $cs = v[\![ts]\!] \Longrightarrow$
$v \in compatible\_vals\ (fv\ (r \dagger ts))\ vs \Longrightarrow$
$cs \in mk\_values\ (vs\{\!|ts|\!\})$
⟨*proof*⟩

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

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

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

8.2 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* $\Rightarrow$ $('n, 'd)$ *formula* $\Rightarrow$ $('n, 'd)$ *vproof* $\Rightarrow$ *bool* **where**
s_check_exec vs f p = (*case* (*f*, *p*) *of*
(⊤, *STT i*) ⇒ *True*
| (*r* † *ts*, *SPred i s ts'*) ⇒
(*r* = *s* ∧ *ts* = *ts'* ∧ *mk_values_subset r* (*vs*⦇*ts*⦈) (Γ *σ i*))
| (*x* ≈ *c*, *SEq_Const i x' c'*) ⇒
(*c* = *c'* ∧ *x* = *x'* ∧ *vs x* = {*c*})
| (¬$_F$ *φ*, *SNeg vp*) ⇒ *v_check_exec vs φ vp*
| (*φ* ∨$_F$ *ψ*, *SOrL sp1*) ⇒ *s_check_exec vs φ sp1*
| (*φ* ∨$_F$ *ψ*, *SOrR sp2*) ⇒ *s_check_exec vs ψ sp2*
| (*φ* ∧$_F$ *ψ*, *SAnd sp1 sp2*) ⇒ *s_check_exec vs φ sp1* ∧ *s_check_exec vs ψ sp2* ∧ *s_at sp1* = *s_at sp2*
| (*φ* ⟶$_F$ *ψ*, *SImpL vp1*) ⇒ *v_check_exec vs φ vp1*
| (*φ* ⟶$_F$ *ψ*, *SImpR sp2*) ⇒ *s_check_exec vs ψ sp2*
| (*φ* ⟷$_F$ *ψ*, *SIffSS sp1 sp2*) ⇒ *s_check_exec vs φ sp1* ∧ *s_check_exec vs ψ sp2* ∧ *s_at sp1* = *s_at sp2*
| (*φ* ⟷$_F$ *ψ*, *SIffVV vp1 vp2*) ⇒ *v_check_exec vs φ vp1* ∧ *v_check_exec vs ψ vp2* ∧ *v_at vp1* = *v_at vp2*
| (∃$_F$*x*. *φ*, *SExists y val sp*) ⇒ (*x* = *y* ∧ *s_check_exec* (*vs* (*x* := {*val*})) *φ sp*)
| (∀$_F$*x*. *φ*, *SForall y sp_part*) ⇒ (*let i* = *s_at* (*part_hd sp_part*)
in x = *y* ∧ (∀ (*sub*, *sp*) ∈ *SubsVals sp_part*. *s_at sp* = *i* ∧ *s_check_exec* (*vs* (*x* := *sub*)) *φ sp*))
| (**Y** *I φ*, *SPrev sp*) ⇒
(*let j* = *s_at sp*; *i* = *s_at* (*SPrev sp*) *in*
i = *j*+1 ∧ *mem* (Δ *σ i*) *I* ∧ *s_check_exec vs φ sp*)
| (**X** *I φ*, *SNext sp*) ⇒
(*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*))
| (_ , _) ⇒ *False*)
| *v_check_exec vs f p* = (*case* (*f*, *p*) *of*
(⊥, *VFF i*) ⇒ *True*

$$
\begin{aligned}
&\mid (r \dagger ts,\ VPred\ i\ pred\ ts') \Rightarrow \\
&\quad (r = pred \wedge ts = ts' \wedge mk\_values\_subset\_Compl\ r\ vs\ ts\ i) \\
&\mid (x \approx c,\ VEq\_Const\ i\ x'\ c') \Rightarrow \\
&\quad (c = c' \wedge x = x' \wedge c \notin vs\ x) \\
&\mid (\neg_F\ \varphi,\ VNeg\ sp) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp \\
&\mid (\varphi \vee_F \psi,\ VOr\ vp1\ vp2) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1 \wedge v\_check\_exec\ vs\ \psi\ vp2 \wedge v\_at\ vp1 = v\_at\ vp2 \\
&\mid (\varphi \wedge_F \psi,\ VAndL\ vp1) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1 \\
&\mid (\varphi \wedge_F \psi,\ VAndR\ vp2) \Rightarrow v\_check\_exec\ vs\ \psi\ vp2 \\
&\mid (\varphi \longrightarrow_F \psi,\ VImp\ sp1\ vp2) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1 \wedge v\_check\_exec\ vs\ \psi\ vp2 \wedge s\_at\ sp1 = v\_at\ vp2 \\
&\mid (\varphi \longleftrightarrow_F \psi,\ VIffSV\ sp1\ vp2) \Rightarrow s\_check\_exec\ vs\ \varphi\ sp1 \wedge v\_check\_exec\ vs\ \psi\ vp2 \wedge s\_at\ sp1 = v\_at\ vp2 \\
&\mid (\varphi \longleftrightarrow_F \psi,\ VIffVS\ vp1\ sp2) \Rightarrow v\_check\_exec\ vs\ \varphi\ vp1 \wedge s\_check\_exec\ vs\ \psi\ sp2 \wedge v\_at\ vp1 = s\_at\ sp2 \\
&\mid (\exists_F x.\ \varphi,\ VExists\ y\ vp\_part) \Rightarrow (let\ i = v\_at\ (part\_hd\ vp\_part) \\
&\qquad in\ x = y \wedge (\forall (sub, vp) \in SubsVals\ vp\_part.\ v\_at\ vp = i \wedge v\_check\_exec\ (vs\ (x := sub))\ \varphi\ vp)) \\
&\mid (\forall_F x.\ \varphi,\ VForall\ y\ val\ vp) \Rightarrow (x = y \wedge v\_check\_exec\ (vs\ (x := \{val\}))\ \varphi\ vp) \\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrev\ vp) \Rightarrow \\
&\quad (let\ j = v\_at\ vp;\ i = v\_at\ (VPrev\ vp)\ in \\
&\quad i = j{+}1 \wedge v\_check\_exec\ vs\ \varphi\ vp) \\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevZ) \Rightarrow True \\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevOutL\ i) \Rightarrow \\
&\quad i > 0 \wedge \Delta\ \sigma\ i < left\ I \\
&\mid (\mathbf{Y}\ I\ \varphi,\ VPrevOutR\ i) \Rightarrow \\
&\quad i > 0 \wedge enat\ (\Delta\ \sigma\ i) > right\ I \\
&\mid (\mathbf{X}\ I\ \varphi,\ VNext\ vp) \Rightarrow \\
&\quad (let\ j = v\_at\ vp;\ i = v\_at\ (VNext\ vp)\ in \\
&\quad j = i{+}1 \wedge v\_check\_exec\ vs\ \varphi\ vp) \\
&\mid (\mathbf{X}\ I\ \varphi,\ VNextOutL\ i) \Rightarrow \\
&\quad \Delta\ \sigma\ (i{+}1) < left\ I \\
&\mid (\mathbf{X}\ I\ \varphi,\ VNextOutR\ i) \Rightarrow \\
&\quad enat\ (\Delta\ \sigma\ (i{+}1)) > right\ I \\
&\mid (\mathbf{P}\ I\ \varphi,\ VOnceOut\ i) \Rightarrow \\
&\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I \\
&\mid (\mathbf{P}\ I\ \varphi,\ VOnce\ i\ li\ vps) \Rightarrow \\
&\quad (li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b) \\
&\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \\
&\quad \wedge map\ v\_at\ vps = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1] \\
&\quad \wedge (\forall vp \in set\ vps.\ v\_check\_exec\ vs\ \varphi\ vp)) \\
&\mid (\mathbf{F}\ I\ \varphi,\ VEventually\ i\ hi\ vps) \Rightarrow \\
&\quad (hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \wedge right\ I \neq \infty \\
&\quad \wedge map\ v\_at\ vps = [(ETP\_f\ \sigma\ i\ I)\ ..<\ hi + 1] \\
&\quad \wedge (\forall vp \in set\ vps.\ v\_check\_exec\ vs\ \varphi\ vp)) \\
&\mid (\mathbf{H}\ I\ \varphi,\ VHistorically\ i\ vp) \Rightarrow \\
&\quad (let\ j = v\_at\ vp\ in \\
&\quad j \leq i \wedge mem\ (\tau\ \sigma\ i - \tau\ \sigma\ j)\ I \wedge v\_check\_exec\ vs\ \varphi\ vp) \\
&\mid (\mathbf{G}\ I\ \varphi,\ VAlways\ i\ vp) \Rightarrow \\
&\quad (let\ j = v\_at\ vp \\
&\quad in\ j \geq i \wedge mem\ (\tau\ \sigma\ j - \tau\ \sigma\ i)\ I \wedge v\_check\_exec\ vs\ \varphi\ vp) \\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceOut\ i) \Rightarrow \\
&\quad \tau\ \sigma\ i < \tau\ \sigma\ 0 + left\ I \\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSince\ i\ vp1\ vp2s) \Rightarrow \\
&\quad (let\ j = v\_at\ vp1\ in \\
&\quad (case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b \leq j) \wedge j \leq i \\
&\quad \wedge \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i \\
&\quad \wedge map\ v\_at\ vp2s = [j\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1] \wedge v\_check\_exec\ vs\ \varphi\ vp1 \\
&\quad \wedge (\forall vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2)) \\
&\mid (\varphi\ \mathbf{S}\ I\ \psi,\ VSinceInf\ i\ li\ vp2s) \Rightarrow
\end{aligned}
$$

$(li = (case\ right\ I\ of\ \infty \Rightarrow 0 \mid enat\ b \Rightarrow ETP\_p\ \sigma\ i\ b)$
$\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
$\wedge\ map\ v\_at\ vp2s = [li\ ..<\ (LTP\_p\ \sigma\ i\ I) + 1]$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$
$\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntil\ i\ vp2s\ vp1) \Rightarrow$
$(let\ j = v\_at\ vp1\ in$
$(case\ right\ I\ of\ \infty \Rightarrow True \mid enat\ b \Rightarrow j < LTP\_f\ \sigma\ i\ b) \wedge i \leq j$
$\wedge\ map\ v\_at\ vp2s = [ETP\_f\ \sigma\ i\ I\ ..<\ j + 1] \wedge v\_check\_exec\ vs\ \varphi\ vp1$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$
$\mid (\varphi\ \mathbf{U}\ I\ \psi,\ VUntilInf\ i\ hi\ vp2s) \Rightarrow$
$(hi = (case\ right\ I\ of\ enat\ b \Rightarrow LTP\_f\ \sigma\ i\ b) \wedge right\ I \neq \infty$
$\wedge\ map\ v\_at\ vp2s = [ETP\_f\ \sigma\ i\ I\ ..<\ hi + 1]$
$\wedge\ (\forall\ vp2 \in set\ vp2s.\ v\_check\_exec\ vs\ \psi\ vp2))$
$\mid (\_\ ,\ \_) \Rightarrow False)$

declare *s_check_exec.simps*[*simp del*] *v_check_exec.simps*[*simp del*]
simps_of_case *s_check_exec_simps*[*simp*, *code*]: *s_check_exec.simps*[*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 φ sp* ⟷ (∀ *v* ∈ *compatible_vals* (*fv φ*) *vs*. *s_check v φ sp*)
and *v_check_exec vs φ vp* ⟷ (∀ *v* ∈ *compatible_vals* (*fv φ*) *vs*. *v_check v φ vp*)
⟨*proof*⟩

lemma *s_check_code*[*code*]: *s_check v φ sp* = *s_check_exec* (λ*x*. {*v x*}) *φ sp*
⟨*proof*⟩

lemma *v_check_code*[*code*]: *v_check v φ vp* = *v_check_exec* (λ*x*. {*v x*}) *φ vp*
⟨*proof*⟩

8.3 Latest Relevant Time-Point

fun *LRTP* :: ('*n*, '*d*) *formula* ⇒ *nat* ⇒ *nat option* **where**
LRTP ⊤ *i* = *Some i*
| *LRTP* ⊥ *i* = *Some i*
| *LRTP* (_ † _) *i* = *Some i*
| *LRTP* (_ ≈ _) *i* = *Some i*
| *LRTP* (¬*F φ*) *i* = *LRTP φ i*
| *LRTP* (*φ* ∨*F ψ*) *i* = *max_opt* (*LRTP φ i*) (*LRTP ψ i*)
| *LRTP* (*φ* ∧*F ψ*) *i* = *max_opt* (*LRTP φ i*) (*LRTP ψ i*)
| *LRTP* (*φ* ⟶*F ψ*) *i* = *max_opt* (*LRTP φ i*) (*LRTP ψ i*)
| *LRTP* (*φ* ⟷*F ψ*) *i* = *max_opt* (*LRTP φ i*) (*LRTP ψ i*)
| *LRTP* (∃*F*_. *φ*) *i* = *LRTP φ i*
| *LRTP* (∀*F*_. *φ*) *i* = *LRTP φ i*
| *LRTP* (**Y** *I φ*) *i* = *LRTP φ* (*i*−*1*)
| *LRTP* (**X** *I φ*) *i* = *LRTP φ* (*i*+*1*)
| *LRTP* (**P** *I φ*) *i* = *LRTP φ* (*LTP_p_safe σ i I*)
| *LRTP* (**H** *I φ*) *i* = *LRTP φ* (*LTP_p_safe σ i I*)
| *LRTP* (**F** *I φ*) *i* = (*case right I of* ∞ ⇒ *None* | *enat b* ⇒ *LRTP φ* (*LTP_f σ i b*))
| *LRTP* (**G** *I φ*) *i* = (*case right I of* ∞ ⇒ *None* | *enat b* ⇒ *LRTP φ* (*LTP_f σ i b*))
| *LRTP* (*φ* **S** *I ψ*) *i* = *max_opt* (*LRTP φ i*) (*LRTP ψ* (*LTP_p_safe σ i I*))
| *LRTP* (*φ* **U** *I ψ*) *i* = (*case right I of* ∞ ⇒ *None* | *enat b* ⇒ *max_opt* (*LRTP φ* ((*LTP_f σ i b*)−*1*))
(*LRTP ψ* (*LTP_f σ i b*)))

lemma *fb_LRTP*:
assumes *future_bounded φ*
shows ¬ *Option.is_none* (*LRTP φ i*)
⟨*proof*⟩

lemma *not_none_fb_LRTP*:
assumes *future_bounded φ*
shows *LRTP φ i* ≠ *None*
⟨*proof*⟩

lemma *is_some_fb_LRTP*:
assumes *future_bounded φ*
shows ∃ *j*. *LRTP φ i* = *Some j*
⟨*proof*⟩

lemma *enat_trans*[*simp*]: *enat i* ≤ *enat j* ∧ *enat j* ≤ *enat k* ⟹ *enat i* ≤ *enat k*
⟨*proof*⟩

8.4 Active Domain

definition *AD* :: ('*n*, '*d*) *formula* ⇒ *nat* ⇒ '*d set*
where *AD φ i* = *consts φ* ∪ (⋃ *k* ≤ *the* (*LRTP φ i*). ⋃ (*set* ‘ *snd* ‘ Γ *σ k*))

lemma *val_in_AD_iff*:
x ∈ *fv φ* ⟹ *v x* ∈ *AD φ i* ⟷ *v x* ∈ *consts φ* ∨

$(\exists\, r\ ts\ k.\ k \leq the\ (LRTP\ \varphi\ i) \wedge (r,\ v[\![ts]\!]) \in \Gamma\ \sigma\ k \wedge x \in \bigcup\ (set\ (map\ fv\_trm\ ts)))$
⟨*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* φ **and** $AD\ \varphi\ i = (UNIV :: 'd\ set)$
shows *finite* $(UNIV :: 'd\ set)$
⟨*proof*⟩

8.5 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 *LTP_p_mono*: $i \leq j \Longrightarrow LTP\_p\_safe\ \sigma\ i\ I \leq LTP\_p\_safe\ \sigma\ j\ I$
⟨*proof*⟩

lemma *LTP_f_mono*:
assumes $i \leq j$
shows*LTP_f* $\sigma\ i\ b \leq LTP\_f\ \sigma\ j\ b$
⟨*proof*⟩

lemma *LRTP_mono*: *future_bounded* $\varphi \Longrightarrow i \leq j \Longrightarrow the\ (LRTP\ \varphi\ i) \leq the\ (LRTP\ \varphi\ j)$
⟨*proof*⟩

lemma *AD_mono*: *future_bounded* $\varphi \Longrightarrow i \leq j \Longrightarrow AD\ \varphi\ i \subseteq AD\ \varphi\ j$
⟨*proof*⟩

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

lemma *check_AD_cong*:
assumes *future_bounded* φ
and $(\forall x \in fv\ \varphi.\ v\ x = v'\ x \vee (v\ x \notin AD\ \varphi\ i \wedge v'\ x \notin AD\ \varphi\ i))$
shows $(s\_at\ sp = i \Longrightarrow s\_check\ v\ \varphi\ sp \longleftrightarrow s\_check\ v'\ \varphi\ sp)$
$(v\_at\ vp = i \Longrightarrow v\_check\ v\ \varphi\ vp \longleftrightarrow v\_check\ v'\ \varphi\ vp)$
⟨*proof*⟩

8.6 Checker Completeness

lemma *part_hd_tabulate*: *distinct xs* $\Longrightarrow$ *part_hd* (*tabulate xs f z*) = (*case xs of* [] ⇒ *z* | (*x* # _) ⇒ (*if set xs = UNIV then f x else z*))
⟨*proof*⟩

lemma *s_at_tabulate*:
assumes $\forall z.\ s\_at\ (mypick\ z) = i$
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* φ *i*)) *mypick* (*mypick* (*SOME z. z* $\notin$ *AD* φ *i*))
shows $\forall (sub,\ vp) \in SubsVals\ mypart.\ s\_at\ vp = i$
⟨*proof*⟩

lemma *v_at_tabulate*:
assumes $\forall z.\ v\_at\ (mypick\ z) = i$
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* φ *i*)) *mypick* (*mypick* (*SOME z. z* $\notin$ *AD* φ *i*))
shows $\forall (sub,\ vp) \in SubsVals\ mypart.\ v\_at\ vp = i$
⟨*proof*⟩

lemma *s_check_tabulate*:
assumes *future_bounded* φ
and $\forall z.\ s\_at\ (mypick\ z) = i$
and $\forall z.\ s\_check\ (v(x{:=}z))\ \varphi\ (mypick\ z)$
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* φ *i*)) *mypick* (*mypick* (*SOME z. z* $\notin$ *AD* φ *i*))
shows $\forall (sub,\ vp) \in SubsVals\ mypart.\ \forall z \in sub.\ s\_check\ (v(x := z))\ \varphi\ vp$
⟨*proof*⟩

lemma *v_check_tabulate*:
assumes *future_bounded* φ
and $\forall z.\ v\_at\ (mypick\ z) = i$
and $\forall z.\ v\_check\ (v(x{:=}z))\ \varphi\ (mypick\ z)$
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* φ *i*)) *mypick* (*mypick* (*SOME z. z* $\notin$ *AD* φ *i*))
shows $\forall (sub,\ vp) \in SubsVals\ mypart.\ \forall z \in sub.\ v\_check\ (v(x := z))\ \varphi\ vp$
⟨*proof*⟩

lemma *s_at_part_hd_tabulate*:
assumes *future_bounded* φ
and $\forall z.\ s\_at\ (f\ z) = i$
and *mypart* = *tabulate* (*sorted_list_of_set* (*AD* 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* φ $\longrightarrow$ *future_bounded* φ $\longrightarrow$ ($\exists$ *sp*. *s_at sp* = *i* $\wedge$ *s_check v* φ *sp*)) $\wedge$
(*VIO* σ *v i* φ $\longrightarrow$ *future_bounded* φ $\longrightarrow$ ($\exists$ *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* $\Rightarrow$ *s_check v* φ *sp* | *Inr vp* $\Rightarrow$ *v_check v* φ *vp*)
definition *p_check_exec vs* φ *p* = (*case p of Inl sp* $\Rightarrow$ *s_check_exec vs* φ *sp* | *Inr vp* $\Rightarrow$ *v_check_exec vs* φ *vp*)

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

end

8.7 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* φ $\cup$ *vars e*. *v x* = *v*′ *x* $\Longrightarrow$ *check_one* σ *v* φ *e* = *check_one* σ *v*′ φ *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 σ *vs* φ *e* $\longleftrightarrow$ ($\forall$ *v* $\in$ *compatible_vals* (*fv* φ) *vs*. *check_one* σ *v* φ *e*)
⟨*proof*⟩

definition *check_all* :: ('*n*, '*d* :: {*default*, *linorder*}) *trace* $\Rightarrow$ ('*n*, '*d*) *formula* $\Rightarrow$ ('*n*, '*d*) *expl* $\Rightarrow$ *bool* **where**
check_all σ φ *e* = (*distinct_paths e* $\wedge$ *check_all_aux* σ (λ_. *UNIV*) φ *e*)

lemma *check_one_alt*: *check_one* σ *v* φ *e* = *p_check* σ *v* φ (*eval_pdt v e*)
⟨*proof*⟩

lemma *check_all_alt*: *check_all* σ φ *e* = (*distinct_paths e* $\wedge$ ($\forall$ *v*. *p_check* σ *v* φ (*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 φ :: ('*n*, '*d* :: {*default*, *linorder*}) *formula*
shows *set vs* $\subseteq$ *fv* φ $\Longrightarrow$ *future_bounded* φ $\Longrightarrow$ *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* σ *v* φ (*eval_pdt v e*))
⟨*proof*⟩

lemma *check_all_completeness*:
fixes φ :: ('*n*, '*d* :: {*default*, *linorder*}) *formula*
assumes *future_bounded* φ
shows $\exists$ *e*. *pdt_at i e* $\wedge$ *check_all* σ φ *e*
⟨*proof*⟩

lemma *check_all_soundness_aux*: *check_all* σ φ *e* $\Longrightarrow$ *p* = *eval_pdt v e* $\Longrightarrow$ *isl p* $\longleftrightarrow$ *sat* σ *v* (*p_at p*) φ
⟨*proof*⟩

lemma *check_all_soundness*: *check_all* σ φ *e* $\Longrightarrow$ *pdt_at i e* $\Longrightarrow$ *isl* (*eval_pdt v e*) $\longleftrightarrow$ *sat* σ *v i* φ
⟨*proof*⟩

unbundle *MFOTL_no_notation* — disable notation

9 Type of Events

9.1 Code Adaptation for 8-bit strings

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

setup_lifting *type_definition_string8*

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

9.2 Event Parameters

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

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

lemma *b* ≠ *0* ⟹ *div_to_zero a b* ∗ *b* + *mod_to_zero a b* = *a*
⟨*proof*⟩

datatype *event_data* = *EInt integer* | *EString string8*

instantiation *event_data* :: {*ord*, *plus*, *minus*, *uminus*, *times*, *divide*, *modulo*}
begin

fun *less_eq_event_data* **where**
EInt x ≤ *EInt y* ⟷ *x* ≤ *y*
| *EString x* ≤ *EString y* ⟷ *x* ≤ *y*
| *EInt* _ ≤ *EString* _ ⟷ *True*
| (_ :: *event_data*) ≤ _ ⟷ *False*

definition *less_event_data* :: *event_data* ⇒ *event_data* ⇒ *bool* **where**
less_event_data x y ⟷ *x* ≤ *y* ∧ ¬ *y* ≤ *x*

fun *plus_event_data* **where**
EInt x + *EInt y* = *EInt* (*x* + *y*)

| (_::*event_data*) + _ = *undefined*

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

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

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

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

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

instance ⟨*proof*⟩

end

lemma *infinite_UNIV_event_data*:
  ¬*finite* (*UNIV* :: *event_data set*)
⟨*proof*⟩

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

instantiation *event_data* :: *default* **begin**

definition *default_event_data* :: *event_data* **where** *default* = *EInt 0*

instance ⟨*proof*⟩

end

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

end

10 Code Generation

10.1 Type Class Instances

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

begin

lemma *finite_coset*: *finite* (*List.coset* (*xs* :: *'a list*)) = (*case universe of None* ⇒ *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

10.2 Progress

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

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

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

lemma *progress_Eventually_code*: *progress σ* (*Formula.Eventually I φ*) *j* =
(*let m* = *min j* (*Suc* (*progress σ φ j*)) − *1 in Min* (*Set.filter* (λ*i*. *enat* (*δ σ m i*) ≤ *right I*) {..*j*}))
⟨*proof*⟩

lemma *progress_Always_code*: *progress σ* (*Formula.Always I φ*) *j* =
(*let m* = *min j* (*Suc* (*progress σ φ j*)) − *1 in Min* (*Set.filter* (λ*i*. *enat* (*δ σ m i*) ≤ *right I*) {..*j*}))
⟨*proof*⟩

lemma *progress_Until_code*: *progress σ* (*Formula.Until φ I ψ*) *j* =
(*let m* = *min j* (*Suc* (*min* (*progress σ φ j*) (*progress σ ψ j*))) − *1 in Min* (*Set.filter* (λ*i*. *enat* (*δ σ m i*) ≤ *right I*) {..*j*}))
⟨*proof*⟩

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

10.3 Trace

lemma *snth_Stream_eq*: (*x* ## *s*) !! *n* = (*case n of 0* ⇒ *x* | *Suc m* ⇒ *s* !! *m*)
⟨*proof*⟩

lemma *extend_is_stream*:
assumes *sorted* (*map snd list*)
and ⋀*x*. *x* ∈ *set list* ⟹ *snd x* ≤ *m*
and ⋀*x*. *x* ∈ *set list* ⟹ *finite* (*fst x*)
shows *ssorted* (*smap snd* (*list* @− *smap* (λ*n*. ({}, *n* + *m*)) *nats*)) ∧
sincreasing (*smap snd* (*list* @− *smap* (λ*n*. ({}, *n* + *m*)) *nats*)) ∧
sfinite (*smap fst* (*list* @− *smap* (λ*n*. ({}, *n* + *m*)) *nats*))
⟨*proof*⟩

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

setup_lifting *type_definition_trace_mapping*

lemma *lookup_bulkload_Some*: *i* < *length list* ⟹
Mapping.lookup (*Mapping.bulkload list*) *i* = *Some* (*list* ! *i*)
⟨*proof*⟩

lift_definition *trace_mapping_of_list* :: (*'a set* × *nat*) *list* ⇒ *'a trace_mapping* **is**
λ*xs*. *if sorted* (*map snd xs*) ∧ (∀ *x* ∈ *set xs*. *finite* (*fst x*)) *then* (*if xs* = [] *then* (*0*, *0*, *Mapping.empty*)
else (*length xs*, *snd* (*last xs*), *Mapping.bulkload xs*))
else (*0*, *0*, *Mapping.empty*)
⟨*proof*⟩

lift_definition *trace_mapping_nth* :: *'a trace_mapping* ⇒ *nat* ⇒ (*'a set* × *nat*) **is**
λ(*n*, *m*, *t*) *i*. *if i* < *n then the* (*Mapping.lookup t i*) *else* ({}, (*i* − *n*) + *m*) ⟨*proof*⟩

lift_definition *Trace_Mapping* :: *'a trace_mapping* ⇒ *'a Trace.trace* **is**
λ(*n*, *m*, *t*). *map* (*the* ∘ *Mapping.lookup t*) [*0*..<*n*] @− *smap* (λ*n*. ({} :: *'a set*, *n* + *m*)) *nats*

⟨*proof*⟩

code_datatype *Trace_Mapping*

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

lemma Γ*_rbt_code*[*code*]: Γ (*Trace_Mapping t*) *i* = *fst* (*trace_mapping_nth t i*)
⟨*proof*⟩

lemma τ*_rbt_code*[*code*]: τ (*Trace_Mapping t*) *i* = *snd* (*trace_mapping_nth t i*)
⟨*proof*⟩

lemma *trace_mapping_of_list_sound*: *sorted* (*map snd xs*) ∧ (∀ *x* ∈ *set xs*. *finite* (*fst x*)) ⟹ *i* < *length xs* ⟹
xs ! *i* = (Γ (*trace_of_list xs*) *i*, τ (*trace_of_list xs*) *i*)
⟨*proof*⟩

10.4 Auxiliary results

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

lemma *sum_proofs_empty*[*simp*]: *sum_proofs f* [] = *0*
⟨*proof*⟩

lemma *sum_proofs_fundef_cong*[*fundef_cong*]: (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *f′ x*) ⟹
sum_proofs f xs = *sum_proofs f′ xs*
⟨*proof*⟩

lemma *sum_proofs_Cons*:
fixes *f* :: *′a* ⇒ *nat*
shows *sum_proofs f* (*p* # *qs*) = *f p* + *sum_proofs f qs*
⟨*proof*⟩

lemma *sum_proofs_app*:
fixes *f* :: *′a* ⇒ *nat*
shows *sum_proofs f* (*qs* @ [*p*]) = *f p* + *sum_proofs f qs*
⟨*proof*⟩

context
fixes *w* :: *′n* ⇒ *nat*
begin

function (*sequential*) *s_pred* :: (*′n*, *′d*) *sproof* ⇒ *nat*
and *v_pred* :: (*′n*, *′d*) *vproof* ⇒ *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
```

10.5 *v_check_exec* setup

```
lemma ETP_minus_le_iff: ETP σ (τ σ i − n) ≤ j ⟷ δ σ i j ≤ n
  ⟨proof⟩

lemma ETP_minus_gt_iff: j < ETP σ (τ σ i − n) ⟷ δ σ i j > n
  ⟨proof⟩

lemma nat_le_iff_less:
  fixes n :: nat
  shows (j ≤ n) ⟷ (j = 0 ∨ j − 1 < n)
  ⟨proof⟩
```

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

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

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

lemma *LTP_minus_lt_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 $LTP\ \sigma\ (\tau\ \sigma\ i - n) < j \longleftrightarrow (\textit{if}\ \neg\ j \leq i \wedge n = 0\ \textit{then}\ \delta\ \sigma\ j\ i > 0\ \textit{else}\ \delta\ \sigma\ i\ j < n)$
⟨*proof*⟩

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

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

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

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

lemma *check_upt_l_cong*:
assumes $\bigwedge j.\ j \leq max\ i\ li \Longrightarrow \tau\ \sigma\ j = \tau\ \sigma'\ j$
shows *check_upt_LTP_p* $\sigma\ I\ li\ xs\ i =$ *check_upt_LTP_p* $\sigma'\ I\ li\ xs\ i$
⟨*proof*⟩

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

lemma *v_check_exec_Once_code*[*code*]: *v_check_exec* $\sigma\ vs\ (Formula.Once\ I\ \varphi)\ vp = (\textit{case vp of}$
VOnce i li vps $\Rightarrow$
(*case right I of* $\infty \Rightarrow li = 0 \mid enat\ b \Rightarrow ((li = 0 \vee b < \delta\ \sigma\ i\ (li - 1)) \wedge \delta\ \sigma\ i\ li \leq b)$)
$\wedge\ \tau\ \sigma\ 0 + left\ I \leq \tau\ \sigma\ i$
$\wedge$ *check_upt_LTP_p* $\sigma\ I\ li\ (map\ v\_at\ vps)\ i \wedge Ball\ (set\ vps)\ (v\_check\_exec\ \sigma\ vs\ \varphi)$

| *VOnceOut i* ⇒ *τ σ i* < *τ σ 0* + *left I*
| _ ⇒ *False*)
⟨*proof*⟩

lemma *s_check_exec_Historically_code*[*code*]: *s_check_exec σ vs* (*Formula.Historically I φ*) *vp* = (*case vp of*
SHistorically i li vps ⇒
(*case right I of* ∞ ⇒ *li* = *0* | *enat b* ⇒ ((*li* = *0* ∨ *b* < *δ σ i* (*li* − *1*)) ∧ *δ σ i li* ≤ *b*))
∧ *τ σ 0* + *left I* ≤ *τ σ i*
∧ *check_upt_LTP_p σ I li* (*map s_at vps*) *i* ∧ *Ball* (*set vps*) (*s_check_exec σ vs φ*)
| *SHistoricallyOut i* ⇒ *τ σ i* < *τ σ 0* + *left I*
| _ ⇒ *False*)
⟨*proof*⟩

lemma *v_check_exec_Since_code*[*code*]: *v_check_exec σ vs* (*Formula.Since φ I ψ*) *vp* = (*case vp of*
VSince i vp1 vp2s ⇒
let j = *v_at vp1 in*
(*case right I of* ∞ ⇒ *True* | *enat b* ⇒ *δ σ i j* ≤ *b*) ∧ *j* ≤ *i*
∧ *τ σ 0* + *left I* ≤ *τ σ i*
∧ *check_upt_LTP_p σ I j* (*map v_at vp2s*) *i*
∧ *v_check_exec σ vs φ vp1* ∧ *Ball* (*set vp2s*) (*v_check_exec σ vs ψ*)
| *VSinceInf i li vp2s* ⇒
(*case right I of* ∞ ⇒ *li* = *0* | *enat b* ⇒ ((*li* = *0* ∨ *b* < *δ σ i* (*li* − *1*)) ∧ *δ σ i li* ≤ *b*)) ∧
τ σ 0 + *left I* ≤ *τ σ i* ∧
check_upt_LTP_p σ I li (*map v_at vp2s*) *i* ∧ *Ball* (*set vp2s*) (*v_check_exec σ vs ψ*)
| *VSinceOut i* ⇒ *τ σ i* < *τ σ 0* + *left I*
| _ ⇒ *False*)
⟨*proof*⟩

lemma *ETP_f_le_iff*: *max i* (*ETP σ* (*τ σ i* + *a*)) ≤ *j* ⟷ *i* ≤ *j* ∧ *δ σ j i* ≥ *a*
⟨*proof*⟩

lemma *ETP_f_ge_iff*: *j* ≤ *max i* (*ETP σ* (*τ σ i* + *n*)) ⟷ (*case n of 0* ⇒ *j* ≤ *i*
| *Suc n′* ⇒ (*case j of 0* ⇒ *True* | *Suc j′* ⇒ *δ σ 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*⟩

10.6 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 σ t* = *ETP_rec σ t 0*
⟨*proof*⟩

lemma *LTP_aux*:
assumes *τ σ* (*Suc i*) ≤ *t*
shows *i* ≤ *Max* {*i. τ σ i* ≤ *t*}
⟨*proof*⟩

function (*sequential*) *LTP_rec* **where**
LTP_rec σ t i = (*if τ σ* (*Suc i*) ≤ *t then LTP_rec σ t* (*i* + *1*) *else i*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *LTP_rec_sound*: *LTP_rec σ t j* = *Max* ({*i. i* ≥ *j* ∧ (*τ σ i*) ≤ *t*} ∪ {*j*})
⟨*proof*⟩

lemma *LTP_code*[*code*]: *LTP σ t* = (*if t* < *τ σ 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*⟩

10.7 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**
λ*xs*.
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*⟩

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

collect_paths_specialized
in *OCaml* **module_name** *Checker* **file_prefix** *checker*

11 Unverified Explanation-Producing Monitoring Algorithm

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

fun *merge_part3_raw* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'e) \Rightarrow ('d\ set \times 'a)\ list \Rightarrow ('d\ set \times 'b)\ list \Rightarrow ('d\ set \times 'c)\ list \Rightarrow ('d\ set \times 'e)\ list$ **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 $\mathcal{P}$ $\Longrightarrow$ $\mathcal{P}$ = {} $\longleftrightarrow$ *X* = {}
partition_on X $\mathcal{P}$ $\Longrightarrow$ $\mathcal{P}$ $\neq$ {} $\longleftrightarrow$ *X* $\neq$ {}
⟨*proof*⟩

lemma *wf_part_list_filter_inter*:
defines *inP1 P1 f v1 part2*
≡ *List.map_filter* (λ(*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*
≡ *List.map_filter* (λ(*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*)) ∧ *distinct* (*map fst part1*) ⟹
partition_on X (*set* (*map fst part2*)) ∧ *distinct* (*map fst part2*) ⟹
partition_on X (*set* (*map fst* (*merge_part2_raw f part1 part2*)))
∧ *distinct* (*map fst* (*merge_part2_raw f part1 part2*))
⟨*proof*⟩

lemma *wf_part_list_merge_part3_raw*:
partition_on X (*set* (*map fst part1*)) ∧ *distinct* (*map fst part1*) ⟹
partition_on X (*set* (*map fst part2*)) ∧ *distinct* (*map fst part2*) ⟹
partition_on X (*set* (*map fst part3*)) ∧ *distinct* (*map fst part3*) ⟹
partition_on X (*set* (*map fst* (*merge_part3_raw f part1 part2 part3*)))
∧ *distinct* (*map fst* (*merge_part3_raw f part1 part2 part3*))
⟨*proof*⟩

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

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

definition *proof_app* :: ('*n*, '*d*) *proof* ⇒ ('*n*, '*d*) *proof* ⇒ ('*n*, '*d*) *proof* (**infixl** ⊕ *65*) **where**
p ⊕ *q* = (*case* (*p*, *q*) *of*
(*Inl* (*SHistorically i li sps*), *Inl q*) ⇒ *Inl* (*SHistorically* (*i*+1) *li* (*sps* @ [*q*]))
| (*Inl* (*SAlways i hi sps*), *Inl q*) ⇒ *Inl* (*SAlways* (*i*−1) *hi* (*q* # *sps*))
| (*Inl* (*SSince sp2 sp1s*), *Inl q*) ⇒ *Inl* (*SSince sp2* (*sp1s* @ [*q*]))
| (*Inl* (*SUntil sp1s sp2*), *Inl q*) ⇒ *Inl* (*SUntil* (*q* # *sp1s*) *sp2*)
| (*Inr* (*VSince i vp1 vp2s*), *Inr q*) ⇒ *Inr* (*VSince* (*i*+1) *vp1* (*vp2s* @ [*q*]))
| (*Inr* (*VOnce i li vps*), *Inr q*) ⇒ *Inr* (*VOnce* (*i*+1) *li* (*vps* @ [*q*]))
| (*Inr* (*VEventually i hi vps*), *Inr q*) ⇒ *Inr* (*VEventually* (*i*−1) *hi* (*q* # *vps*))
| (*Inr* (*VSinceInf i li vp2s*), *Inr q*) ⇒ *Inr* (*VSinceInf* (*i*+1) *li* (*vp2s* @ [*q*]))
| (*Inr* (*VUntil i vp2s vp1*), *Inr q*) ⇒ *Inr* (*VUntil* (*i*−1) (*q* # *vp2s*) *vp1*)
| (*Inr* (*VUntilInf i hi vp2s*), *Inr q*) ⇒ *Inr* (*VUntilInf* (*i*−1) *hi* (*q* # *vp2s*)))

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

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

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

```
  do_neg p = (case p of
  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 ⇒ ('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 ⇒ ('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 ⇒ 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 ⇒ 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*] (λ*c*. *Leaf* (*Inl* (*SEq_Const i x c*))) (*Leaf* (*Inr* (*VEq_Const i x c*))))
| *eval vs i* (*Formula.Pred r ts*) =
  (*pdt_of i r ts* (*filter* (λ*x*. *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* (λ*p*. *min_list_wrt cmp* (*do_neg p*)) (*eval vs i φ*)
| *eval vs i* (*Formula.Or φ ψ*) = *apply_pdt2 vs* (λ*p1 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* (λ*p1 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* (λ*p1 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* (λ*p1 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*]) (λ*p*. *min_list_wrt cmp* (*do_exists x p*)) (*eval* (*vs* @ [*x*]) *i φ*)
| *eval vs i* (*Formula.Forall x φ*) = *hide_pdt* (*vs* @ [*x*]) (λ*p*. *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 (λ*p*. *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* (λ*l*. *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 (λ*p*. *min_list_wrt cmp* (*do_once_base 0 0 p*)) *expl*
        else (*if right I* ≥ *enat* (Δ *σ i*) *then*
                apply_pdt2 vs (λ*p 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 (λ*p*. *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 (λ*p*. *min_list_wrt cmp* (*do_historically_base 0 0 p*)) *expl*
        else (*if right I* ≥ *enat* (Δ *σ i*) *then*
                apply_pdt2 vs (λ*p p*′. *min_list_wrt cmp* (*do_historically i* (*left I*) *p p*′)) *expl*

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

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

lemma i_less_LTP: τ σ (Suc i) ≤ b + τ σ i ⟹ i < LTP σ (b + τ σ i)
  ⟨proof⟩

termination eval
  ⟨proof⟩

end

end
```

12 Examples

definition *monitor* :: (('*n* :: *linorder* × '*d* :: {*default*, *linorder*} *list*) *set* × *nat*) *list* ⇒ ('*n*, '*d*) *formula* ⇒ ('*n*, '*d*) *expl list* **where**
monitor π φ = *map* (λ*i*. *eval* (*trace_of_list* π) (λ*p q*. *size p* ≤ *size q*) (*sorted_list_of_set* (*fv* φ)) *i* φ) [*0* ..< *length* π]
definition *check* :: (('*n* :: *linorder* × '*d* :: {*default*, *linorder*} *list*) *set* × *nat*) *list* ⇒ ('*n*, '*d*) *formula* ⇒ *bool* **where**
check π φ = *list_all* (*check_all* (*trace_of_list* π) φ) (*monitor* π φ)

12.1 Infinite Domain

definition *prefix* :: ((*string* × *string list*) *set* × *nat*) *list* **where**
prefix =
[({(''*mgr_S*'', [''*Mallory*'', ''*Alice*'']),
(''*mgr_S*'', [''*Merlin*'', ''*Bob*'']),
(''*mgr_S*'', [''*Merlin*'', ''*Charlie*''])}, *1307532861*::*nat*),
({(''*approve*'', [''*Mallory*'', ''*152*''])}, *1307532861*),
({(''*approve*'', [''*Merlin*'', ''*163*'']),
(''*publish*'', [''*Alice*'', ''*160*'']),
(''*mgr_F*'', [''*Merlin*'', ''*Charlie*''])}, *1307955600*),
({(''*approve*'', [''*Merlin*'', ''*187*'']),
(''*publish*'', [''*Bob*'', ''*163*'']),
(''*publish*'', [''*Alice*'', ''*163*'']),
(''*publish*'', [''*Charlie*'', ''*163*'']),
(''*publish*'', [''*Charlie*'', ''*152*''])}, *1308477599*)]

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

value *monitor prefix phi*
lemma *check prefix phi*
⟨*proof*⟩

12.2 Finite Domain

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

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

instantiation *Domain* :: *default* **begin**
definition *default_Domain* = *Default*
instance ⟨*proof*⟩
end
instantiation *Domain* :: *universe* **begin**
definition *universe_Domain* = *Some* [*Mallory*, *Merlin*, *Martin*, *Alice*, *Bob*, *Charlie*, *David*, *Default*, *R42*, *R152*, *R160*, *R163*, *R187*]
instance ⟨*proof*⟩
end
instantiation *Domain* :: *linorder* **begin**
definition *less_eq_Domain d d'* = (*ord d* ≤ *ord d'*)
definition *less_Domain d d'* = (*ord d* < *ord d'*)
instance ⟨*proof*⟩
end

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

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

value *monitor fprefix fphi*
lemma *check fprefix fphi*
⟨*proof*⟩

References

[1] L. Lima, A. Herasimau, M. Raszyk, D. Traytel, and S. Yuan. Explainable online monitoring of metric temporal logic. In S. Sankaranarayanan and N. Sharygina, editors, *TACAS 2023*, volume 13994 of *LNCS*, pages 473–491. Springer, 2023.

[2] L. Lima, J. J. H. y Munive, and D. Traytel. Explainable online monitoring of metric first-order temporal logic. In B. Finkbeiner and L. Kovács, editors, *TACAS 2024*, volume 14570 of *LNCS*, pages 288–307. Springer, 2024.