Formalization of a Monitoring Algorithm for Metric First-Order Temporal Logic

Joshua Schneider Dmitriy Traytel

April 19, 2024

Abstract

A monitor is a runtime verification tool that solves the following problem: Given a stream of time-stamped events and a policy formulated in a specification language, decide whether the policy is satisfied at every point in the stream. We verify the correctness of an executable monitor for specifications given as formulas in metric first-order temporal logic (MFOTL) [1], an expressive extension of linear temporal logic with real-time constraints and first-order quantification. The verified monitor implements a simplified variant of the algorithm used in the efficient MonPoly monitoring tool [2]. The formalization is presented in a RV 2019 paper [4], which also compares the output of the verified monitor to that of other monitoring tools on randomly generated inputs. This case study revealed several errors in the optimized but unverified tools.

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*⟩

typedef $'a\ trace = \{s :: ('a\ set \times nat)\ stream.\ ssorted\ (smap\ snd\ s) \wedge sincreasing\ (smap\ snd\ s)\}$
⟨*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 $\Delta\ s\ i \equiv \tau\ s\ i - \tau\ s\ (i - 1)$

lift_definition *map_*$\Gamma :: ('a\ set \Rightarrow 'b\ set) \Rightarrow 'a\ trace \Rightarrow 'b\ trace$ **is**
$\lambda f\ s.\ smap\ (\lambda(x,\ i).\ (f\ x,\ i))\ s$
⟨*proof*⟩

lemma Γ*_map_*Γ[*simp*]: $\Gamma\ (map\_\Gamma\ f\ s)\ i = f\ (\Gamma\ s\ i)$
⟨*proof*⟩

lemma τ*_map_*Γ[*simp*]: $\tau\ (map\_\Gamma\ f\ s)\ i = \tau\ s\ i$
⟨*proof*⟩

lemma *map_*Γ*_id*[*simp*]: *map_*Γ *id s* = *s*

⟨*proof*⟩

lemma *map_Γ_comp*: *map_Γ g* (*map_Γ f s*) = *map_Γ* (*g* ∘ *f*) *s*
⟨*proof*⟩

lemma *map_Γ_cong*: $\sigma_1 = \sigma_2 \Longrightarrow (\bigwedge x.\ f_1\ x = f_2\ x) \Longrightarrow$ *map_Γ* $f_1\ \sigma_1$ = *map_Γ* $f_2\ \sigma_2$
⟨*proof*⟩

1.2 Finite trace prefixes

typedef *'a prefix* = {*p* :: (*'a set* × *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**
λf. *map* (*λ*(*x*, *i*). (*f x*, *i*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

instance
⟨*proof*⟩

end

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

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

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

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

lemma *prefix_of_pmap_Γ*[*simp*]: *prefix_of* π $\sigma \Longrightarrow$ *prefix_of* (*pmap_Γ* f π) (*map_Γ* f σ)
⟨*proof*⟩

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

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc* p x) $s \Longrightarrow$ *last_ts* $p \leq$ *snd* $x \Longrightarrow$
(*prefix_of* p $s \Longrightarrow$ Γ s (*plen* p) = *fst* $x \Longrightarrow$ τ s (*plen* p) = *snd* $x \Longrightarrow P$) $\Longrightarrow P$
⟨*proof*⟩

lemma *le_pnil*[*simp*]: *pnil* $\leq \pi$
⟨*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 *ex_prefix_of*: ∃ s. *prefix_of* p s
⟨*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* $p \Longrightarrow$ Γ s i = Γ 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 *replace_prefix* :: *'a prefix* ⇒ *'a trace* ⇒ *'a trace* **is**
λπ σ. *if ssorted* (*smap snd* (π @− *sdrop* (*length* π) σ)) *then*
π @− *sdrop* (*length* π) σ *else smap* (λi. ({}, i)) *nats*
⟨*proof*⟩

lemma *prefix_of_replace_prefix*:
prefix_of (*pmap_Γ f* π) σ ⟹ *prefix_of* π (*replace_prefix* π σ)
⟨*proof*⟩

lemma *map_Γ_replace_prefix*:
∀ *x*. *f* (*f x*) = *f x* ⟹ *prefix_of* (*pmap_Γ f* π) σ ⟹ *map_Γ f* (*replace_prefix* π σ) = *map_Γ f* σ
⟨*proof*⟩

lemma *prefix_of_pmap_Γ_D*:
assumes *prefix_of* (*pmap_Γ f* π) σ
shows ∃ σ′. *prefix_of* π σ′ ∧ *prefix_of* (*pmap_Γ f* π) (*map_Γ f* σ′)
⟨*proof*⟩

lemma *prefix_of_map_Γ_D*:
assumes *prefix_of* π′ (*map_Γ f* σ)
shows ∃ π″. π′ = *pmap_Γ f* π″ ∧ *prefix_of* π″ σ
⟨*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*⟩

2 Finite tables

primrec *tabulate* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *nat* ⇒ *'a list* **where**
tabulate f x 0 = []
| *tabulate f x* (*Suc n*) = *f x* # *tabulate f* (*Suc x*) *n*

lemma *tabulate_alt*: *tabulate f x n* = *map f* [*x* ..< *x* + *n*]
⟨*proof*⟩

lemma *length_tabulate*[*simp*]: *length* (*tabulate f x n*) = *n*
⟨*proof*⟩

lemma *map_tabulate*[*simp*]: *map f* (*tabulate g x n*) = *tabulate* (λx. *f* (*g x*)) *x n*
⟨*proof*⟩

lemma *nth_tabulate*[*simp*]: *k* < *n* ⟹ *tabulate f x n* ! *k* = *f* (*x* + *k*)
⟨*proof*⟩

type_synonym *'a tuple* = *'a option list*
type_synonym *'a table* = *'a tuple set*

definition *wf_tuple* :: *nat* ⇒ *nat set* ⇒ *'a tuple* ⇒ *bool* **where**
wf_tuple n V x ⟷ *length x* = *n* ∧ (∀ *i*<*n*. *x*!*i* = *None* ⟷ *i* ∉ *V*)

definition *table* :: *nat* ⇒ *nat set* ⇒ *'a table* ⇒ *bool* **where**
table n V X ⟷ (∀ *x*∈*X*. *wf_tuple n V x*)

definition *empty_table* = {}

definition *unit_table n* = {*replicate n None*}

definition *singleton_table n i x* = {*tabulate* (λ*j*. *if i* = *j then Some x else None*) *0 n*}

lemma *in_empty_table*[*simp*]: ¬ *x* ∈ *empty_table*
⟨*proof*⟩

lemma *empty_table*[*simp*]: *table n V empty_table*
⟨*proof*⟩

lemma *unit_table_wf_tuple*[*simp*]: *V* = {} ⟹ *x* ∈ *unit_table n* ⟹ *wf_tuple n V x*
⟨*proof*⟩

lemma *unit_table*[*simp*]: *V* = {} ⟹ *table n V* (*unit_table n*)
⟨*proof*⟩

lemma *in_unit_table*: *v* ∈ *unit_table n* ⟷ *wf_tuple n* {} *v*
⟨*proof*⟩

lemma *singleton_table_wf_tuple*[*simp*]: *V* = {*i*} ⟹ *x* ∈ *singleton_table n i z* ⟹ *wf_tuple n V x*
⟨*proof*⟩

lemma *singleton_table*[*simp*]: *V* = {*i*} ⟹ *table n V* (*singleton_table n i z*)
⟨*proof*⟩

lemma *table_Un*[*simp*]: *table n V X* ⟹ *table n V Y* ⟹ *table n V* (*X* ∪ *Y*)
⟨*proof*⟩

lemma *wf_tuple_length*: *wf_tuple n V x* ⟹ *length x* = *n*
⟨*proof*⟩

fun *join1* :: *'a tuple* × *'a tuple* ⇒ *'a tuple option* **where**
join1 ([], []) = *Some* []
| *join1* (*None* # *xs*, *None* # *ys*) = *map_option* (*Cons None*) (*join1* (*xs*, *ys*))
| *join1* (*Some x* # *xs*, *None* # *ys*) = *map_option* (*Cons* (*Some x*)) (*join1* (*xs*, *ys*))
| *join1* (*None* # *xs*, *Some y* # *ys*) = *map_option* (*Cons* (*Some y*)) (*join1* (*xs*, *ys*))
| *join1* (*Some x* # *xs*, *Some y* # *ys*) = (*if x* = *y*
then map_option (*Cons* (*Some x*)) (*join1* (*xs*, *ys*))
else None)
| *join1* _ = *None*

definition *join* :: *'a table* ⇒ *bool* ⇒ *'a table* ⇒ *'a table* **where**
join A pos B = (*if pos then Option.these* (*join1* ‘ (*A* × *B*))
else A − *Option.these* (*join1* ‘ (*A* × *B*)))

lemma *join_True_code*[*code*]: *join A True B* = (⋃ *a* ∈ *A*. ⋃ *b* ∈ *B*. *set_option* (*join1* (*a*, *b*)))
⟨*proof*⟩

lemma *join_False_alt*: *join X False Y* = *X* − *join X True Y*
⟨*proof*⟩

lemma *self_join1*: *join1* (*xs*, *ys*) ≠ *Some xs* ⟹ *join1* (*zs*, *ys*) ≠ *Some xs*
⟨*proof*⟩

lemma *join_False_code*[*code*]: *join A False B* = {*a* ∈ *A*. ∀ *b* ∈ *B*. *join1* (*a*, *b*) ≠ *Some a*}

⟨*proof*⟩

lemma *wf_tuple_Nil*[*simp*]: *wf_tuple n A* [] = (*n* = *0*)
⟨*proof*⟩

lemma *Suc_pred'*: *Suc* (*x* − *Suc 0*) = (*case x of 0* ⇒ *Suc 0* | _ ⇒ *x*)
⟨*proof*⟩

lemma *wf_tuple_Cons*[*simp*]:
wf_tuple n A (*x* # *xs*) ⟷ ((*if x* = *None then 0* ∉ *A else 0* ∈ *A*) ∧
(∃ *m*. *n* = *Suc m* ∧ *wf_tuple m* ((λ*x*. *x* − *1*) ‘ (*A* − {*0*})) *xs*))
⟨*proof*⟩

lemma *join1_wf_tuple*:
join1 (*v1*, *v2*) = *Some v* ⟹ *wf_tuple n A v1* ⟹ *wf_tuple n B v2* ⟹ *wf_tuple n* (*A* ∪ *B*) *v*
⟨*proof*⟩

lemma *join_wf_tuple*: *x* ∈ *join X b Y* ⟹
∀ *v* ∈ *X*. *wf_tuple n A v* ⟹ ∀ *v* ∈ *Y*. *wf_tuple n B v* ⟹ (¬ *b* ⟹ *B* ⊆ *A*) ⟹ *A* ∪ *B* = *C* ⟹
wf_tuple n C x
⟨*proof*⟩

lemma *join_table*: *table n A X* ⟹ *table n B Y* ⟹ (¬ *b* ⟹ *B* ⊆ *A*) ⟹ *A* ∪ *B* = *C* ⟹
table n C (*join X b Y*)
⟨*proof*⟩

lemma *wf_tuple_Suc*: *wf_tuple* (*Suc m*) *A a* ⟷ *a* ≠ [] ∧
wf_tuple m ((λ*x*. *x* − *1*) ‘ (*A* − {*0*})) (*tl a*) ∧ (*0* ∈ *A* ⟷ *hd a* ≠ *None*)
⟨*proof*⟩

lemma *table_project*: *table* (*Suc n*) *A X* ⟹ *table n* ((λ*x*. *x* − *Suc 0*) ‘ (*A* − {*0*})) (*tl* ‘ *X*)
⟨*proof*⟩

definition *restrict* **where**
restrict A v = *map* (λ*i*. *if i* ∈ *A then v* ! *i else None*) [*0* ..< *length v*]

lemma *restrict_Nil*[*simp*]: *restrict A* [] = []
⟨*proof*⟩

lemma *restrict_Cons*[*simp*]: *restrict A* (*x* # *xs*) =
(*if 0* ∈ *A then x* # *restrict* ((λ*x*. *x* − *1*) ‘ (*A* − {*0*})) *xs else None* # *restrict* (λ*x*. *x* − *1*) ‘ *A*) *xs*)
⟨*proof*⟩

lemma *wf_tuple_restrict*: *wf_tuple n B v* ⟹ *A* ∩ *B* = *C* ⟹ *wf_tuple n C* (*restrict A v*)
⟨*proof*⟩

lemma *wf_tuple_restrict_simple*: *wf_tuple n B v* ⟹ *A* ⊆ *B* ⟹ *wf_tuple n A* (*restrict A v*)
⟨*proof*⟩

lemma *nth_restrict*: *i* ∈ *A* ⟹ *i* < *length v* ⟹ *restrict A v* ! *i* = *v* ! *i*
⟨*proof*⟩

lemma *restrict_eq_Nil*[*simp*]: *restrict A v* = [] ⟷ *v* = []
⟨*proof*⟩

lemma *length_restrict*[*simp*]: *length* (*restrict A v*) = *length v*
⟨*proof*⟩

lemma *join1_Some_restrict*:
fixes $x\ y :: 'a\ tuple$
assumes $wf\_tuple\ n\ A\ x\ wf\_tuple\ n\ B\ y$
shows $join1\ (x, y) = Some\ z \longleftrightarrow wf\_tuple\ n\ (A \cup B)\ z \wedge restrict\ A\ z = x \wedge restrict\ B\ z = y$
⟨*proof*⟩

lemma *restrict_idle*: $wf\_tuple\ n\ A\ v \Longrightarrow restrict\ A\ v = v$
⟨*proof*⟩

lemma *map_the_restrict*:
$i \in A \Longrightarrow map\ the\ (restrict\ A\ v)\ !\ i = map\ the\ v\ !\ i$
⟨*proof*⟩

lemma *join_restrict*:
fixes $X\ Y :: 'a\ tuple\ set$
assumes $\bigwedge v.\ v \in X \Longrightarrow wf\_tuple\ n\ A\ v\ \bigwedge v.\ v \in Y \Longrightarrow wf\_tuple\ n\ B\ v \neg\ b \Longrightarrow B \subseteq A$
shows $v \in join\ X\ b\ Y \longleftrightarrow$
$wf\_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$
⟨*proof*⟩

lemma *join_restrict_table*:
assumes $table\ n\ A\ X\ table\ n\ B\ Y \neg\ b \Longrightarrow B \subseteq A$
shows $v \in join\ X\ b\ Y \longleftrightarrow$
$wf\_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$
⟨*proof*⟩

lemma *join_restrict_annotated*:
fixes $X\ Y :: 'a\ tuple\ set$
assumes $\neg\ b =simp=> B \subseteq A$
shows $join\ \{v.\ wf\_tuple\ n\ A\ v \wedge P\ v\}\ b\ \{v.\ wf\_tuple\ n\ B\ v \wedge Q\ v\} =$
$\{v.\ wf\_tuple\ n\ (A \cup B)\ v \wedge P\ (restrict\ A\ v) \wedge (if\ b\ then\ Q\ (restrict\ B\ v)\ else \neg\ Q\ (restrict\ B\ v))\}$
⟨*proof*⟩

lemma *in_joinI*: $table\ n\ A\ X \Longrightarrow table\ n\ B\ Y \Longrightarrow (\neg b \Longrightarrow B \subseteq A) \Longrightarrow wf\_tuple\ n\ (A \cup B)\ v \Longrightarrow$
$restrict\ A\ v \in X \Longrightarrow (b \Longrightarrow restrict\ B\ v \in Y) \Longrightarrow (\neg b \Longrightarrow restrict\ B\ v \notin Y) \Longrightarrow v \in join\ X\ b\ Y$
⟨*proof*⟩

lemma *in_joinE*: $v \in join\ X\ b\ Y \Longrightarrow table\ n\ A\ X \Longrightarrow table\ n\ B\ Y \Longrightarrow (\neg\ b \Longrightarrow B \subseteq A) \Longrightarrow$
$(wf\_tuple\ n\ (A \cup B)\ v \Longrightarrow restrict\ A\ v \in X \Longrightarrow if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

definition *qtable* :: $nat \Rightarrow nat\ set \Rightarrow ('a\ tuple \Rightarrow bool) \Rightarrow ('a\ tuple \Rightarrow bool) \Rightarrow$
$'a\ table \Rightarrow bool$ **where**
$qtable\ n\ A\ P\ Q\ X \longleftrightarrow table\ n\ A\ X \wedge (\forall x.\ (x \in X \wedge P\ x \longrightarrow Q\ x) \wedge (wf\_tuple\ n\ A\ x \wedge P\ x \wedge Q\ x \longrightarrow x \in X))$

abbreviation *wf_table* **where**
$wf\_table\ n\ A\ Q\ X \equiv qtable\ n\ A\ (\lambda\_.\ True)\ Q\ X$

lemma *wf_table_iff*: $wf\_table\ n\ A\ Q\ X \longleftrightarrow (\forall x.\ x \in X \longleftrightarrow (Q\ x \wedge wf\_tuple\ n\ A\ x))$
⟨*proof*⟩

lemma *table_wf_table*: $table\ n\ A\ X = wf\_table\ n\ A\ (\lambda v.\ v \in X)\ X$
⟨*proof*⟩

lemma *qtableI*: $table\ n\ A\ X \Longrightarrow$
$(\bigwedge x.\ x \in X \Longrightarrow wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow$

$(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow x \in X) \Longrightarrow$
$qtable\ n\ A\ P\ Q\ X$
⟨*proof*⟩

lemma *in_qtableI*: $qtable\ n\ A\ P\ Q\ X \Longrightarrow wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow x \in X$
⟨*proof*⟩

lemma *in_qtableE*: $qtable\ n\ A\ P\ Q\ X \Longrightarrow x \in X \Longrightarrow P\ x \Longrightarrow (wf\_tuple\ n\ A\ x \Longrightarrow Q\ x \Longrightarrow R) \Longrightarrow R$
⟨*proof*⟩

lemma *qtable_empty*: $(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow False) \Longrightarrow qtable\ n\ A\ P\ Q\ empty\_table$
⟨*proof*⟩

lemma *qtable_empty_iff*: $qtable\ n\ A\ P\ Q\ empty\_table = (\forall x.\ wf\_tuple\ n\ A\ x \longrightarrow P\ x \longrightarrow Q\ x \longrightarrow False)$
⟨*proof*⟩

lemma *qtable_unit_table*: $(\bigwedge x.\ wf\_tuple\ n\ \{\}\ x \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow qtable\ n\ \{\}\ P\ Q\ (unit\_table\ n)$
⟨*proof*⟩

lemma *qtable_union*: $qtable\ n\ A\ P\ Q1\ X \Longrightarrow qtable\ n\ A\ P\ Q2\ Y \Longrightarrow$
$(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ x \vee Q2\ x) \Longrightarrow qtable\ n\ A\ P\ Q\ (X \cup Y)$
⟨*proof*⟩

lemma *qtable_Union*: $finite\ I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow qtable\ n\ A\ P\ (Qi\ i)\ (Xi\ i)) \Longrightarrow$
$(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow (\exists i \in I.\ Qi\ i\ x)) \Longrightarrow qtable\ n\ A\ P\ Q\ (\bigcup i \in I.\ Xi\ i)$
⟨*proof*⟩

lemma *qtable_join*:
assumes $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y\ \neg\ b \Longrightarrow B \subseteq A\ C = A \cup B$
$\bigwedge x.\ wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$
$\bigwedge x.\ b \Longrightarrow wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge Q2\ (restrict\ B\ x)$
$\bigwedge x.\ \neg\ b \Longrightarrow wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge \neg\ Q2\ (restrict\ B\ x)$
shows $qtable\ n\ C\ P\ Q\ (join\ X\ b\ Y)$
⟨*proof*⟩

lemma *qtable_join_fixed*:
assumes $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y\ \neg\ b \Longrightarrow B \subseteq A\ C = A \cup B$
$\bigwedge x.\ wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$
shows $qtable\ n\ C\ P\ (\lambda x.\ Q1\ (restrict\ A\ x) \wedge (if\ b\ then\ Q2\ (restrict\ B\ x)\ else\ \neg\ Q2\ (restrict\ B\ x)))$
$(join\ X\ b\ Y)$
⟨*proof*⟩

lemma *wf_tuple_cong*:
assumes $wf\_tuple\ n\ A\ v\ wf\_tuple\ n\ A\ w\ \forall x \in A.\ map\ the\ v\ !\ x = map\ the\ w\ !\ x$
shows $v = w$
⟨*proof*⟩

definition $mem\_restr :: 'a\ list\ set \Rightarrow 'a\ tuple \Rightarrow bool$ **where**
$mem\_restr\ A\ x \longleftrightarrow (\exists y \in A.\ list\_all2\ (\lambda a\ b.\ a \neq None \longrightarrow a = Some\ b)\ x\ y)$

lemma *mem_restrI*: $y \in A \Longrightarrow length\ y = n \Longrightarrow wf\_tuple\ n\ V\ x \Longrightarrow \forall i \in V.\ x\ !\ i = Some\ (y\ !\ i) \Longrightarrow mem\_restr\ A\ x$
⟨*proof*⟩

lemma *mem_restrE*: $mem\_restr\ A\ x \Longrightarrow wf\_tuple\ n\ V\ x \Longrightarrow \forall i \in V.\ i < n \Longrightarrow$
$(\bigwedge y.\ y \in A \Longrightarrow \forall i \in V.\ x\ !\ i = Some\ (y\ !\ i) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *mem_restr_IntD*: *mem_restr* $(A \cap B)$ $v \Longrightarrow$ *mem_restr* A $v \wedge$ *mem_restr* B v
⟨*proof*⟩

lemma *mem_restr_Un_iff*: *mem_restr* $(A \cup B)$ $x \longleftrightarrow$ *mem_restr* A $x \vee$ *mem_restr* B x
⟨*proof*⟩

lemma *mem_restr_UNIV* [*simp*]: *mem_restr UNIV* x
⟨*proof*⟩

lemma *restrict_mem_restr*[*simp*]: *mem_restr* A $x \Longrightarrow$ *mem_restr* A (*restrict* V x)
⟨*proof*⟩

definition *lift_envs* :: *'a list set* ⇒ *'a list set* **where**
lift_envs $R = (\lambda(a,b).\ a \# b)$ ‘ $(UNIV \times R)$

lemma *lift_envs_mem_restr*[*simp*]: *mem_restr* A $x \Longrightarrow$ *mem_restr* (*lift_envs* A) $(a \# x)$
⟨*proof*⟩

lemma *qtable_project*:
assumes *qtable* (*Suc* n) A (*mem_restr* (*lift_envs* R)) P X
shows *qtable* n $((\lambda x.\ x - Suc\ 0)$ ‘ $(A - \{0\}))$ (*mem_restr* R)
$(\lambda v.\ \exists x.\ P\ ((if\ 0 \in A\ then\ Some\ x\ else\ None) \#\ v))$ (*tl* ‘ X)
(**is** *qtable* n *?A* (*mem_restr* R) *?P ?X*)
⟨*proof*⟩

lemma *qtable_cong*: *qtable* n A P Q $X \Longrightarrow A = B \Longrightarrow (\bigwedge v.\ P\ v \Longrightarrow Q\ v \longleftrightarrow Q'\ v) \Longrightarrow$ *qtable* n B P Q' X
⟨*proof*⟩

3 Abstract monitors and slicing

3.1 First-order specifications

We abstract from first-order trace specifications by referring only to their semantics. A first-order specification is described by a finite set of free variables and a satisfaction function that defines for every trace the pairs of valuations and time-points for which the specification is satisfied.

locale *fo_spec* =
fixes
nfv :: *nat* **and** *fv* :: *nat set* **and**
sat :: *'a trace* ⇒ *'b list* ⇒ *nat* ⇒ *bool*
assumes
fv_less_nfv: $x \in fv \Longrightarrow x < nfv$ **and**
sat_fv_cong: $(\bigwedge x.\ x \in fv \Longrightarrow v!x = v'!x) \Longrightarrow sat\ \sigma\ v\ i = sat\ \sigma\ v'\ i$
begin

definition *verdicts* :: *'a trace* ⇒ (*nat* × *'b tuple*) *set* **where**
verdicts $\sigma = \{(i,\ v).\ wf\_tuple\ nfv\ fv\ v \wedge sat\ \sigma\ (map\ the\ v)\ i\}$

end

We usually employ a monitor to find the *violations* of a specification. That is, the monitor should output the satisfactions of its negation. Moreover, all monitor implementations must work with finite prefixes. We are therefore interested in co-safety properties, which allow us to identify all satisfactions on finite prefixes.

locale *cosafety_fo_spec* = *fo_spec* +
assumes *cosafety_lr*: $sat\ \sigma\ v\ i \Longrightarrow \exists \pi.$ *prefix_of* $\pi\ \sigma \wedge (\forall \sigma'.$ *prefix_of* $\pi\ \sigma' \longrightarrow sat\ \sigma'\ v\ i)$

begin

lemma *cosafety*: *sat* σ *v i* $\longleftrightarrow$ ($\exists\,\pi$. *prefix_of* π σ $\wedge$ ($\forall\,\sigma'$. *prefix_of* π σ' $\longrightarrow$ *sat* σ' *v i*))
⟨*proof*⟩

end

3.2 Monitor function

We model monitors abstractly as functions from prefixes to verdict sets. The following locale specifies a minimal set of properties that any reasonable monitor should have.

locale *monitor* = *fo_spec* +
fixes *M* :: *'a prefix* ⇒ (*nat* × *'b tuple*) *set*
assumes
mono_monitor: $\pi \leq \pi' \Longrightarrow M\ \pi \subseteq M\ \pi'$ **and**
wf_monitor: (*i*, *v*) ∈ *M* π ⟹ *wf_tuple nfv fv v* **and**
sound_monitor: (*i*, *v*) ∈ *M* π ⟹ *prefix_of* π σ ⟹ *sat* σ (*map the v*) *i* **and**
complete_monitor: *prefix_of* π σ ⟹ *wf_tuple nfv fv v* ⟹
($\bigwedge\sigma$. *prefix_of* π σ ⟹ *sat* σ (*map the v*) *i*) ⟹ $\exists\,\pi'$. *prefix_of* π' σ $\wedge$ (*i*, *v*) ∈ *M* π'

A monitor for a co-safety specification computes precisely the set of all satisfactions in the limit:

abbreviation (**in** *monitor*) *M_limit* $\sigma \equiv \bigcup\{M\ \pi \mid \pi.\ \textit{prefix\_of}\ \pi\ \sigma\}$

locale *cosafety_monitor* = *cosafety_fo_spec* + *monitor*
begin

lemma *M_limit_eq*: *M_limit* σ = *verdicts* σ
⟨*proof*⟩

end

The monitor function *M* adds some information over *sat*, namely when a verdict is output. One possible behavior is that the monitor outputs its verdicts for one time-point at a time, in increasing order of time-points. Then *M* is uniquely defined by a progress function, which returns for every prefix the time-point up to which all verdicts are computed.

locale *progress* = *fo_spec* _ _ *sat* **for** *sat* :: *'a trace* ⇒ *'b list* ⇒ *nat* ⇒ *bool* +
fixes *progress* :: *'a prefix* ⇒ *nat*
assumes
progress_mono: $\pi \leq \pi' \Longrightarrow$ *progress* $\pi \leq$ *progress* π' **and**
ex_progress_ge: $\exists\,\pi$. *prefix_of* π σ $\wedge$ *x* ≤ *progress* π **and**
progress_sat_cong: *prefix_of* π σ ⟹ *prefix_of* π σ' ⟹ *i* < *progress* π ⟹
sat σ *v i* $\longleftrightarrow$ *sat* σ' *v i*
— The last condition is not necessary to obtain a proper monitor function. However, it corresponds to the intuitive understanding of monitor progress, and it results in a stronger characterisation. In particular, it implies that the specification is co-safety, as we will show below.
begin

definition *M* :: *'a prefix* ⇒ (*nat* × *'b tuple*) *set* **where**
M π = {(*i*, *v*). *i* < *progress* π $\wedge$ *wf_tuple nfv fv v* $\wedge$
($\forall\,\sigma$. *prefix_of* π σ $\longrightarrow$ *sat* σ (*map the v*) *i*)}

lemma *M_alt*: *M* π = {(*i*, *v*). *i* < *progress* π $\wedge$ *wf_tuple nfv fv v* $\wedge$
($\exists\,\sigma$. *prefix_of* π σ $\wedge$ *sat* σ (*map the v*) *i*)}
⟨*proof*⟩

end

sublocale *progress* $\subseteq$ *cosafety_monitor* _ _ _ *M*
⟨*proof*⟩

3.3 Slicing

Sliceable specifications can be evaluated meaningfully on a subset of events.

locale *abstract_slicer* =
  fixes *relevant_events* :: *'b list set* ⇒ *'a set*
begin

abbreviation *slice* :: *'b list set* ⇒ *'a trace* ⇒ *'a trace* **where**
  slice S ≡ *map_Γ* (λ*D*. *D* ∩ *relevant_events S*)

abbreviation *pslice* :: *'b list set* ⇒ *'a prefix* ⇒ *'a prefix* **where**
  pslice S ≡ *pmap_Γ* (λ*D*. *D* ∩ *relevant_events S*)

lemma *prefix_of_psliceI*: *prefix_of* π σ ⟹ *prefix_of* (*pslice S* π) (*slice S* σ)
  ⟨*proof*⟩

lemma *plen_pslice*[*simp*]: *plen* (*pslice S* π) = *plen* π
  ⟨*proof*⟩

lemma *pslice_pnil*[*simp*]: *pslice S pnil* = *pnil*
  ⟨*proof*⟩

lemma *last_ts_pslice*[*simp*]: *last_ts* (*pslice S* π) = *last_ts* π
  ⟨*proof*⟩

abbreviation *verdict_filter* :: *'b list set* ⇒ (*nat* × *'b tuple*) *set* ⇒ (*nat* × *'b tuple*) *set* **where**
  verdict_filter S V ≡ {(*i*, *v*) ∈ *V*. *mem_restr S v*}

end

locale *sliceable_fo_spec* = *fo_spec* _ _ *sat* + *abstract_slicer relevant_events*
  for *relevant_events* :: *'b list set* ⇒ *'a set* **and** *sat* :: *'a trace* ⇒ *'b list* ⇒ *nat* ⇒ *bool* +
  assumes *sliceable*: *v* ∈ *S* ⟹ *sat* (*slice S* σ) *v i* ⟷ *sat* σ *v i*
begin

lemma *union_verdicts_slice*:
  assumes *part*: ⋃ 𝒮 = *UNIV*
  shows ⋃ ((λ*S*. *verdict_filter S* (*verdicts* (*slice S* σ))) ‘ 𝒮) = *verdicts* σ
⟨*proof*⟩

end

We define a similar notion for monitors. It is potentially stronger because the time-point at which verdicts are output must not change.

locale *sliceable_monitor* = *monitor* _ _ *sat M* + *abstract_slicer relevant_events*
  for *relevant_events* :: *'b list set* ⇒ *'a set* **and** *sat* :: *'a trace* ⇒ *'b list* ⇒ *nat* ⇒ *bool* **and** *M* +
  assumes *sliceable_M*: *mem_restr S v* ⟹ (*i*, *v*) ∈ *M* (*pslice S* π) ⟷ (*i*, *v*) ∈ *M* π
begin

lemma *union_M_pslice*:
  assumes *part*: ⋃ 𝒮 = *UNIV*
  shows ⋃ ((λ*S*. *verdict_filter S* (*M* (*pslice S* π))) ‘ 𝒮) = *M* π
⟨*proof*⟩

end

If the specification is sliceable and the monitor's progress depends only on time-stamps, then also the monitor itself is sliceable.

locale *timed_progress* = *progress* +
  assumes *progress_time_conv*: $pts\ \pi = pts\ \pi' \Longrightarrow progress\ \pi = progress\ \pi'$

locale *sliceable_timed_progress* = *sliceable_fo_spec* + *timed_progress*
begin

lemma *progress_pslice*[*simp*]: *progress* (*pslice S* π) = *progress* π
  ⟨*proof*⟩

end

sublocale *sliceable_timed_progress* ⊆ *sliceable_monitor* _ _ _ _ *M*
⟨*proof*⟩

4 Intervals

typedef $\mathcal{I}$ = {(*i* :: *nat*, *j* :: *enat*). $i \leq j$}
  ⟨*proof*⟩

setup_lifting *type_definition_*$\mathcal{I}$

instantiation $\mathcal{I}$ :: *equal* **begin**
lift_definition *equal_*$\mathcal{I}$ :: $\mathcal{I} \Rightarrow \mathcal{I} \Rightarrow$ *bool* **is** (=) ⟨*proof*⟩
instance ⟨*proof*⟩
end

instantiation $\mathcal{I}$ :: *linorder* **begin**
lift_definition *less_eq_*$\mathcal{I}$ :: $\mathcal{I} \Rightarrow \mathcal{I} \Rightarrow$ *bool* **is** ($\leq$) ⟨*proof*⟩
lift_definition *less_*$\mathcal{I}$ :: $\mathcal{I} \Rightarrow \mathcal{I} \Rightarrow$ *bool* **is** (<) ⟨*proof*⟩
instance ⟨*proof*⟩
end

lift_definition *all* :: $\mathcal{I}$ **is** (*0*, ∞) ⟨*proof*⟩
lift_definition *left* :: $\mathcal{I} \Rightarrow$ *nat* **is** *fst* ⟨*proof*⟩
lift_definition *right* :: $\mathcal{I} \Rightarrow$ *enat* **is** *snd* ⟨*proof*⟩
lift_definition *point* :: *nat* $\Rightarrow$ $\mathcal{I}$ **is** λn. (*n*, *enat n*) ⟨*proof*⟩
lift_definition *init* :: *nat* $\Rightarrow$ $\mathcal{I}$ **is** λn. (*0*, *enat n*) ⟨*proof*⟩
abbreviation *mem* **where** *mem n I* ≡ (*left I* $\leq$ *n* $\wedge$ *n* $\leq$ *right I*)
lift_definition *subtract* :: *nat* $\Rightarrow$ $\mathcal{I}$ $\Rightarrow$ $\mathcal{I}$ **is**
  λn (*i*, *j*). (*i* − *n*, *j* − *enat n*) ⟨*proof*⟩
lift_definition *add* :: *nat* $\Rightarrow$ $\mathcal{I}$ $\Rightarrow$ $\mathcal{I}$ **is**
  λn (*a*, *b*). (*a*, *b* + *enat n*) ⟨*proof*⟩

lemma *left_right*: *left I* $\leq$ *right I*
  ⟨*proof*⟩

lemma *point_simps*[*simp*]:
  left (*point n*) = *n*
  right (*point n*) = *n*
  ⟨*proof*⟩

lemma *init_simps*[*simp*]:
  left (*init n*) = *0*

right (*init n*) = *n*
⟨*proof*⟩

lemma *subtract_simps*[*simp*]:
left (*subtract n I*) = *left I* − *n*
right (*subtract n I*) = *right I* − *n*
subtract 0 I = *I*
subtract x (*point y*) = *point* (*y* − *x*)
⟨*proof*⟩

definition *shifted* :: *ℐ* ⇒ *ℐ set* **where**
shifted I = (*λn. subtract n I*) ‘ {*0* .. (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*)}

lemma *subtract_too_much*: *i* > (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*) ⟹
subtract i I = *subtract* (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*) *I*
⟨*proof*⟩

lemma *subtract_shifted*: *subtract n I* ∈ *shifted I*
⟨*proof*⟩

lemma *finite_shifted*: *finite* (*shifted I*)
⟨*proof*⟩

definition *interval* :: *nat* ⇒ *enat* ⇒ *ℐ* **where**
interval l r = (*if l* ≤ *r then Abs_ℐ* (*l*, *r*) *else undefined*)

lemma [*code abstract*]: *Rep_ℐ* (*interval l r*) = (*if l* ≤ *r then* (*l*, *r*) *else Rep_ℐ undefined*)
⟨*proof*⟩

5 Metric first-order temporal logic

context begin

5.1 Formulas and satisfiability

qualified type_synonym *name* = *string*
qualified type_synonym *'a event* = (*name* × *'a list*)
qualified type_synonym *'a database* = *'a event set*
qualified type_synonym *'a prefix* = (*name* × *'a list*) *prefix*
qualified type_synonym *'a trace* = (*name* × *'a list*) *trace*

qualified type_synonym *'a env* = *'a list*

qualified datatype *'a trm* = *Var nat* | *is_Const*: *Const 'a*

qualified primrec *fvi_trm* :: *nat* ⇒ *'a trm* ⇒ *nat set* **where**
fvi_trm b (*Var x*) = (*if b* ≤ *x then* {*x* − *b*} *else* {})
| *fvi_trm b* (*Const _*) = {}

abbreviation *fv_trm* ≡ *fvi_trm 0*

qualified primrec *eval_trm* :: *'a env* ⇒ *'a trm* ⇒ *'a* **where**
eval_trm v (*Var x*) = *v* ! *x*
| *eval_trm v* (*Const x*) = *x*

lemma *eval_trm_cong*: ∀ *x*∈*fv_trm t. v* ! *x* = *v'* ! *x* ⟹ *eval_trm v t* = *eval_trm v' t*
⟨*proof*⟩ **datatype** (*discs_sels*) *'a formula* = *Pred name 'a trm list* | *Eq 'a trm 'a trm*
| *Neg 'a formula* | *Or 'a formula 'a formula* | *Exists 'a formula*

| *Prev* $\mathcal{I}$ *'a formula* | *Next* $\mathcal{I}$ *'a formula*
| *Since 'a formula* $\mathcal{I}$ *'a formula* | *Until 'a formula* $\mathcal{I}$ *'a formula*

qualified primrec *fvi* :: *nat* ⇒ *'a formula* ⇒ *nat set* **where**
fvi b (*Pred r ts*) = (⋃ *t*∈*set ts*. *fvi_trm b t*)
| *fvi b* (*Eq t1 t2*) = *fvi_trm b t1* ∪ *fvi_trm b t2*
| *fvi b* (*Neg* φ) = *fvi b* φ
| *fvi b* (*Or* φ ψ) = *fvi b* φ ∪ *fvi b* ψ
| *fvi b* (*Exists* φ) = *fvi* (*Suc b*) φ
| *fvi b* (*Prev I* φ) = *fvi b* φ
| *fvi b* (*Next I* φ) = *fvi b* φ
| *fvi b* (*Since* φ *I* ψ) = *fvi b* φ ∪ *fvi b* ψ
| *fvi b* (*Until* φ *I* ψ) = *fvi b* φ ∪ *fvi b* ψ

abbreviation *fv* ≡ *fvi 0*

lemma *finite_fvi_trm*[*simp*]: *finite* (*fvi_trm b t*)
⟨*proof*⟩

lemma *finite_fvi*[*simp*]: *finite* (*fvi b* φ)
⟨*proof*⟩

lemma *fvi_trm_Suc*: *x* ∈ *fvi_trm* (*Suc b*) *t* ⟷ *Suc x* ∈ *fvi_trm b t*
⟨*proof*⟩

lemma *fvi_Suc*: *x* ∈ *fvi* (*Suc b*) φ ⟷ *Suc x* ∈ *fvi b* φ
⟨*proof*⟩

lemma *fvi_Suc_bound*:
assumes ∀ *i*∈*fvi* (*Suc b*) φ. *i* < *n*
shows ∀ *i*∈*fvi b* φ. *i* < *Suc n*
⟨*proof*⟩ **definition** *nfv* :: *'a formula* ⇒ *nat* **where**
nfv φ = *Max* (*insert 0* (*Suc* ‘ *fv* φ))

qualified definition *envs* :: *'a formula* ⇒ *'a env set* **where**
envs φ = {*v*. *length v* = *nfv* φ}

lemma *nfv_simps*[*simp*]:
nfv (*Neg* φ) = *nfv* φ
nfv (*Or* φ ψ) = *max* (*nfv* φ) (*nfv* ψ)
nfv (*Prev I* φ) = *nfv* φ
nfv (*Next I* φ) = *nfv* φ
nfv (*Since* φ *I* ψ) = *max* (*nfv* φ) (*nfv* ψ)
nfv (*Until* φ *I* ψ) = *max* (*nfv* φ) (*nfv* ψ)
⟨*proof*⟩

lemma *fvi_less_nfv*: ∀ *i*∈*fv* φ. *i* < *nfv* φ
⟨*proof*⟩ **primrec** *future_reach* :: *'a formula* ⇒ *enat* **where**
future_reach (*Pred* _ _) = *0*
| *future_reach* (*Eq* _ _) = *0*
| *future_reach* (*Neg* φ) = *future_reach* φ
| *future_reach* (*Or* φ ψ) = *max* (*future_reach* φ) (*future_reach* ψ)
| *future_reach* (*Exists* φ) = *future_reach* φ
| *future_reach* (*Prev I* φ) = *future_reach* φ − *left I*
| *future_reach* (*Next I* φ) = *future_reach* φ + *right I* + *1*
| *future_reach* (*Since* φ *I* ψ) = *max* (*future_reach* φ) (*future_reach* ψ − *left I*)
| *future_reach* (*Until* φ *I* ψ) = *max* (*future_reach* φ) (*future_reach* ψ) + *right I* + *1*

qualified primrec *sat* :: *'a trace* ⇒ *'a env* ⇒ *nat* ⇒ *'a formula* ⇒ *bool* **where**
  sat σ v i (*Pred r ts*) = ((*r*, *map* (*eval_trm v*) *ts*) ∈ Γ *σ i*)
| *sat σ v i* (*Eq t1 t2*) = (*eval_trm v t1* = *eval_trm v t2*)
| *sat σ v i* (*Neg φ*) = (¬ *sat σ v i φ*)
| *sat σ v i* (*Or φ ψ*) = (*sat σ v i φ* ∨ *sat σ v i ψ*)
| *sat σ v i* (*Exists φ*) = (∃ *z*. *sat σ* (*z* # *v*) *i φ*)
| *sat σ v i* (*Prev I φ*) = (*case i of* 0 ⇒ *False* | *Suc j* ⇒ *mem* (*τ σ i* − *τ σ j*) *I* ∧ *sat σ v j φ*)
| *sat σ v i* (*Next I φ*) = (*mem* (*τ σ* (*Suc i*) − *τ σ i*) *I* ∧ *sat σ v* (*Suc i*) *φ*)
| *sat σ v i* (*Since φ I ψ*) = (∃ *j*≤*i*. *mem* (*τ σ i* − *τ σ j*) *I* ∧ *sat σ v j ψ* ∧ (∀ *k* ∈ {*j* <.. *i*}. *sat σ v k φ*))
| *sat σ v i* (*Until φ I ψ*) = (∃ *j*≥*i*. *mem* (*τ σ j* − *τ σ i*) *I* ∧ *sat σ v j ψ* ∧ (∀ *k* ∈ {*i* ..< *j*}. *sat σ v k φ*))

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

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

lemma *sat_Since_0*: *sat σ v* 0 (*Since φ I ψ*) ⟷ *mem* 0 *I* ∧ *sat σ v* 0 *ψ*
  ⟨*proof*⟩

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

lemma *sat_Since_pointD*: *sat σ v i* (*Since φ* (*point t*) *ψ*) ⟹ *mem t I* ⟹ *sat σ v i* (*Since φ I ψ*)
  ⟨*proof*⟩

lemma *eval_trm_fvi_cong*: ∀ *x*∈*fv_trm t*. *v*!*x* = *v'*!*x* ⟹ *eval_trm v t* = *eval_trm v' t*
  ⟨*proof*⟩

lemma *sat_fvi_cong*: ∀ *x*∈*fv φ*. *v*!*x* = *v'*!*x* ⟹ *sat σ v i φ* = *sat σ v' i φ*
⟨*proof*⟩

5.2 Defined connectives

qualified definition *And φ ψ* = *Neg* (*Or* (*Neg φ*) (*Neg ψ*))

lemma *fvi_And*: *fvi b* (*And φ ψ*) = *fvi b φ* ∪ *fvi b ψ*
  ⟨*proof*⟩

lemma *nfv_And*[*simp*]: *nfv* (*And φ ψ*) = *max* (*nfv φ*) (*nfv ψ*)
  ⟨*proof*⟩

lemma *future_reach_And*: *future_reach* (*And φ ψ*) = *max* (*future_reach φ*) (*future_reach ψ*)
  ⟨*proof*⟩

lemma *sat_And*: *sat σ v i* (*And φ ψ*) = (*sat σ v i φ* ∧ *sat σ v i ψ*)
  ⟨*proof*⟩ **definition** *And_Not φ ψ* = *Neg* (*Or* (*Neg φ*) *ψ*)

lemma *fvi_And_Not*: *fvi b* (*And_Not φ ψ*) = *fvi b φ* ∪ *fvi b ψ*
  ⟨*proof*⟩

lemma *nfv_And_Not*[*simp*]: *nfv* (*And_Not* φ ψ) = *max* (*nfv* φ) (*nfv* ψ)
⟨*proof*⟩

lemma *future_reach_And_Not*: *future_reach* (*And_Not* φ ψ) = *max* (*future_reach* φ) (*future_reach* ψ)
⟨*proof*⟩

lemma *sat_And_Not*: *sat* σ *v i* (*And_Not* φ ψ) = (*sat* σ *v i* φ ∧ ¬ *sat* σ *v i* ψ)
⟨*proof*⟩

5.3 Safe formulas

fun *safe_formula* :: *'a MFOTL.formula* ⇒ *bool* **where**
safe_formula (*MFOTL.Eq t1 t2*) = (*MFOTL.is_Const t1* ∨ *MFOTL.is_Const t2*)
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Const x*) (*MFOTL.Const y*))) = *True*
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Var x*) (*MFOTL.Var y*))) = (*x* = *y*)
| *safe_formula* (*MFOTL.Pred e ts*) = *True*
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Or* (*MFOTL.Neg* φ) ψ)) = (*safe_formula* φ ∧
(*safe_formula* ψ ∧ *MFOTL.fv* ψ ⊆ *MFOTL.fv* φ ∨ (*case* ψ *of MFOTL.Neg* ψ' ⇒ *safe_formula* ψ' | _ ⇒ *False*)))
| *safe_formula* (*MFOTL.Or* φ ψ) = (*MFOTL.fv* ψ = *MFOTL.fv* φ ∧ *safe_formula* φ ∧ *safe_formula* ψ)
| *safe_formula* (*MFOTL.Exists* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Prev I* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Next I* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Since* φ *I* ψ) = (*MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ∧
(*safe_formula* φ ∨ (*case* φ *of MFOTL.Neg* φ' ⇒ *safe_formula* φ' | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* (*MFOTL.Until* φ *I* ψ) = (*MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ∧
(*safe_formula* φ ∨ (*case* φ *of MFOTL.Neg* φ' ⇒ *safe_formula* φ' | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* _ = *False*

lemma *disjE_Not2*: *P* ∨ *Q* ⟹ (*P* ⟹ *R*) ⟹ (¬*P* ⟹ *Q* ⟹ *R*) ⟹ *R*
⟨*proof*⟩

lemma *safe_formula_induct*[*consumes 1*]:
assumes *safe_formula* φ
and ⋀*t1 t2*. *MFOTL.is_Const t1* ⟹ *P* (*MFOTL.Eq t1 t2*)
and ⋀*t1 t2*. *MFOTL.is_Const t2* ⟹ *P* (*MFOTL.Eq t1 t2*)
and ⋀*x y*. *P* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Const x*) (*MFOTL.Const y*)))
and ⋀*x y*. *x* = *y* ⟹ *P* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Var x*) (*MFOTL.Var y*)))
and ⋀*e ts*. *P* (*MFOTL.Pred e ts*)
and ⋀φ ψ. ¬ (*safe_formula* (*MFOTL.Neg* ψ) ∧ *MFOTL.fv* ψ ⊆ *MFOTL.fv* φ) ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.And* φ ψ)
and ⋀φ ψ. *safe_formula* ψ ⟹ *MFOTL.fv* ψ ⊆ *MFOTL.fv* φ ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.And_Not* φ ψ)
and ⋀φ ψ. *MFOTL.fv* ψ = *MFOTL.fv* φ ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.Or* φ ψ)
and ⋀φ. *P* φ ⟹ *P* (*MFOTL.Exists* φ)
and ⋀*I* φ. *P* φ ⟹ *P* (*MFOTL.Prev I* φ)
and ⋀*I* φ. *P* φ ⟹ *P* (*MFOTL.Next I* φ)
and ⋀φ *I* ψ. *MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ⟹ *safe_formula* φ ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.Since* φ *I* ψ)
and ⋀φ *I* ψ. *MFOTL.fv* (*MFOTL.Neg* φ) ⊆ *MFOTL.fv* ψ ⟹
¬ *safe_formula* (*MFOTL.Neg* φ) ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.Since* (*MFOTL.Neg* φ) *I* ψ)
and ⋀φ *I* ψ. *MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ⟹ *safe_formula* φ ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.Until* φ *I* ψ)
and ⋀φ *I* ψ. *MFOTL.fv* (*MFOTL.Neg* φ) ⊆ *MFOTL.fv* ψ ⟹
¬ *safe_formula* (*MFOTL.Neg* φ) ⟹ *P* φ ⟹ *P* ψ ⟹ *P* (*MFOTL.Until* (*MFOTL.Neg* φ) *I* ψ)

shows $P\ \varphi$
⟨*proof*⟩

5.4 Slicing traces

qualified primrec *matches* :: $'a\ env \Rightarrow 'a\ formula \Rightarrow name \times 'a\ list \Rightarrow bool$ **where**
matches v (Pred r ts) e = (r = fst e ∧ map (eval_trm v) ts = snd e)
| *matches v (Eq _ _) e = False*
| *matches v (Neg φ) e = matches v φ e*
| *matches v (Or φ ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (Exists φ) e = (∃ z. matches (z # v) φ e)*
| *matches v (Prev I φ) e = matches v φ e*
| *matches v (Next I φ) e = matches v φ e*
| *matches v (Since φ I ψ) e = (matches v φ e ∨ matches v ψ e)*
| *matches v (Until φ I ψ) e = (matches v φ e ∨ matches v ψ e)*

lemma *matches_fvi_cong*: $\forall x{\in}fv\ \varphi.\ v!x = v'!x \Longrightarrow matches\ v\ \varphi\ e = matches\ v'\ \varphi\ e$
⟨*proof*⟩

abbreviation *relevant_events* **where** $relevant\_events\ \varphi\ S \equiv \{e.\ S \cap \{v.\ matches\ v\ \varphi\ e\} \neq \{\}\}$

lemma *sat_slice_strong*: $relevant\_events\ \varphi\ S \subseteq E \Longrightarrow v \in S \Longrightarrow$
sat σ v i φ ⟷ sat (map_Γ (λD. D ∩ E) σ) v i φ
⟨*proof*⟩

end

interpretation *MFOTL_slicer*: *abstract_slicer relevant_events φ* **for** *φ* ⟨*proof*⟩

lemma *sat_slice_iff*:
assumes $v \in S$
shows $MFOTL.sat\ \sigma\ v\ i\ \varphi \longleftrightarrow MFOTL.sat\ (MFOTL\_slicer.slice\ \varphi\ S\ \sigma)\ v\ i\ \varphi$
⟨*proof*⟩

lemma *slice_replace_prefix*:
$prefix\_of\ (MFOTL\_slicer.pslice\ \varphi\ R\ \pi)\ \sigma \Longrightarrow$
$MFOTL\_slicer.slice\ \varphi\ R\ (replace\_prefix\ \pi\ \sigma) = MFOTL\_slicer.slice\ \varphi\ R\ \sigma$
⟨*proof*⟩

6 Monitor implementation

6.1 Monitorable formulas

definition $mmonitorable\ \varphi \longleftrightarrow safe\_formula\ \varphi \wedge MFOTL.future\_reach\ \varphi \neq \infty$

fun $mmonitorable\_exec :: 'a\ MFOTL.formula \Rightarrow bool$ **where**
mmonitorable_exec (MFOTL.Eq t1 t2) = (MFOTL.is_Const t1 ∨ MFOTL.is_Const t2)
| *mmonitorable_exec (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y))) = True*
| *mmonitorable_exec (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y))) = (x = y)*
| *mmonitorable_exec (MFOTL.Pred e ts) = True*
| *mmonitorable_exec (MFOTL.Neg (MFOTL.Or (MFOTL.Neg φ) ψ)) = (mmonitorable_exec φ ∧*
(mmonitorable_exec ψ ∧ MFOTL.fv ψ ⊆ MFOTL.fv φ ∨ (case ψ of MFOTL.Neg ψ' ⇒ mmonitorable_exec ψ' | _ ⇒ False)))
| *mmonitorable_exec (MFOTL.Or φ ψ) = (MFOTL.fv ψ = MFOTL.fv φ ∧ mmonitorable_exec φ ∧ mmonitorable_exec ψ)*
| *mmonitorable_exec (MFOTL.Exists φ) = (mmonitorable_exec φ)*
| *mmonitorable_exec (MFOTL.Prev I φ) = (mmonitorable_exec φ)*
| *mmonitorable_exec (MFOTL.Next I φ) = (mmonitorable_exec φ ∧ right I ≠ ∞)*

| *mmonitorable_exec* (*MFOTL.Since* φ *I* ψ) = (*MFOTL.fv* $\varphi \subseteq$ *MFOTL.fv* $\psi \wedge$
(*mmonitorable_exec* $\varphi \vee$ (*case* φ *of MFOTL.Neg* $\varphi' \Rightarrow$ *mmonitorable_exec* φ' | _ $\Rightarrow$ *False*)) $\wedge$
mmonitorable_exec ψ)
| *mmonitorable_exec* (*MFOTL.Until* φ *I* ψ) = (*MFOTL.fv* $\varphi \subseteq$ *MFOTL.fv* $\psi \wedge$ *right I* $\neq \infty \wedge$
(*mmonitorable_exec* $\varphi \vee$ (*case* φ *of MFOTL.Neg* $\varphi' \Rightarrow$ *mmonitorable_exec* φ' | _ $\Rightarrow$ *False*)) $\wedge$
mmonitorable_exec ψ)
| *mmonitorable_exec* _ = *False*

lemma *plus_eq_enat_iff*: $a + b =$ *enat* $i \longleftrightarrow (\exists j\ k.\ a =$ *enat* $j \wedge b =$ *enat* $k \wedge j + k = i)$
⟨*proof*⟩

lemma *minus_eq_enat_iff*: $a -$ *enat* $k =$ *enat* $i \longleftrightarrow (\exists j.\ a =$ *enat* $j \wedge j - k = i)$
⟨*proof*⟩

lemma *safe_formula_mmonitorable_exec*: *safe_formula* $\varphi \Longrightarrow$ *MFOTL.future_reach* $\varphi \neq \infty \Longrightarrow$ *mmonitorable_exec* φ
⟨*proof*⟩

lemma *mmonitorable_exec_mmonitorable*: *mmonitorable_exec* $\varphi \Longrightarrow$ *mmonitorable* φ
⟨*proof*⟩

lemma *monitorable_formula_code*[*code*]: *mmonitorable* $\varphi =$ *mmonitorable_exec* φ
⟨*proof*⟩

6.2 The executable monitor

type_synonym *ts* = *nat*

type_synonym *'a mbuf2* = *'a table list* × *'a table list*
type_synonym *'a msaux* = (*ts* × *'a table*) *list*
type_synonym *'a muaux* = (*ts* × *'a table* × *'a table*) *list*

datatype *'a mformula* =
MRel 'a table
| *MPred MFOTL.name 'a MFOTL.trm list*
| *MAnd 'a mformula bool 'a mformula 'a mbuf2*
| *MOr 'a mformula 'a mformula 'a mbuf2*
| *MExists 'a mformula*
| *MPrev* $\mathcal{I}$ *'a mformula bool 'a table list ts list*
| *MNext* $\mathcal{I}$ *'a mformula bool ts list*
| *MSince bool 'a mformula* $\mathcal{I}$ *'a mformula 'a mbuf2 ts list 'a msaux*
| *MUntil bool 'a mformula* $\mathcal{I}$ *'a mformula 'a mbuf2 ts list 'a muaux*

record *'a mstate* =
mstate_i :: *nat*
mstate_m :: *'a mformula*
mstate_n :: *nat*

fun *eq_rel* :: *nat* ⇒ *'a MFOTL.trm* ⇒ *'a MFOTL.trm* ⇒ *'a table* **where**
eq_rel n (*MFOTL.Const x*) (*MFOTL.Const y*) = (*if x = y then unit_table n else empty_table*)
| *eq_rel n* (*MFOTL.Var x*) (*MFOTL.Const y*) = *singleton_table n x y*
| *eq_rel n* (*MFOTL.Const x*) (*MFOTL.Var y*) = *singleton_table n y x*
| *eq_rel n* (*MFOTL.Var x*) (*MFOTL.Var y*) = *undefined*

fun *neq_rel* :: *nat* ⇒ *'a MFOTL.trm* ⇒ *'a MFOTL.trm* ⇒ *'a table* **where**
neq_rel n (*MFOTL.Const x*) (*MFOTL.Const y*) = (*if x = y then empty_table else unit_table n*)
| *neq_rel n* (*MFOTL.Var x*) (*MFOTL.Var y*) = (*if x = y then empty_table else undefined*)
| *neq_rel* _ _ _ = *undefined*

```
fun minit0 :: nat ⇒ 'a MFOTL.formula ⇒ 'a mformula where
  minit0 n (MFOTL.Neg φ) = (case φ of
    MFOTL.Eq t1 t2 ⇒ MRel (neq_rel n t1 t2)
  | MFOTL.Or (MFOTL.Neg φ) ψ ⇒ (if safe_formula ψ ∧ MFOTL.fv ψ ⊆ MFOTL.fv φ
      then MAnd (minit0 n φ) False (minit0 n ψ) ([], [])
      else (case ψ of MFOTL.Neg ψ ⇒ MAnd (minit0 n φ) True (minit0 n ψ) ([], []) | _ ⇒ undefined))
  | _ ⇒ undefined)
| minit0 n (MFOTL.Eq t1 t2) = MRel (eq_rel n t1 t2)
| minit0 n (MFOTL.Pred e ts) = MPred e ts
| minit0 n (MFOTL.Or φ ψ) = MOr (minit0 n φ) (minit0 n ψ) ([], [])
| minit0 n (MFOTL.Exists φ) = MExists (minit0 (Suc n) φ)
| minit0 n (MFOTL.Prev I φ) = MPrev I (minit0 n φ) True [] []
| minit0 n (MFOTL.Next I φ) = MNext I (minit0 n φ) True []
| minit0 n (MFOTL.Since φ I ψ) = (if safe_formula φ
    then MSince True (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    else (case φ of
      MFOTL.Neg φ ⇒ MSince False (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    | _ ⇒ undefined))
| minit0 n (MFOTL.Until φ I ψ) = (if safe_formula φ
    then MUntil True (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    else (case φ of
      MFOTL.Neg φ ⇒ MUntil False (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    | _ ⇒ undefined))

definition minit :: 'a MFOTL.formula ⇒ 'a mstate where
  minit φ = (let n = MFOTL.nfv φ in (|mstate_i = 0, mstate_m = minit0 n φ, mstate_n = n|))

fun mprev_next :: ℐ ⇒ 'a table list ⇒ ts list ⇒ 'a table list × 'a table list × ts list where
  mprev_next I [] ts = ([], [], ts)
| mprev_next I xs [] = ([], xs, [])
| mprev_next I xs [t] = ([], xs, [t])
| mprev_next I (x # xs) (t # t' # ts) = (let (ys, zs) = mprev_next I xs (t' # ts)
    in ((if mem (t' − t) I then x else empty_table) # ys, zs))

fun mbuf2_add :: 'a table list ⇒ 'a table list ⇒ 'a mbuf2 ⇒ 'a mbuf2 where
 mbuf2_add xs' ys' (xs, ys) = (xs @ xs', ys @ ys')

fun mbuf2_take :: ('a table ⇒ 'a table ⇒ 'b) ⇒ 'a mbuf2 ⇒ 'b list × 'a mbuf2 where
  mbuf2_take f (x # xs, y # ys) = (let (zs, buf) = mbuf2_take f (xs, ys) in (f x y # zs, buf))
| mbuf2_take f (xs, ys) = ([], (xs, ys))

fun mbuf2t_take :: ('a table ⇒ 'a table ⇒ ts ⇒ 'b ⇒ 'b) ⇒ 'b ⇒
  'a mbuf2 ⇒ ts list ⇒ 'b × 'a mbuf2 × ts list where
  mbuf2t_take f z (x # xs, y # ys) (t # ts) = mbuf2t_take f (f x y t z) (xs, ys) ts
| mbuf2t_take f z (xs, ys) ts = (z, (xs, ys), ts)

fun match :: 'a MFOTL.trm list ⇒ 'a list ⇒ (nat ⇀ 'a) option where
  match [] [] = Some Map.empty
| match (MFOTL.Const x # ts) (y # ys) = (if x = y then match ts ys else None)
| match (MFOTL.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

definition update_since :: ℐ ⇒ bool ⇒ 'a table ⇒ 'a table ⇒ ts ⇒
```

```
  'a msaux ⇒ 'a table × 'a msaux where
  update_since I pos rel1 rel2 nt aux =
    (let aux = (case [(t, join rel pos rel1). (t, rel) ← aux, nt − t ≤ right I] of
        [] ⇒ [(nt, rel2)]
      | x # aux' ⇒ (if fst x = nt then (fst x, snd x ∪ rel2) # aux' else (nt, rel2) # x # aux'))
    in (foldr (∪) [rel. (t, rel) ← aux, left I ≤ nt − t] {}, aux))

definition update_until :: ℐ ⇒ bool ⇒ 'a table ⇒ 'a table ⇒ ts ⇒ 'a muaux ⇒ 'a muaux where
  update_until I pos rel1 rel2 nt aux =
    (map (λx. case x of (t, a1, a2) ⇒ (t, if pos then join a1 True rel1 else a1 ∪ rel1,
      if mem (nt − t) I then a2 ∪ join rel2 pos a1 else a2)) aux) @
    [(nt, rel1, if left I = 0 then rel2 else empty_table)]

fun eval_until :: ℐ ⇒ ts ⇒ 'a muaux ⇒ 'a table list × 'a muaux where
  eval_until I nt [] = ([], [])
| eval_until I nt ((t, a1, a2) # aux) = (if t + right I < nt then
    (let (xs, aux) = eval_until I nt aux in (a2 # xs, aux)) else ([], (t, a1, a2) # aux))

primrec meval :: nat ⇒ ts ⇒ 'a MFOTL.database ⇒ 'a mformula ⇒ 'a table list × 'a mformula where
  meval n t db (MRel rel) = ([rel], MRel rel)
| meval n t db (MPred e ts) = ([(λf. tabulate f 0 n) ‘ Option.these
    (match ts ‘ (⋃(e', x)∈db. if e = e' then {x} else {}))], MPred e ts)
| meval n t db (MAnd φ pos ψ buf) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      (zs, buf) = mbuf2_take (λr1 r2. join r1 pos r2) (mbuf2_add xs ys buf)
    in (zs, MAnd φ pos ψ buf))
| meval n t db (MOr φ ψ buf) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      (zs, buf) = mbuf2_take (λr1 r2. r1 ∪ r2) (mbuf2_add xs ys buf)
    in (zs, MOr φ ψ buf))
| meval n t db (MExists φ) =
    (let (xs, φ) = meval (Suc n) t db φ in (map (λr. tl ‘ r) xs, MExists φ))
| meval n t db (MPrev I φ first buf nts) =
    (let (xs, φ) = meval n t db φ;
      (zs, buf, nts) = mprev_next I (buf @ xs) (nts @ [t])
    in (if first then empty_table # zs else zs, MPrev I φ False buf nts))
| meval n t db (MNext I φ first nts) =
    (let (xs, φ) = meval n t db φ;
      (xs, first) = (case (xs, first) of (_ # xs, True) ⇒ (xs, False) | a ⇒ a);
      (zs, _, nts) = mprev_next I xs (nts @ [t])
    in (zs, MNext I φ first nts))
| meval n t db (MSince pos φ I ψ buf nts aux) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      ((zs, aux), buf, nts) = mbuf2t_take (λr1 r2 t (zs, aux).
        let (z, aux) = update_since I pos r1 r2 t aux
        in (zs @ [z], aux)) ([], aux) (mbuf2_add xs ys buf) (nts @ [t])
    in (zs, MSince pos φ I ψ buf nts aux))
| meval n t db (MUntil pos φ I ψ buf nts aux) =
    (let (xs, φ) = meval n t db φ; (ys, ψ) = meval n t db ψ;
      (aux, buf, nts) = mbuf2t_take (update_until I pos) aux (mbuf2_add xs ys buf) (nts @ [t]);
      (zs, aux) = eval_until I (case nts of [] ⇒ t | nt # _ ⇒ nt) aux
    in (zs, MUntil pos φ I ψ buf nts aux))

definition mstep :: 'a MFOTL.database × ts ⇒ 'a mstate ⇒ (nat × 'a tuple) set × 'a mstate where
  mstep tdb st =
    (let (xs, m) = meval (mstate_n st) (snd tdb) (fst tdb) (mstate_m st)
    in (⋃ (set (map (λ(i, X). (λv. (i, v)) ‘ X) (List.enumerate (mstate_i st) xs))),
     (|mstate_i = mstate_i st + length xs, mstate_m = m, mstate_n = mstate_n st|)))
```

lemma *mstep_alt*: *mstep tdb st* =
(*let* (*xs*, *m*) = *meval* (*mstate_n st*) (*snd tdb*) (*fst tdb*) (*mstate_m st*)
in (⋃(*i*, *X*) ∈ *set* (*List.enumerate* (*mstate_i st*) *xs*). ⋃*v* ∈ *X*. {(*i*,*v*)},
(⦇*mstate_i* = *mstate_i st* + *length xs*, *mstate_m* = *m*, *mstate_n* = *mstate_n st*⦈)))
⟨*proof*⟩

6.3 Progress

primrec *progress* :: *'a MFOTL.trace* ⇒ *'a MFOTL.formula* ⇒ *nat* ⇒ *nat* **where**
progress σ (*MFOTL.Pred e ts*) *j* = *j*
| *progress σ* (*MFOTL.Eq t1 t2*) *j* = *j*
| *progress σ* (*MFOTL.Neg φ*) *j* = *progress σ φ j*
| *progress σ* (*MFOTL.Or φ ψ*) *j* = *min* (*progress σ φ j*) (*progress σ ψ j*)
| *progress σ* (*MFOTL.Exists φ*) *j* = *progress σ φ j*
| *progress σ* (*MFOTL.Prev I φ*) *j* = (*if j* = *0 then 0 else min* (*Suc* (*progress σ φ j*)) *j*)
| *progress σ* (*MFOTL.Next I φ*) *j* = *progress σ φ j* − *1*
| *progress σ* (*MFOTL.Since φ I ψ*) *j* = *min* (*progress σ φ j*) (*progress σ ψ j*)
| *progress σ* (*MFOTL.Until φ I ψ*) *j* =
Inf {*i*. ∀*k*. *k* < *j* ∧ *k* ≤ *min* (*progress σ φ j*) (*progress σ ψ j*) ⟶ *τ σ i* + *right I* ≥ *τ σ k*}

lemma *progress_And*[*simp*]: *progress σ* (*MFOTL.And φ ψ*) *j* = *min* (*progress σ φ j*) (*progress σ ψ j*)
⟨*proof*⟩

lemma *progress_And_Not*[*simp*]: *progress σ* (*MFOTL.And_Not φ ψ*) *j* = *min* (*progress σ φ j*) (*progress σ ψ j*)
⟨*proof*⟩

lemma *progress_mono*: *j* ≤ *j'* ⟹ *progress σ φ j* ≤ *progress σ φ j'*
⟨*proof*⟩

lemma *progress_le*: *progress σ φ j* ≤ *j*
⟨*proof*⟩

lemma *progress_0*[*simp*]: *progress σ φ 0* = *0*
⟨*proof*⟩

lemma *progress_ge*: *MFOTL.future_reach φ* ≠ ∞ ⟹ ∃*j*. *i* ≤ *progress σ φ j*
⟨*proof*⟩

lemma *cInf_restrict_nat*:
fixes *x* :: *nat*
assumes *x* ∈ *A*
shows *Inf A* = *Inf* {*y* ∈ *A*. *y* ≤ *x*}
⟨*proof*⟩

lemma *progress_time_conv*:
assumes ∀*i*<*j*. *τ σ i* = *τ σ' i*
shows *progress σ φ j* = *progress σ' φ j*
⟨*proof*⟩

lemma *Inf_UNIV_nat*: (*Inf UNIV* :: *nat*) = *0*
⟨*proof*⟩

lemma *progress_prefix_conv*:
assumes *prefix_of π σ* **and** *prefix_of π σ'*
shows *progress σ φ* (*plen π*) = *progress σ' φ* (*plen π*)
⟨*proof*⟩

lemma *sat_prefix_conv*:
assumes *prefix_of* π σ **and** *prefix_of* π σ' **and** $i <$ *progress* σ φ (*plen* π)
shows *MFOTL.sat* σ v i $\varphi \longleftrightarrow$ *MFOTL.sat* σ' v i φ
⟨*proof*⟩

6.4 Specification

definition *pprogress* :: *'a MFOTL.formula* ⇒ *'a MFOTL.prefix* ⇒ *nat* **where**
pprogress φ π = (*THE n*. ∀σ. *prefix_of* π σ ⟶ *progress* σ φ (*plen* π) = *n*)

lemma *pprogress_eq*: *prefix_of* π σ ⟹ *pprogress* φ π = *progress* σ φ (*plen* π)
⟨*proof*⟩

locale *future_bounded_mfotl* =
fixes φ :: *'a MFOTL.formula*
assumes *future_bounded*: *MFOTL.future_reach* $\varphi \neq \infty$

sublocale *future_bounded_mfotl* ⊆ *sliceable_timed_progress MFOTL.nfv* φ *MFOTL.fv* φ *relevant_events* φ
$\lambda\sigma$ v i. *MFOTL.sat* σ v i φ *pprogress* φ
⟨*proof*⟩

locale *monitorable_mfotl* =
fixes φ :: *'a MFOTL.formula*
assumes *monitorable*: *mmonitorable* φ

sublocale *monitorable_mfotl* ⊆ *future_bounded_mfotl*
⟨*proof*⟩

6.5 Correctness

6.5.1 Invariants

definition *wf_mbuf2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a table* ⇒ *bool*) ⇒ (*nat* ⇒ *'a table* ⇒ *bool*) ⇒ *'a mbuf2* ⇒ *bool* **where**
wf_mbuf2 i ja jb P Q buf ⟷ $i \leq ja \wedge i \leq jb \wedge$ (*case buf of* (*xs*, *ys*) ⇒
list_all2 P [*i*..<*ja*] *xs* ∧ *list_all2 Q* [*i*..<*jb*] *ys*)

definition *wf_mbuf2'* :: *'a MFOTL.trace* ⇒ *nat* ⇒ *nat* ⇒ *'a list set* ⇒
'a MFOTL.formula ⇒ *'a MFOTL.formula* ⇒ *'a mbuf2* ⇒ *bool* **where**
wf_mbuf2' σ *j n R* φ ψ *buf* ⟷ *wf_mbuf2* (*min* (*progress* σ φ *j*) (*progress* σ ψ *j*))
(*progress* σ φ *j*) (*progress* σ ψ *j*)
(λi. *qtable n* (*MFOTL.fv* φ) (*mem_restr R*) (λv. *MFOTL.sat* σ (*map the v*) *i* φ))
(λi. *qtable n* (*MFOTL.fv* ψ) (*mem_restr R*) (λv. *MFOTL.sat* σ (*map the v*) *i* ψ)) *buf*

lemma *wf_mbuf2'_UNIV_alt*: *wf_mbuf2'* σ *j n UNIV* φ ψ *buf* ⟷ (*case buf of* (*xs*, *ys*) ⇒
list_all2 (λi. *wf_table n* (*MFOTL.fv* φ) (λv. *MFOTL.sat* σ (*map the v*) *i* φ))
[*min* (*progress* σ φ *j*) (*progress* σ ψ *j*) ..< (*progress* σ φ *j*)] *xs* ∧
list_all2 (λi. *wf_table n* (*MFOTL.fv* ψ) (λv. *MFOTL.sat* σ (*map the v*) *i* ψ))
[*min* (*progress* σ φ *j*) (*progress* σ ψ *j*) ..< (*progress* σ ψ *j*)] *ys*)
⟨*proof*⟩

definition *wf_ts* :: *'a MFOTL.trace* ⇒ *nat* ⇒ *'a MFOTL.formula* ⇒ *'a MFOTL.formula* ⇒ *ts list* ⇒ *bool* **where**
wf_ts σ *j* φ ψ *ts* ⟷ *list_all2* (λi *t*. *t* = τ σ *i*) [*min* (*progress* σ φ *j*) (*progress* σ ψ *j*)..<*j*] *ts*

abbreviation *Sincep pos* φ *I* ψ ≡ *MFOTL.Since* (*if pos then* φ *else MFOTL.Neg* φ) *I* ψ

definition *wf_since_aux* :: *'a MFOTL.trace ⇒ nat ⇒ 'a list set ⇒ bool ⇒*
'a MFOTL.formula ⇒ ℐ ⇒ 'a MFOTL.formula ⇒ 'a msaux ⇒ nat ⇒ bool **where**
wf_since_aux σ n R pos φ I ψ aux ne ⟷ sorted_wrt (λx y. fst x > fst y) aux ∧
(∀ t X. (t, X) ∈ set aux ⟶ ne ≠ 0 ∧ t ≤ τ σ (ne−1) ∧ τ σ (ne−1) − t ≤ right I ∧ (∃ i. τ σ i = t) ∧
qtable n (MFOTL.fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne−1) (Sincep pos φ (point (τ σ (ne−1) − t)) ψ)) X) ∧
(∀ t. ne ≠ 0 ∧ t ≤ τ σ (ne−1) ∧ τ σ (ne−1) − t ≤ right I ∧ (∃ i. τ σ i = t) ⟶
(∃ X. (t, X) ∈ set aux))

lemma *qtable_mem_restr_UNIV*: *qtable n A (mem_restr UNIV) Q X = wf_table n A Q X*
⟨*proof*⟩

lemma *wf_since_aux_UNIV_alt*:
wf_since_aux σ n UNIV pos φ I ψ aux ne ⟷ sorted_wrt (λx y. fst x > fst y) aux ∧
(∀ t X. (t, X) ∈ set aux ⟶ ne ≠ 0 ∧ t ≤ τ σ (ne−1) ∧ τ σ (ne−1) − t ≤ right I ∧ (∃ i. τ σ i = t) ∧
wf_table n (MFOTL.fv ψ)
(λv. MFOTL.sat σ (map the v) (ne−1) (Sincep pos φ (point (τ σ (ne−1) − t)) ψ)) X) ∧
(∀ t. ne ≠ 0 ∧ t ≤ τ σ (ne−1) ∧ τ σ (ne−1) − t ≤ right I ∧ (∃ i. τ σ i = t) ⟶
(∃ X. (t, X) ∈ set aux))
⟨*proof*⟩

definition *wf_until_aux* :: *'a MFOTL.trace ⇒ nat ⇒ 'a list set ⇒ bool ⇒*
'a MFOTL.formula ⇒ ℐ ⇒ 'a MFOTL.formula ⇒ 'a muaux ⇒ nat ⇒ bool **where**
wf_until_aux σ n R pos φ I ψ aux ne ⟷ list_all2 (λx i. case x of (t, r1, r2) ⇒ t = τ σ i ∧
qtable n (MFOTL.fv φ) (mem_restr R) (λv. if pos then (∀ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)
else (∃ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)) r1 ∧
qtable n (MFOTL.fv ψ) (mem_restr R) (λv. (∃ j. i ≤ j ∧ j < ne + length aux ∧ mem (τ σ j − τ σ i) I ∧
MFOTL.sat σ (map the v) j ψ ∧
(∀ k∈{i..<j}. if pos then MFOTL.sat σ (map the v) k φ else ¬ MFOTL.sat σ (map the v) k φ))) r2)
aux [ne..<ne+length aux]

lemma *wf_until_aux_UNIV_alt*:
wf_until_aux σ n UNIV pos φ I ψ aux ne ⟷ list_all2 (λx i. case x of (t, r1, r2) ⇒ t = τ σ i ∧
wf_table n (MFOTL.fv φ) (λv. if pos
then (∀ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)
else (∃ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)) r1 ∧
wf_table n (MFOTL.fv ψ) (λv. ∃ j. i ≤ j ∧ j < ne + length aux ∧ mem (τ σ j − τ σ i) I ∧
MFOTL.sat σ (map the v) j ψ ∧
(∀ k∈{i..<j}. if pos then MFOTL.sat σ (map the v) k φ else ¬ MFOTL.sat σ (map the v) k φ)) r2)
aux [ne..<ne+length aux]
⟨*proof*⟩

inductive *wf_mformula* :: *'a MFOTL.trace ⇒ nat ⇒*
nat ⇒ 'a list set ⇒ 'a mformula ⇒ 'a MFOTL.formula ⇒ bool
for *σ j* **where**
Eq: *MFOTL.is_Const t1 ∨ MFOTL.is_Const t2 ⟹*
∀ x∈MFOTL.fv_trm t1. x < n ⟹ ∀ x∈MFOTL.fv_trm t2. x < n ⟹
wf_mformula σ j n R (MRel (eq_rel n t1 t2)) (MFOTL.Eq t1 t2)
| *neq_Const*: *φ = MRel (neq_rel n (MFOTL.Const x) (MFOTL.Const y)) ⟹*
wf_mformula σ j n R φ (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))
| *neq_Var*: *x < n ⟹*

wf_mformula σ j n R (*MRel empty_table*) (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Var* x) (*MFOTL.Var* x)))
| *Pred*: $\forall x{\in}$*MFOTL.fv* (*MFOTL.Pred* e ts). $x < n \Longrightarrow$
wf_mformula σ j n R (*MPred* e ts) (*MFOTL.Pred* e ts)
| *And*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$ *wf_mformula* σ j n R ψ ψ' $\Longrightarrow$
if pos then χ = *MFOTL.And* φ' $\psi' \wedge \neg$ (*safe_formula* (*MFOTL.Neg* ψ') $\wedge$ *MFOTL.fv* $\psi' \subseteq$ *MFOTL.fv* φ')
else χ = *MFOTL.And_Not* φ' $\psi' \wedge$ *safe_formula* $\psi' \wedge$ *MFOTL.fv* $\psi' \subseteq$ *MFOTL.fv* $\varphi' \Longrightarrow$
wf_mbuf2' σ j n R φ' ψ' *buf* $\Longrightarrow$
wf_mformula σ j n R (*MAnd* φ *pos* ψ *buf*) χ
| *Or*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$ *wf_mformula* σ j n R ψ ψ' $\Longrightarrow$
MFOTL.fv φ' = *MFOTL.fv* ψ' $\Longrightarrow$
wf_mbuf2' σ j n R φ' ψ' *buf* $\Longrightarrow$
wf_mformula σ j n R (*MOr* φ ψ *buf*) (*MFOTL.Or* φ' ψ')
| *Exists*: *wf_mformula* σ j (*Suc* n) (*lift_envs* R) φ φ' $\Longrightarrow$
wf_mformula σ j n R (*MExists* φ) (*MFOTL.Exists* φ')
| *Prev*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$
first $\longleftrightarrow$ j = 0 $\Longrightarrow$
list_all2 (λi. *qtable* n (*MFOTL.fv* φ') (*mem_restr* R) (λv. *MFOTL.sat* σ (*map the* v) i φ'))
[*min* (*progress* σ φ' j) (j−1)..<*progress* σ φ' j] *buf* $\Longrightarrow$
list_all2 (λi t. t = τ σ i) [*min* (*progress* σ φ' j) (j−1)..<j] *nts* $\Longrightarrow$
wf_mformula σ j n R (*MPrev I* φ *first buf nts*) (*MFOTL.Prev I* φ')
| *Next*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$
first $\longleftrightarrow$ *progress* σ φ' j = 0 $\Longrightarrow$
list_all2 (λi t. t = τ σ i) [*progress* σ φ' j − 1..<j] *nts* $\Longrightarrow$
wf_mformula σ j n R (*MNext I* φ *first nts*) (*MFOTL.Next I* φ')
| *Since*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$ *wf_mformula* σ j n R ψ ψ' $\Longrightarrow$
if pos then φ'' = φ' *else* φ'' = *MFOTL.Neg* φ' $\Longrightarrow$
safe_formula φ'' = *pos* $\Longrightarrow$
MFOTL.fv $\varphi' \subseteq$ *MFOTL.fv* ψ' $\Longrightarrow$
wf_mbuf2' σ j n R φ' ψ' *buf* $\Longrightarrow$
wf_ts σ j φ' ψ' *nts* $\Longrightarrow$
wf_since_aux σ n R *pos* φ' I ψ' *aux* (*progress* σ (*MFOTL.Since* φ'' I ψ') j) $\Longrightarrow$
wf_mformula σ j n R (*MSince pos* φ I ψ *buf nts aux*) (*MFOTL.Since* φ'' I ψ')
| *Until*: *wf_mformula* σ j n R φ φ' $\Longrightarrow$ *wf_mformula* σ j n R ψ ψ' $\Longrightarrow$
if pos then φ'' = φ' *else* φ'' = *MFOTL.Neg* φ' $\Longrightarrow$
safe_formula φ'' = *pos* $\Longrightarrow$
MFOTL.fv $\varphi' \subseteq$ *MFOTL.fv* ψ' $\Longrightarrow$
wf_mbuf2' σ j n R φ' ψ' *buf* $\Longrightarrow$
wf_ts σ j φ' ψ' *nts* $\Longrightarrow$
wf_until_aux σ n R *pos* φ' I ψ' *aux* (*progress* σ (*MFOTL.Until* φ'' I ψ') j) $\Longrightarrow$
progress σ (*MFOTL.Until* φ'' I ψ') j + *length aux* = *min* (*progress* σ φ' j) (*progress* σ ψ' j) $\Longrightarrow$
wf_mformula σ j n R (*MUntil pos* φ I ψ *buf nts aux*) (*MFOTL.Until* φ'' I ψ')

definition *wf_mstate* :: *'a MFOTL.formula* ⇒ *'a MFOTL.prefix* ⇒ *'a list set* ⇒ *'a mstate* ⇒ *bool* **where**
wf_mstate φ π R *st* $\longleftrightarrow$ *mstate_n st* = *MFOTL.nfv* $\varphi \wedge$ ($\forall \sigma$. *prefix_of* π σ $\longrightarrow$
mstate_i st = *progress* σ φ (*plen* π) $\wedge$
wf_mformula σ (*plen* π) (*mstate_n st*) R (*mstate_m st*) φ)

6.5.2 Initialisation

lemma *minit0_And*: ¬ (*safe_formula* (*MFOTL.Neg* ψ) $\wedge$ *MFOTL.fv* $\psi \subseteq$ *MFOTL.fv* φ) $\Longrightarrow$
minit0 n (*MFOTL.And* φ ψ) = *MAnd* (*minit0 n* φ) *True* (*minit0 n* ψ) ([], [])
⟨*proof*⟩

lemma *minit0_And_Not*: *safe_formula* $\psi \wedge$ *MFOTL.fv* $\psi \subseteq$ *MFOTL.fv* $\varphi \Longrightarrow$
minit0 n (*MFOTL.And_Not* φ ψ) = (*MAnd* (*minit0 n* φ) *False* (*minit0 n* ψ) ([], []))
⟨*proof*⟩

lemma *wf_mbuf2'_0*: *wf_mbuf2' σ 0 n R φ ψ* ([], [])
⟨*proof*⟩

lemma *wf_ts_0*: *wf_ts σ 0 φ ψ* []
⟨*proof*⟩

lemma *wf_since_aux_Nil*: *wf_since_aux σ n R pos φ' I ψ'* [] *0*
⟨*proof*⟩

lemma *wf_until_aux_Nil*: *wf_until_aux σ n R pos φ' I ψ'* [] *0*
⟨*proof*⟩

lemma *wf_minit0*: *safe_formula φ* ⟹ ∀ *x*∈*MFOTL.fv φ*. *x* < *n* ⟹
wf_mformula σ 0 n R (*minit0 n φ*) *φ*
⟨*proof*⟩

lemma *wf_mstate_minit*: *safe_formula φ* ⟹ *wf_mstate φ pnil R* (*minit φ*)
⟨*proof*⟩

6.5.3 Evaluation

lemma *match_wf_tuple*: *Some f* = *match ts xs* ⟹ *wf_tuple n* (⋃ *t*∈*set ts*. *MFOTL.fv_trm t*) (*tabulate f 0 n*)
⟨*proof*⟩

lemma *match_fvi_trm_None*: *Some f* = *match ts xs* ⟹ ∀ *t*∈*set ts*. *x* ∉ *MFOTL.fv_trm t* ⟹ *f x* = *None*
⟨*proof*⟩

lemma *match_fvi_trm_Some*: *Some f* = *match ts xs* ⟹ *t* ∈ *set ts* ⟹ *x* ∈ *MFOTL.fv_trm t* ⟹ *f x* ≠ *None*
⟨*proof*⟩

lemma *match_eval_trm*: ∀ *t*∈*set ts*. ∀ *i*∈*MFOTL.fv_trm t*. *i* < *n* ⟹ *Some f* = *match ts xs* ⟹
map (*MFOTL.eval_trm* (*tabulate* (λ*i*. *the* (*f i*)) *0 n*)) *ts* = *xs*
⟨*proof*⟩

lemma *wf_tuple_tabulate_Some*: *wf_tuple n A* (*tabulate f 0 n*) ⟹ *x* ∈ *A* ⟹ *x* < *n* ⟹ ∃ *y*. *f x* = *Some y*
⟨*proof*⟩

lemma *ex_match*: *wf_tuple n* (⋃ *t*∈*set ts*. *MFOTL.fv_trm t*) *v* ⟹ ∀ *t*∈*set ts*. ∀ *x*∈*MFOTL.fv_trm t*. *x* < *n* ⟹
∃ *f*. *match ts* (*map* (*MFOTL.eval_trm* (*map the v*)) *ts*) = *Some f* ∧ *v* = *tabulate f 0 n*
⟨*proof*⟩

lemma *eq_rel_eval_trm*: *v* ∈ *eq_rel n t1 t2* ⟹ *MFOTL.is_Const t1* ∨ *MFOTL.is_Const t2* ⟹
∀ *x*∈*MFOTL.fv_trm t1* ∪ *MFOTL.fv_trm t2*. *x* < *n* ⟹
MFOTL.eval_trm (*map the v*) *t1* = *MFOTL.eval_trm* (*map the v*) *t2*
⟨*proof*⟩

lemma *in_eq_rel*: *wf_tuple n* (*MFOTL.fv_trm t1* ∪ *MFOTL.fv_trm t2*) *v* ⟹
MFOTL.is_Const t1 ∨ *MFOTL.is_Const t2* ⟹
MFOTL.eval_trm (*map the v*) *t1* = *MFOTL.eval_trm* (*map the v*) *t2* ⟹
v ∈ *eq_rel n t1 t2*
⟨*proof*⟩

lemma *table_eq_rel*: *MFOTL.is_Const t1* ∨ *MFOTL.is_Const t2* ⟹
table n (*MFOTL.fv_trm t1* ∪ *MFOTL.fv_trm t2*) (*eq_rel n t1 t2*)

⟨*proof*⟩

lemma *wf_tuple_Suc_fviD*: *wf_tuple* (*Suc n*) (*MFOTL.fvi b* φ) *v* $\Longrightarrow$ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) φ) (*tl v*)
⟨*proof*⟩

lemma *table_fvi_tl*: *table* (*Suc n*) (*MFOTL.fvi b* φ) *X* $\Longrightarrow$ *table n* (*MFOTL.fvi* (*Suc b*) φ) (*tl* ‘ *X*)
⟨*proof*⟩

lemma *wf_tuple_Suc_fvi_SomeI*: *0* $\in$ *MFOTL.fvi b* φ $\Longrightarrow$ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) φ) *v* $\Longrightarrow$
wf_tuple (*Suc n*) (*MFOTL.fvi b* φ) (*Some x* # *v*)
⟨*proof*⟩

lemma *wf_tuple_Suc_fvi_NoneI*: *0* $\notin$ *MFOTL.fvi b* φ $\Longrightarrow$ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) φ) *v* $\Longrightarrow$
wf_tuple (*Suc n*) (*MFOTL.fvi b* φ) (*None* # *v*)
⟨*proof*⟩

lemma *qtable_project_fv*: *qtable* (*Suc n*) (*fv* φ) (*mem_restr* (*lift_envs R*)) *P X* $\Longrightarrow$
qtable n (*MFOTL.fvi* (*Suc 0*) φ) (*mem_restr R*)
(λ*v*. ∃ *x*. *P* ((*if 0* $\in$ *fv* φ *then Some x else None*) # *v*)) (*tl* ‘ *X*)
⟨*proof*⟩

lemma *mprev*: *mprev_next I xs ts* = (*ys*, *xs*′, *ts*′) $\Longrightarrow$
list_all2 P [*i*..<*j*′] *xs* $\Longrightarrow$ *list_all2* (λ*i t*. *t* = τ σ *i*) [*i*..<*j*] *ts* $\Longrightarrow$ *i* $\leq$ *j*′ $\Longrightarrow$ *i* < *j* $\Longrightarrow$
list_all2 (λ*i X*. *if mem* (τ σ (*Suc i*) − τ σ *i*) *I then P i X else X* = *empty_table*)
[*i*..<*min j*′ (*j*−*1*)] *ys* ∧
list_all2 P [*min j*′ (*j*−*1*)..<*j*′] *xs*′ ∧
list_all2 (λ*i t*. *t* = τ σ *i*) [*min j*′ (*j*−*1*)..<*j*] *ts*′
⟨*proof*⟩

lemma *mnext*: *mprev_next I xs ts* = (*ys*, *xs*′, *ts*′) $\Longrightarrow$
list_all2 P [*Suc i*..<*j*′] *xs* $\Longrightarrow$ *list_all2* (λ*i t*. *t* = τ σ *i*) [*i*..<*j*] *ts* $\Longrightarrow$ *Suc i* $\leq$ *j*′ $\Longrightarrow$ *i* < *j* $\Longrightarrow$
list_all2 (λ*i X*. *if mem* (τ σ (*Suc i*) − τ σ *i*) *I then P* (*Suc i*) *X else X* = *empty_table*)
[*i*..<*min* (*j*′−*1*) (*j*−*1*)] *ys* ∧
list_all2 P [*Suc* (*min* (*j*′−*1*) (*j*−*1*))..<*j*′] *xs*′ ∧
list_all2 (λ*i t*. *t* = τ σ *i*) [*min* (*j*′−*1*) (*j*−*1*)..<*j*] *ts*′
⟨*proof*⟩

lemma *in_foldr_UnI*: *x* $\in$ *A* $\Longrightarrow$ *A* $\in$ *set xs* $\Longrightarrow$ *x* $\in$ *foldr* (∪) *xs* {}
⟨*proof*⟩

lemma *in_foldr_UnE*: *x* $\in$ *foldr* (∪) *xs* {} $\Longrightarrow$ (⋀*A*. *A* $\in$ *set xs* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *sat_the_restrict*: *fv* φ $\subseteq$ *A* $\Longrightarrow$ *MFOTL.sat* σ (*map the* (*restrict A v*)) *i* φ = *MFOTL.sat* σ (*map the v*) *i* φ
⟨*proof*⟩

lemma *update_since*:
assumes *pre*: *wf_since_aux* σ *n R pos* φ *I* ψ *aux ne*
and *qtable1*: *qtable n* (*MFOTL.fv* φ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) *ne* φ) *rel1*
and *qtable2*: *qtable n* (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) *ne* ψ) *rel2*
and *result_eq*: (*rel*, *aux*′) = *update_since I pos rel1 rel2* (τ σ *ne*) *aux*
and *fvi_subset*: *MFOTL.fv* φ $\subseteq$ *MFOTL.fv* ψ
shows *wf_since_aux* σ *n R pos* φ *I* ψ *aux*′ (*Suc ne*)
and *qtable n* (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) *ne* (*Sincep pos* φ *I* ψ)) *rel*
⟨*proof*⟩

lemma *length_update_until*: *length* (*update_until pos I rel1 rel2 nt aux*) = *Suc* (*length aux*)
⟨*proof*⟩

lemma *wf_update_until*:
assumes *pre*: *wf_until_aux* σ *n R pos* φ *I* ψ *aux ne*
and *qtable1*: *qtable n* (*MFOTL.fv* φ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) (*ne* + *length aux*) φ) *rel1*
and *qtable2*: *qtable n* (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) (*ne* + *length aux*) ψ) *rel2*
and *fvi_subset*: *MFOTL.fv* φ ⊆ *MFOTL.fv* ψ
shows *wf_until_aux* σ *n R pos* φ *I* ψ (*update_until I pos rel1 rel2* (τ σ (*ne* + *length aux*)) *aux*) *ne*
⟨*proof*⟩

lemma *wf_until_aux_Cons*: *wf_until_aux* σ *n R pos* φ *I* ψ (*a* # *aux*) *ne* ⟹
wf_until_aux σ *n R pos* φ *I* ψ *aux* (*Suc ne*)
⟨*proof*⟩

lemma *wf_until_aux_Cons1*: *wf_until_aux* σ *n R pos* φ *I* ψ ((*t*, *a1*, *a2*) # *aux*) *ne* ⟹ *t* = τ σ *ne*
⟨*proof*⟩

lemma *wf_until_aux_Cons3*: *wf_until_aux* σ *n R pos* φ *I* ψ ((*t*, *a1*, *a2*) # *aux*) *ne* ⟹
qtable n (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. (∃ *j*. *ne* ≤ *j* ∧ *j* < *Suc* (*ne* + *length aux*) ∧ *mem* (τ σ *j* − τ σ *ne*) *I* ∧
MFOTL.sat σ (*map the v*) *j* ψ ∧ (∀ *k*∈{*ne*..<*j*}. *if pos then MFOTL.sat* σ (*map the v*) *k* φ *else* ¬ *MFOTL.sat* σ (*map the v*) *k* φ))) *a2*
⟨*proof*⟩

lemma *upt_append*: *a* ≤ *b* ⟹ *b* ≤ *c* ⟹ [*a*..<*b*] @ [*b*..<*c*] = [*a*..<*c*]
⟨*proof*⟩

lemma *wf_mbuf2_add*:
assumes *wf_mbuf2 i ja jb P Q buf*
and *list_all2 P* [*ja*..<*ja'*] *xs*
and *list_all2 Q* [*jb*..<*jb'*] *ys*
and *ja* ≤ *ja'* *jb* ≤ *jb'*
shows *wf_mbuf2 i ja' jb' P Q* (*mbuf2_add xs ys buf*)
⟨*proof*⟩

lemma *mbuf2_take_eqD*:
assumes *mbuf2_take f buf* = (*xs*, *buf'*)
and *wf_mbuf2 i ja jb P Q buf*
shows *wf_mbuf2* (*min ja jb*) *ja jb P Q buf'*
and *list_all2* (λ*i z*. ∃ *x y*. *P i x* ∧ *Q i y* ∧ *z* = *f x y*) [*i*..<*min ja jb*] *xs*
⟨*proof*⟩

lemma *mbuf2t_take_eqD*:
assumes *mbuf2t_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbuf2 i ja jb P Q buf*
and *list_all2 R* [*i*..<*j*] *nts*
and *ja* ≤ *j* *jb* ≤ *j*
shows *wf_mbuf2* (*min ja jb*) *ja jb P Q buf'*
and *list_all2 R* [*min ja jb*..<*j*] *nts'*
⟨*proof*⟩

lemma *mbuf2t_take_induct*[*consumes 5*, *case_names base step*]:
assumes *mbuf2t_take f z buf nts* = (*z'*, *buf'*, *nts'*)
and *wf_mbuf2 i ja jb P Q buf*

and *list_all2 R* [*i*..<*j*] *nts*
and *ja* ≤ *j jb* ≤ *j*
and *U i z*
and ⋀*k x y t z*. *i* ≤ *k* ⟹ *Suc k* ≤ *ja* ⟹ *Suc k* ≤ *jb* ⟹
P k x ⟹ *Q k y* ⟹ *R k t* ⟹ *U k z* ⟹ *U* (*Suc k*) (*f x y t z*)
shows *U* (*min ja jb*) *z*′
⟨*proof*⟩

lemma *mbuf2_take_add*′:
assumes *eq*: *mbuf2_take f* (*mbuf2_add xs ys buf*) = (*zs*, *buf*′)
and *pre*: *wf_mbuf2*′ *σ j n R φ ψ buf*
and *xs*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv φ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i φ*))
[*progress σ φ j*..<*progress σ φ j*′] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv ψ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i ψ*))
[*progress σ ψ j*..<*progress σ ψ j*′] *ys*
and *j* ≤ *j*′
shows *wf_mbuf2*′ *σ j*′ *n R φ ψ buf*′
and *list_all2* (λ*i Z*. ∃ *X Y*.
qtable n (*MFOTL.fv φ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i φ*) *X* ∧
qtable n (*MFOTL.fv ψ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i ψ*) *Y* ∧
Z = *f X Y*)
[*min* (*progress σ φ j*) (*progress σ ψ j*)..<*min* (*progress σ φ j*′) (*progress σ ψ j*′)] *zs*
⟨*proof*⟩

lemma *mbuf2t_take_add*′:
assumes *eq*: *mbuf2t_take f z* (*mbuf2_add xs ys buf*) *nts* = (*z*′, *buf*′, *nts*′)
and *pre_buf*: *wf_mbuf2*′ *σ j n R φ ψ buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = *τ σ i*) [*min* (*progress σ φ j*) (*progress σ ψ j*)..<*j*′] *nts*
and *xs*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv φ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i φ*))
[*progress σ φ j*..<*progress σ φ j*′] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv ψ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i ψ*))
[*progress σ ψ j*..<*progress σ ψ j*′] *ys*
and *j* ≤ *j*′
shows *wf_mbuf2*′ *σ j*′ *n R φ ψ buf*′
and *wf_ts σ j*′ *φ ψ nts*′
⟨*proof*⟩

lemma *mbuf2t_take_add_induct*′[*consumes* 6, *case_names base step*]:
assumes *eq*: *mbuf2t_take f z* (*mbuf2_add xs ys buf*) *nts* = (*z*′, *buf*′, *nts*′)
and *pre_buf*: *wf_mbuf2*′ *σ j n R φ ψ buf*
and *pre_nts*: *list_all2* (λ*i t*. *t* = *τ σ i*) [*min* (*progress σ φ j*) (*progress σ ψ j*)..<*j*′] *nts*
and *xs*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv φ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i φ*))
[*progress σ φ j*..<*progress σ φ j*′] *xs*
and *ys*: *list_all2* (λ*i*. *qtable n* (*MFOTL.fv ψ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *i ψ*))
[*progress σ ψ j*..<*progress σ ψ j*′] *ys*
and *j* ≤ *j*′
and *base*: *U* (*min* (*progress σ φ j*) (*progress σ ψ j*)) *z*
and *step*: ⋀*k X Y z*. *min* (*progress σ φ j*) (*progress σ ψ j*) ≤ *k* ⟹
Suc k ≤ *progress σ φ j*′ ⟹ *Suc k* ≤ *progress σ ψ j*′ ⟹
qtable n (*MFOTL.fv φ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *k φ*) *X* ⟹
qtable n (*MFOTL.fv ψ*) (*mem_restr R*) (λ*v*. *MFOTL.sat σ* (*map the v*) *k ψ*) *Y* ⟹
U k z ⟹ *U* (*Suc k*) (*f X Y* (*τ σ k*) *z*)
shows *U* (*min* (*progress σ φ j*′) (*progress σ ψ j*′)) *z*′
⟨*proof*⟩

lemma *progress_Until_le*: *progress σ* (*formula.Until φ I ψ*) *j* ≤ *min* (*progress σ φ j*) (*progress σ ψ j*)
⟨*proof*⟩

lemma *list_all2_upt_Cons*: $P\ a\ x \Longrightarrow list\_all2\ P\ [Suc\ a..<b]\ xs \Longrightarrow Suc\ a \leq b \Longrightarrow$
$list\_all2\ P\ [a..<b]\ (x\ \#\ xs)$
⟨*proof*⟩

lemma *list_all2_upt_append*: $list\_all2\ P\ [a..<b]\ xs \Longrightarrow list\_all2\ P\ [b..<c]\ ys \Longrightarrow$
$a \leq b \Longrightarrow b \leq c \Longrightarrow list\_all2\ P\ [a..<c]\ (xs\ @\ ys)$
⟨*proof*⟩

lemma *meval*:
assumes *wf_mformula* $\sigma\ j\ n\ R\ \varphi\ \varphi'$
shows *case meval* $n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi$ *of* $(xs, \varphi_n) \Rightarrow wf\_mformula\ \sigma\ (Suc\ j)\ n\ R\ \varphi_n\ \varphi' \wedge$
$list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \varphi')\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi'))$
$[progress\ \sigma\ \varphi'\ j..<progress\ \sigma\ \varphi'\ (Suc\ j)]\ xs$
⟨*proof*⟩

6.5.4 Monitor step

lemma *wf_mstate_mstep*: $wf\_mstate\ \varphi\ \pi\ R\ st \Longrightarrow last\_ts\ \pi \leq snd\ tdb \Longrightarrow$
$wf\_mstate\ \varphi\ (psnoc\ \pi\ tdb)\ R\ (snd\ (mstep\ tdb\ st))$
⟨*proof*⟩

lemma *mstep_output_iff*:
assumes *wf_mstate* $\varphi\ \pi\ R\ st\ last\_ts\ \pi \leq snd\ tdb\ prefix\_of\ (psnoc\ \pi\ tdb)\ \sigma\ mem\_restr\ R\ v$
shows $(i, v) \in fst\ (mstep\ tdb\ st) \longleftrightarrow$
$progress\ \sigma\ \varphi\ (plen\ \pi) \leq i \wedge i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi)) \wedge$
$wf\_tuple\ (MFOTL.nfv\ \varphi)\ (MFOTL.fv\ \varphi)\ v \wedge MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi$
⟨*proof*⟩

6.5.5 Monitor function

definition *minit_safe* **where**
$minit\_safe\ \varphi = (if\ mmonitorable\_exec\ \varphi\ then\ minit\ \varphi\ else\ undefined)$

lemma *minit_safe_minit*: $mmonitorable\ \varphi \Longrightarrow minit\_safe\ \varphi = minit\ \varphi$
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *mstep_mverdicts*:
assumes *wf*: *wf_mstate* $\varphi\ \pi\ R\ st$
and *le*[*simp*]: $last\_ts\ \pi \leq snd\ tdb$
and *restrict*: $mem\_restr\ R\ v$
shows $(i, v) \in fst\ (mstep\ tdb\ st) \longleftrightarrow (i, v) \in M\ (psnoc\ \pi\ tdb) - M\ \pi$
⟨*proof*⟩

primrec *msteps0* **where**
$msteps0\ []\ st = (\{\}, st)$
| $msteps0\ (tdb\ \#\ \pi)\ st =$
$(let\ (V', st') = mstep\ tdb\ st;\ (V'', st'') = msteps0\ \pi\ st'\ in\ (V' \cup V'', st''))$

primrec *msteps0_stateless* **where**
$msteps0\_stateless\ []\ st = \{\}$
| $msteps0\_stateless\ (tdb\ \#\ \pi)\ st = (let\ (V', st') = mstep\ tdb\ st\ in\ V' \cup msteps0\_stateless\ \pi\ st')$

lemma *msteps0_msteps0_stateless*: $fst\ (msteps0\ w\ st) = msteps0\_stateless\ w\ st$
⟨*proof*⟩

lift_definition *msteps* :: *'a MFOTL.prefix* ⇒ *'a mstate* ⇒ $(nat \times$ *'a option list*$)\ set \times$ *'a mstate*
is *msteps0* ⟨*proof*⟩

lift_definition *msteps_stateless* :: *'a MFOTL.prefix* ⇒ *'a mstate* ⇒ $(nat \times$ *'a option list*$)\ set$

is *msteps0_stateless* ⟨*proof*⟩

lemma *msteps_msteps_stateless*: *fst* (*msteps w st*) = *msteps_stateless w st*
⟨*proof*⟩

lemma *msteps0_snoc*: *msteps0* (*π* @ [*tdb*]) *st* =
(*let* (*V'*, *st'*) = *msteps0 π st*; (*V''*, *st''*) = *mstep tdb st' in* (*V'* ∪ *V''*, *st''*))
⟨*proof*⟩

lemma *msteps_psnoc*: *last_ts π* ≤ *snd tdb* ⟹ *msteps* (*psnoc π tdb*) *st* =
(*let* (*V'*, *st'*) = *msteps π st*; (*V''*, *st''*) = *mstep tdb st' in* (*V'* ∪ *V''*, *st''*))
⟨*proof*⟩

definition *monitor* **where**
monitor φ π = *msteps_stateless π* (*minit_safe φ*)

lemma *Suc_length_conv_snoc*: (*Suc n* = *length xs*) = (∃ *y ys*. *xs* = *ys* @ [*y*] ∧ *length ys* = *n*)
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps*: *wf_mstate φ π R st* ⟹ *mem_restr R v* ⟹ *π* ≤ *π'* ⟹
X = *msteps* (*pdrop* (*plen π*) *π'*) *st* ⟹ *wf_mstate φ π' R* (*snd X*) ∧
((*i*, *v*) ∈ *fst X*) = ((*i*, *v*) ∈ *M π'* − *M π*)
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless*:
assumes *wf_mstate φ π R st mem_restr R v π* ≤ *π'*
shows (*i*, *v*) ∈ *msteps_stateless* (*pdrop* (*plen π*) *π'*) *st* ⟷ (*i*, *v*) ∈ *M π'* − *M π*
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless_UNIV*: *wf_mstate φ π UNIV st* ⟹ *π* ≤ *π'* ⟹
msteps_stateless (*pdrop* (*plen π*) *π'*) *st* = *M π'* − *M π*
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *mverdicts_Nil*: *M pnil* = {}
⟨*proof*⟩

lemma *wf_mstate_minit_safe*: *mmonitorable φ* ⟹ *wf_mstate φ pnil R* (*minit_safe φ*)
⟨*proof*⟩

lemma (**in** *monitorable_mfotl*) *monitor_mverdicts*: *monitor φ π* = *M π*
⟨*proof*⟩

6.6 Collected correctness results

context *monitorable_mfotl*
begin

We summarize the main results proved above.

1. The term M describes semantically the monitor's expected behaviour:
   - *mono_monitor*: $\pi \leq \pi' \Longrightarrow M\ \pi \subseteq M\ \pi'$
   - *sound_monitor*: $[\![(i, v) \in M\ \pi; \mathit{prefix\_of}\ \pi\ \sigma]\!] \Longrightarrow \mathit{MFOTL.sat}\ \sigma\ (\mathit{map}\ \mathit{the}\ v)\ i\ \varphi$
   - *complete_monitor*: $[\![\mathit{prefix\_of}\ \pi\ \sigma; \mathit{wf\_tuple}\ (\mathit{MFOTL.nfv}\ \varphi)\ (\mathit{fv}\ \varphi)\ v; \bigwedge\sigma.\ \mathit{prefix\_of}\ \pi\ \sigma \Longrightarrow \mathit{MFOTL.sat}\ \sigma\ (\mathit{map}\ \mathit{the}\ v)\ i\ \varphi]\!] \Longrightarrow \exists \pi'.\ \mathit{prefix\_of}\ \pi'\ \sigma \wedge (i, v) \in M\ \pi'$

- *sliceable_M*: *mem_restr S v* $\Longrightarrow$ $((i, v) \in M\ (pmap\_\Gamma\ (\lambda D.\ D \cap relevant\_events\ \varphi\ S)\ \pi)) = ((i, v) \in M\ \pi)$

2. The executable monitor's online interface *minit_safe* and *mstep* preserves the invariant *wf_mstate* and produces the the verdicts according to M:
   - *wf_mstate_minit_safe*: *mmonitorable* φ' $\Longrightarrow$ *wf_mstate* φ' *pnil R* (*minit_safe* φ')
   - *wf_mstate_mstep*: $[\![$*wf_mstate* $\varphi'\ \pi\ R\ st$; *last_ts* $\pi \leq$ *snd tdb*$]\!]$ $\Longrightarrow$ *wf_mstate* φ' (*psnoc* π *tdb*) *R* (*snd* (*mstep tdb st*))
   - *mstep_mverdicts*: $[\![$*wf_mstate* $\varphi\ \pi\ R\ st$; *last_ts* $\pi \leq$ *snd tdb*; *mem_restr R v*$]\!]$ $\Longrightarrow$ $((i, v) \in fst\ (mstep\ tdb\ st)) = ((i, v) \in M\ (psnoc\ \pi\ tdb) - M\ \pi)$
3. The executable monitor's offline interface *Monitor.monitor* implements M:
   - *monitor_mverdicts*: *Monitor.monitor* $\varphi\ \pi = M\ \pi$

end

7 Slicing framework

This section formalizes the abstract slicing framework and the joint data slicer presented in the article [3, Sections 4.2 and 4.3].

7.1 Abstract slicing

7.1.1 Definition 1

Corresponds to locale *monitor* defined in theory *MFOTL_Monitor.Abstract_Monitor*.

7.1.2 Definition 2

locale *slicer* = *monitor* +
  fixes *submonitor* :: *'k* :: *finite* $\Rightarrow$ *'a prefix* $\Rightarrow$ (*nat* $\times$ *'b option list*) *set*
  and *splitter* :: *'a prefix* $\Rightarrow$ *'k* $\Rightarrow$ *'a prefix*
  and *joiner* :: (*'k* $\Rightarrow$ (*nat* $\times$ *'b option list*) *set*) $\Rightarrow$ (*nat* $\times$ *'b option list*) *set*
assumes *mono_splitter*: $\pi \leq \pi' \Longrightarrow$ *splitter* π *k* $\leq$ *splitter* π' *k*
  and *correct_slicer*: *joiner* $(\lambda k.$ *submonitor k* (*splitter* π *k*)) = *M* π
begin

lemmas *sound_slicer* = *equalityD1*[*OF correct_slicer*]
lemmas *complete_slicer* = *equalityD2*[*OF correct_slicer*]

end

locale *self_slicer* = *slicer nfv fv sat M* $\lambda\_.$ *M splitter joiner* **for** *nfv fv sat M splitter joiner*

7.1.3 Definition 3

locale *event_separable_splitter* =
  fixes *event_splitter* :: *'a* $\Rightarrow$ *'k* :: *finite set*
begin

lift_definition *splitter* :: *'a prefix* $\Rightarrow$ *'k* $\Rightarrow$ *'a prefix* **is**
  $\lambda\pi$ *k*. *map* $(\lambda(D, t).\ (\{e \in D.\ k \in event\_splitter\ e\}, t))\ \pi$
  ⟨*proof*⟩

7.1.4 Lemma 1

lemma *mono_splitter*: $\pi \leq \pi' \Longrightarrow$ *splitter* $\pi\ k \leq$ *splitter* $\pi'\ k$
⟨*proof*⟩

end

7.2 Joint data slicer

abbreviation (*input*) *ok* $\varphi\ v \equiv$ *wf_tuple* (*MFOTL.nfv* φ) (*MFOTL.fv* φ) v

locale *splitting_strategy* =
fixes φ :: *'a MFOTL.formula*
and *strategy* :: *'a option list* ⇒ *'k* :: *finite set*
assumes *strategy_nonempty*: *ok* $\varphi\ v \Longrightarrow$ *strategy* $v \neq \{\}$
begin

abbreviation *slice_set* **where**
slice_set $k \equiv \{v.\ \exists v'.$ *map the* $v' = v \wedge$ *ok* $\varphi\ v' \wedge k \in$ *strategy* $v'\}$

end

7.2.1 Definition 4

locale *MFOTL_monitor* =
monitor MFOTL.nfv φ *MFOTL.fv* $\varphi\ \lambda\sigma\ v\ i.$ *MFOTL.sat* $\sigma\ v\ i\ \varphi\ M$ **for** $\varphi\ M$

locale *joint_data_slicer* = *MFOTL_monitor* $\varphi\ M$ + *splitting_strategy* φ *strategy*
for φ *M strategy*
begin

definition *event_splitter* **where**
event_splitter $e = (\bigcup$(*strategy* ‘ $\{v.$ *ok* $\varphi\ v \wedge$ *MFOTL.matches* (*map the* v) $\varphi\ e\}$))

sublocale *event_separable_splitter* **where** *event_splitter* = *event_splitter* ⟨*proof*⟩

definition *joiner* **where**
joiner = $(\lambda s.\ \bigcup k.\ s\ k \cap$ (*UNIV* :: *nat set*) × $\{v.\ k \in$ *strategy* $v\})$

lemma *splitter_pslice*: *splitter* $\pi\ k =$ *MFOTL_slicer.pslice* φ (*slice_set* k) π
⟨*proof*⟩

7.2.2 Lemma 2

Corresponds to the following theorem *sat_slice_strong* proved in theory *MFOTL_Monitor.Abstract_Monitor*: ⟦*relevant_events* $\varphi'\ S \subseteq E$; $v \in S$⟧ $\Longrightarrow$ *MFOTL.sat* $\sigma\ v\ i\ \varphi' =$ *MFOTL.sat* (*map_*Γ ($\lambda D.\ D \cap E$) σ) $v\ i\ \varphi'$

7.2.3 Theorem 1

sublocale *joint_monitor*: *MFOTL_monitor* $\varphi\ \lambda\pi.$ *joiner* ($\lambda k.$ *M* (*splitter* $\pi\ k$))
⟨*proof*⟩

7.2.4 Corollary 1

sublocale *joint_slicer*: *slicer MFOTL.nfv* φ *MFOTL.fv* $\varphi\ \lambda\sigma\ v\ i.$ *MFOTL.sat* $\sigma\ v\ i\ \varphi$
$\lambda\pi.$ *joiner* ($\lambda k.$ *M* (*splitter* $\pi\ k$)) $\lambda\_.$ *M splitter joiner*
⟨*proof*⟩

end

7.2.5 Definition 5

Corresponds to locale *sliceable_monitor* defined in theory *MFOTL_Monitor.Abstract_Monitor*.

locale *slicable_joint_data_slicer* =
sliceable_monitor MFOTL.nfv φ MFOTL.fv φ relevant_events φ λσ v i. MFOTL.sat σ v i φ M +
joint_data_slicer φ M strategy **for** *φ M strategy*
begin

lemma *monitor_split*: *ok φ v* ⟹ *k* ∈ *strategy v* ⟹ (*i*, *v*) ∈ *M* (*splitter π k*) ⟷ (*i*, *v*) ∈ *M π*
⟨*proof*⟩

7.2.6 Theorem 2

sublocale *self_slicer MFOTL.nfv φ MFOTL.fv φ λσ v i. MFOTL.sat σ v i φ M splitter joiner*
⟨*proof*⟩

end

7.2.7 Towards Theorem 3

fun *names* :: *'a MFOTL.formula* ⇒ *MFOTL.name set* **where**
names (*MFOTL.Pred e* _) = {*e*}
| *names* (*MFOTL.Eq* _ _) = {}
| *names* (*MFOTL.Neg ψ*) = *names ψ*
| *names* (*MFOTL.Or α β*) = *names α* ∪ *names β*
| *names* (*MFOTL.Exists ψ*) = *names ψ*
| *names* (*MFOTL.Prev I ψ*) = *names ψ*
| *names* (*MFOTL.Next I ψ*) = *names ψ*
| *names* (*MFOTL.Since α I β*) = *names α* ∪ *names β*
| *names* (*MFOTL.Until α I β*) = *names α* ∪ *names β*

fun *gen_unique* :: *'a MFOTL.formula* ⇒ *bool* **where**
gen_unique (*MFOTL.Pred* _ _) = *True*
| *gen_unique* (*MFOTL.Eq* (*MFOTL.Var* _) (*MFOTL.Const* _)) = *False*
| *gen_unique* (*MFOTL.Eq* (*MFOTL.Const* _) (*MFOTL.Var* _)) = *False*
| *gen_unique* (*MFOTL.Eq* _ _) = *True*
| *gen_unique* (*MFOTL.Neg ψ*) = *gen_unique ψ*
| *gen_unique* (*MFOTL.Or α β*) = (*gen_unique α* ∧ *gen_unique β* ∧ *names α* ∩ *names β* = {})
| *gen_unique* (*MFOTL.Exists ψ*) = *gen_unique ψ*
| *gen_unique* (*MFOTL.Prev I ψ*) = *gen_unique ψ*
| *gen_unique* (*MFOTL.Next I ψ*) = *gen_unique ψ*
| *gen_unique* (*MFOTL.Since α I β*) = (*gen_unique α* ∧ *gen_unique β* ∧ *names α* ∩ *names β* = {})
| *gen_unique* (*MFOTL.Until α I β*) = (*gen_unique α* ∧ *gen_unique β* ∧ *names α* ∩ *names β* = {})

lemma *sat_inter_names_cong*: (⋀*e*. *e* ∈ *names φ* ⟹ {*xs*. (*e*, *xs*) ∈ *E*} = {*xs*. (*e*, *xs*) ∈ *F*}) ⟹
MFOTL.sat (*map_Γ* (λ*D*. *D* ∩ *E*) *σ*) *v i φ* ⟷ *MFOTL.sat* (*map_Γ* (λ*D*. *D* ∩ *F*) *σ*) *v i φ*
⟨*proof*⟩

lemma *matches_in_names*: *MFOTL.matches v φ x* ⟹ *fst x* ∈ *names φ*
⟨*proof*⟩

lemma *unique_names_matches_absorb*: *fst x* ∈ *names α* ⟹ *names α* ∩ *names β* = {} ⟹
MFOTL.matches v α x ∨ *MFOTL.matches v β x* ⟷ *MFOTL.matches v α x*
fst x ∈ *names β* ⟹ *names α* ∩ *names β* = {} ⟹
MFOTL.matches v α x ∨ *MFOTL.matches v β x* ⟷ *MFOTL.matches v β x*
⟨*proof*⟩

definition *mergeable_envs* **where**
mergeable_envs n S $\longleftrightarrow$ ($\forall$ *v1*$\in$*S*. $\forall$ *v2*$\in$*S*. ($\forall$ *A B f*.
($\forall$ *x*$\in$*A*. *x* < *n* $\wedge$ *v1* ! *x* = *f x*) $\wedge$ ($\forall$ *x*$\in$*B*. *x* < *n* $\wedge$ *v2* ! *x* = *f x*) $\longrightarrow$
($\exists$ *v*$\in$*S*. $\forall$ *x*$\in$*A* $\cup$ *B*. *v* ! *x* = *f x*)))

lemma *mergeable_envsI*:
assumes $\bigwedge$*v1 v2 v*. *v1* $\in$ *S* $\Longrightarrow$ *v2* $\in$ *S* $\Longrightarrow$ *length v* = *n* $\Longrightarrow$ $\forall$ *x* < *n*. *v* ! *x* = *v1* ! *x* $\vee$ *v* ! *x* = *v2* ! *x* $\Longrightarrow$ *v* $\in$ *S*
shows *mergeable_envs n S*
⟨*proof*⟩

lemma *in_listset_nth*: *x* $\in$ *listset As* $\Longrightarrow$ *i* < *length As* $\Longrightarrow$ *x* ! *i* $\in$ *As* ! *i*
⟨*proof*⟩

lemma *all_nth_in_listset*: *length x* = *length As* $\Longrightarrow$ ($\bigwedge$*i*. *i* < *length As* $\Longrightarrow$ *x* ! *i* $\in$ *As* ! *i*) $\Longrightarrow$ *x* $\in$ *listset As*
⟨*proof*⟩

lemma *mergeable_envs_listset*: *mergeable_envs* (*length As*) (*listset As*)
⟨*proof*⟩

lemma *mergeable_envs_Ex*: *mergeable_envs n S* $\Longrightarrow$ *MFOTL.nfv* α $\leq$ *n* $\Longrightarrow$ *MFOTL.nfv* β $\leq$ *n* $\Longrightarrow$
($\exists$ *v*′$\in$*S*. $\forall$ *x*$\in$*fv* α. *v*′ ! *x* = *v* ! *x*) $\Longrightarrow$ ($\exists$ *v*′$\in$*S*. $\forall$ *x*$\in$*fv* β. *v*′ ! *x* = *v* ! *x*) $\Longrightarrow$
($\exists$ *v*′$\in$*S*. $\forall$ *x*$\in$*fv* α $\cup$ *fv* β. *v*′ ! *x* = *v* ! *x*)
⟨*proof*⟩

lemma *in_set_ConsE*: *xs* $\in$ *set_Cons A As* $\Longrightarrow$ ($\bigwedge$*y ys*. *xs* = *y* # *ys* $\Longrightarrow$ *y* $\in$ *A* $\Longrightarrow$ *ys* $\in$ *As* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *mergeable_envs_set_Cons*: *mergeable_envs n S* $\Longrightarrow$ *mergeable_envs* (*Suc n*) (*set_Cons UNIV S*)
⟨*proof*⟩

lemma *slice_Exists*: *MFOTL_slicer.slice* (*MFOTL.Exists* φ) *S* σ = *MFOTL_slicer.slice* φ (*set_Cons UNIV S*) σ
⟨*proof*⟩

lemma *image_Suc_fvi*: *Suc* ‘ *MFOTL.fvi* (*Suc b*) φ = *MFOTL.fvi b* φ − {*0*}
⟨*proof*⟩

lemma *nfv_Exists*: *MFOTL.nfv* (*MFOTL.Exists* φ) = *MFOTL.nfv* φ − *1*
⟨*proof*⟩

lemma *set_Cons_empty_iff*[*simp*]: *set_Cons A Xs* = {} $\longleftrightarrow$ *A* = {} $\vee$ *Xs* = {}
⟨*proof*⟩

lemma *unique_sat_slice_mem*: *safe_formula* φ $\Longrightarrow$ *gen_unique* φ $\Longrightarrow$ *S* $\neq$ {} $\Longrightarrow$
mergeable_envs n S $\Longrightarrow$ *MFOTL.nfv* φ $\leq$ *n* $\Longrightarrow$
MFOTL.sat (*MFOTL_slicer.slice* φ *S* σ) *v i* φ $\Longrightarrow$ $\exists$ *v*′$\in$*S*. $\forall$ *x*$\in$*fv* φ. *v*′ ! *x* = *v* ! *x*
⟨*proof*⟩

lemma *unique_sat_slice*:
assumes *formula*: *safe_formula* φ *gen_unique* φ
and *restr*: *S* $\neq$ {} *mergeable_envs* (*MFOTL.nfv* φ) *S*
and *sat_slice*: *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) *v i* φ
shows *MFOTL.sat* σ *v i* φ

⟨*proof*⟩

7.2.8 Lemma 3

lemma (**in** *splitting_strategy*) *unique_sat_strategy*:
safe_formula $\varphi \Longrightarrow$ *gen_unique* $\varphi \Longrightarrow$ *slice_set* $k \neq \{\} \Longrightarrow$
mergeable_envs (*MFOTL.nfv* φ) (*slice_set* k) $\Longrightarrow$
MFOTL.sat (*MFOTL_slicer.slice* φ (*slice_set* k) σ) (*map the* v) i $\varphi \Longrightarrow$
ok φ $v \Longrightarrow k \in$ *strategy* v
⟨*proof*⟩

locale *skip_inter* = *joint_data_slicer* +
assumes *nonempty*: *slice_set* $k \neq \{\}$
and *mergeable*: *mergeable_envs* (*MFOTL.nfv* φ) (*slice_set* k)
begin

7.2.9 Definition of J'

definition *skip_joiner* = $(\lambda s.\ \bigcup k.\ s\ k)$

7.2.10 Theorem 3

lemma *skip_joiner*:
assumes *safe_formula* φ *gen_unique* φ
shows *joiner* $(\lambda k.\ M\ (splitter\ \pi\ k)) =$ *skip_joiner* $(\lambda k.\ M\ (splitter\ \pi\ k))$
(**is** *?L* = *?R*)
⟨*proof*⟩

sublocale *skip_joint_monitor*: *MFOTL_monitor* φ
$\lambda\pi.$ (*if safe_formula* $\varphi \wedge$ *gen_unique* φ *then skip_joiner else joiner*) $(\lambda k.\ M\ (splitter\ \pi\ k))$
⟨*proof*⟩

end

References

[1] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[2] D. Basin, F. Klaedtke, and E. Zălinescu. The MonPoly monitoring tool. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.

[3] J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 2020. To appear. Preprint at http://people.inf.ethz.ch/trayteld/papers/sttt20-som_long/som_long.pdf.

[4] J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *RV 2019*, volume 11757 of *LNCS*, pages 310–328. Springer, 2019.